BitShred: Fast, Scalable Malware Triage

Jiyong Jang, David Brumley, Shobha Venkataraman

November 5, 2010

CMU-CyLab-10-022

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

BitShred: Fast, Scalable Malware Triage *

Jiyong Jang and David Brumley
Carnegie Mellon University
{jiyongj, dbrumley}@cmu.edu

Shobha Venkataraman
AT&T Research Labs
shvenk@research.att.com

Abstract

The sheer volume of new malware found each day is enormous. Worse, current trends show the amount of malware is doubling each year. The large-scale volume has created a need for automated large-scale triage techniques. Typical triage tasks include clustering malware into families and finding the nearest neighbor to a given malware.

In this paper we propose efficient techniques for large-scale malware triage. At the core of our work is *BitShred*, a framework for data mining features extracted by existing per-sample malware analysis. BitShred uses a probabilistic data structure created through feature hashing for large-scale correlation that is agnostic to per-sample malware analysis. BitShred then defines a fast variant of the Jaccard similarity metric to compare malware feature sets. We also develop a distributed version of BitShred that is optimal: given 2x more hardware, we get 2x the performance. After clustering, BitShred can go one step further than previous similar work and also automatically discover semantic inter-family and inter-malware distinguishing features, based upon co-clustering techniques adapted to BitShred's fingerprints. We have implemented and evaluated BitShred using two different per-sample analysis routines: one based upon static code reuse detection and one based upon dynamic behavior analysis. Our evaluation show BitShred's probabilistic data structure and algorithms speed up typical malware triage tasks by up to three orders of magnitude and use up to 82x less memory, all with similar accuracy to previous approaches.

1 Introduction

The volume of new malware, fueled by easy-to-use malware morphing engines, is growing at an exponential pace [9]. In 2009 security vendors received upwards of 8,000 unique by hash malware samples per day [9], 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 triaging to sort it by family, functionality, and other qualities. Unfortunately, scaling malware triage to current and future volumes is challenging. For example, can we automatically cluster 1,000,000 malware samples into families? For each new sample, can we identify a previously analyzed sample that is most similar? Can we automatically determine what features distinguish all members of one malware family from another?

In principle we know how to address such questions. At a high level, there are two steps. First, per-sample malware analysis is run on each sample to extract a set of features. For example, dynamic analysis may report observed behaviors as features, while static analysis may report common code segments. Scaling this part of the analysis is easy, e.g., we can distribute per-malware feature extraction across many nodes. Second, we need to be able to perform pairwise comparisons between malware feature sets. Comparisons are the core for fundamental malware triage solutions such as clustering to determine malware families, finding nearest neighbors based upon features, or determining what features may be different between samples.

There are several requirements for an effective, scalable malware triage system. First, the overall approach should be agnostic to the particular per-sample malware analysis feature extraction in step 1. 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. We need comparison techniques that allow us to plug-in the latest or most appropriate analysis for feature extraction. Second, the pairwise malware comparisons (which we also refer to as *distance computations*) must be extremely fast. Clustering techniques for identifying families require $s(s-1)/2$ pairwise comparisons for s samples. While there are data reduction techniques that can sometimes reduce s before clustering (e.g., locality sensitive hashing in § 6 is one such technique), the quadratic pairwise comparison bound after the reduction is fundamental [11]. To get a sense of scale, clustering 1,000,000 malware requires about 10^{12} comparisons. Beyond simple volume of malware, the high-dimensionality of extracted features by evermore detailed per-sample analysis makes typical distance computations too slow to handle current and future volumes. Current techniques do not scale to such volumes. Third, we need techniques that are parallelizable so that we can make full use of multi-core

*This technical report is a follow-up work of CMU-CyLab-10-006.

systems and distributed platforms such as Hadoop [1]. Finally, we need malware data mining techniques that offer semantic insights such as why malware are similar or different, as well as what are the distinguishing features. Existing data mining algorithms for malware analysis [5, 13] act as black boxes and provide no such semantic information.

In this paper, we propose a set of novel techniques for addressing large-scale malware triage in a system called *BitShred*. BitShred is unique in that it uses feature-based hashing for malware clustering and semantic inter-malware feature correlation. At a high level, BitShred hashes the features output by the per-malware analysis into a probabilistic data-structure called a *fingerprint*. The fingerprint encapsulates even high-dimensional feature sets into a data structure that allows large-scale inter-sample analysis to be performed cheaply. BitShred then computes similarity between malware pairs based upon an efficient probabilistic approximation algorithm for the Jaccard distance. The intuition for the Jaccard is that the degree of similarity is relative to the percentage of shared features, e.g., malware that share most of the same features are more similar than malware that do not. Our algorithm well-approximates the Jaccard index for any malware feature set, but is significantly faster.

BitShred is the first algorithm for malware triage that meets all the above desired challenges and requirements. First, by using feature hashing [26, 28], our approach is agnostic to the particular per-malware analysis routine, even when the extracted feature set has very high dimensions. We show this empirically by demonstrating BitShred on two previously proposed per-sample analysis: dynamic behavior analysis from Bayer *et al.* [5], and static code reuse detection as proposed in [3, 17, 27]. Second, BitShred is over 1000 *times* faster than existing approaches at comparing malware, while simultaneously requiring less memory. Third, we develop a parallel version of BitShred that is optimal in terms of inter-node communication, currently implemented on top of Hadoop. We show given $2x$ the hardware, BitShred can scale to approximately $2x$ the malware.

Finally, BitShred identifies semantic information within and between identified families based upon the idea of co-clustering. Roughly speaking, given a clustering, co-clustering determines which features led to the conditions for creating a particular malware cluster. For example, suppose there are two different malware families, both of which use libc, but one family lists all files and another deletes all files. The distinguishing feature is not using libc, but the particular behavior of deleting or listing files. Co-clustering would automatically determine this. Current state-of-the-art techniques would not, and would rely on someone a priori excluding the libc analysis. While libc is a simple example, co-clustering does this for any set of features. As a side benefit, co-clustering also performs a dimensionality reduction in our feature set because all features identified as non-informative can be removed.

Returning to our questions, given a million new malware, which malware is unique and which is a variant of a known family? BitShred performs automatic clustering which identifies malware families. In our experiments we show BitShred's probabilistic approach is up to three orders of magnitude faster than current approaches while enjoying similar accuracy to a deterministic, exact clustering. Our guarantees are independent of the malware itself. The overall effect of the speedup is we can cluster significantly more malware per day than existing approaches.

Given a particular sample, can we identify a previously analyzed sample that is the most similar? Hu *et al.* [13] proposed this as the nearest neighbor problem for malware. BitShred is able to find nearest neighbors again an order of magnitude faster. However, BitShred is the first large-scale malware system that can go one step further and provide semantic information about *inter-family* and *inter-sample* distinguishing features. We achieve this by developing co-clustering [8, 21] techniques based on BitShred fingerprints.

Contributions. In summary, this paper makes the following contributions.

1. *Higher overall throughput with the same accuracy.* A key factor in computer science is finding the right data structure and algorithms for the job. BitShred's fingerprint is 68-1890 times faster than previous approaches without an appreciable loss of accuracy.

2. *Proof of correctness.* We prove that BitShred's distance calculation well-approximates the Jaccard index (i.e., is within epsilon) for any feature sets (Theorem 1). § 9 details why this is also of independent interest.

3. *Per-Malware Analysis Agnostic.* Our hash-based approach is independent of the particular per-malware analysis engine, and works even for high-dimensional feature sets. For example, BitShred works on the feature space from 2^{17} (dynamic analysis from Bayer *et al.* [5]) to 2^{128} (static code reuse detection).

4. *Parallelizable.* We develop a parallelized version of our algorithm (§ 4). The parallelized version (a) extends single-node comparisons to a divide-and-conquer approach, (b) is optimal in that there is no inter-node communication needed, (c) works in MapReduce frameworks [10].

5. We implement our approach and perform extensive experimental analysis using two different previously proposed per-sample analysis: code similarity and dynamic behaviors.

6. We propose techniques based on co-clustering using BitShred's fingerprints for identifying semantic information within and between malware families.

2 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. BitShred's job is to facilitate fast and scalable inter-sample analysis even when there is a very large feature set. BitShred uses existing analysis to perform per-sample feature extraction. For example, Karim *et al.* have proposed using code fragments as features [15], where two malware are considered similar if they share a large amount of code. In a dynamic analysis setting, a feature corresponds to whether a particular behavior was exhibited or not, and malware that exhibit similar behaviors are considered similar. We have implemented both and performed experimentation, though our techniques are more generally applicable.

Many malware triage problems are, at core, a problem of quick pairwise malware feature set comparisons. For example, we can automatically identify malware families by clustering malware based upon similarity. During clustering, there will be many pairwise comparisons. For example, hierarchical clustering has a lower bound of $s(s-1)/2$ comparisons for s malware to cluster [11]. While there are data reduction techniques that reduce the size of s, e.g., locality sensitive hashing (§ 6), we can expect that even after data reduction the number of malware we need to cluster will continue to increase rapidly. Another example is automatically identifying the nearest neighbors, which requires that given a sample m we compute its distance to all other malware. An exacerbating issue is that we want analysis which extracts many features, which in turn creates extremely high-dimensional feature sets, and this, in turn makes each comparison more expensive.

The main idea in BitShred is to use feature hashing to compactly represent even high-dimensional feature sets in a bitvector. We call the bitvector the malware *fingerprint*. The algorithm that calculates the malware fingerprint using feature hashing is called BITSHRED-GEN in Figure 1. We then replace existing exact inter-malware feature set comparison called the Jaccard index with an approximation algorithm called BITSHRED-COMPARE that is just as accurate (with high probability), yet significantly faster. In particular, the main bottleneck with the Jaccard distance computation is it requires a set intersection and union operation with the entire feature space. BitShred's algorithm replaces set operations with bitvector operations, which are orders of magnitude more efficient.

We then perform clustering using BITSHRED-CLUSTER to identify families. We then extend the idea to cluster not just malware, but also to perform semantic analysis to determine which features distinguish identified malware families with BITSHRED-SEMANTIC. These ideas are based upon the idea of co-clustering, where we cluster together features and malware to identify features that matter for a particular family. Why do we do both clustering and co-clustering? Co-clustering is more expensive because it must consider both what features are in common between malware pairs, as well as what features are important. Hierarchical clustering is faster since it only needs to determine whether malware is similar or not. Thus, we run hierarchical clustering to identify families, and co-clustering to identify inter-family and intra-family semantic features. For example, in our experiments co-clustering automatically identifies that within the Allaple malware family a distinguishing feature is the particular IP address contacted.

3 Single Node BitShred

In this section we describe the core BitShred components: BITSHRED-GEN, BITSHRED-COMPARE, and BITSHRED-CLUSTER. In § 4, we show how the algorithm can be parallelized, e.g., to run on top of Hadoop or multi-core systems, and in § 5 co-clustering techniques for identifying distinguishing features. Figure 1 shows the overall flow between components. Throughout this paper we use m_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 m_i.

BITSHRED-GEN: $G \rightarrow F$ BITSHRED-GEN is an algorithm from the extracted feature set $g_i \in G$ to fingerprints $f_i \in F$ for each malware sample m_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 has function $h : \chi \rightarrow 0, 1^m$ where the domain χ is the domain of possible features and m is the length of the bitvector. We use djb2 [6] and reduce the result modulo m. (As we will see in § 6, data reduction techniques such as locality-sensitive hashing [5] and Winnowing [24] can be used to pare down the data set for which we call BITSHRED-GEN and perform subsequent steps.)

BITSHRED-COMPARE: $F \times F \rightarrow \mathbb{R}$ BITSHRED-COMPARE computes the similarity $d \in [0, 1]$ between fingerprints f_a and f_b. A similarity value of 1 means the two samples are identical, while a similarity of 0 means

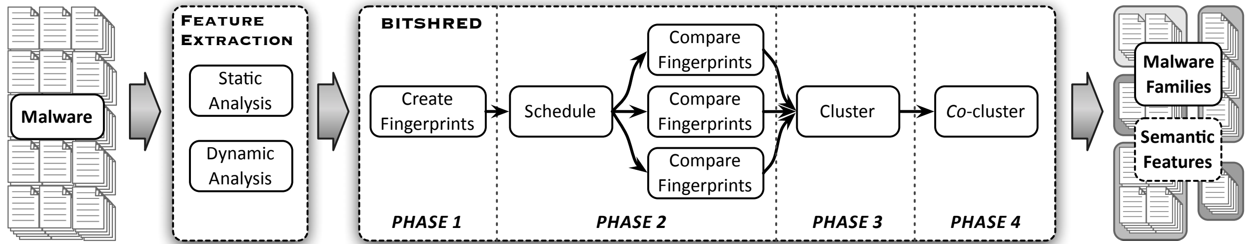


Figure 1: BitShred Overview

the two samples share nothing in common (in our setting, this means they share no features in G).

Our similarity metric is a fast variant of the Jaccard index designed to work on bit-vectors. The intuition behind the Jaccard index is that the similarity of malware is the percentage of shared features. The Jaccard index J of two feature sets g_a and g_b is defined as:

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

The numerator calculates the amount of shared features between two samples m_a and m_b, and the denominator is the total amount of features.

In BitShred, we approximate the Jaccard index with a faster-to-compute approximation using bitvector arithmetic. The Jaccard set operations are slow compared to bitvector arithmetic because they require a duplicate check (because sets cannot have duplicates), and because set representations exhibit poor L1/L2 cache behavior. BitShred calculates:

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

where $S(\cdot)$ means the number of bits set to 1. While an observant reader may notice that the BitShred fingerprint resembles a Bloom filter, we emphasize that the BITSHRED-COMPARE operation is not a Bloom filter operation. Bloom filter operations are set membership tests; here we want to approximate a similarity metric. It turns out that this difference means that the set of parameters we choose is quite different, e.g., all things being equal, BitShred's accuracy improves with fewer hash functions, but Bloom filters improve with more hash functions.

Formally, the following theorem states that BITSHRED-COMPARE 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

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

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

We defer the proof of the theorem to 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 standard for Bloom filters, but instead are of how well our feature hashing data structure lets us approximate the Jaccard index.

BITSHRED-CLUSTER: $(F \times F \times \mathbb{R}\ \textbf{list}) \times \mathbb{R} \rightarrow C$ 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. See § 8 for our experiments for different values of t.

BitShred currently uses an agglomerative hierarchical clustering algorithm to produce malware clusters. Initially each malware sample m_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., $\text{BITSHRED-COMPARE}(c_A, c_B) = \max\{\text{BITSHRED-COMPARE}(f_i, f_j) | f_i \in c_A, f_j \in c_B\}$. 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. We discuss the co-clustering step in detail in Section 5.

Properties of Single Node BitShred

Accuracy. Although BITSHRED-COMPARE is probabilistic, Theorem 1 proves it closely approximates the Jaccard index. While an attacker could certainly try and manipulate the per-sample analysis, an attacker cannot affect the accuracy of BitShred's feature hashing as long as the hash function is either unknown or collision-resistant.

High Throughput. BITSHRED-COMPARE uses simple bit-vector arithmetic that can be computed in constant time. In practice, constant-sized bit-vector arithmetic is L1/L2 cache-friendly, while typical algorithms over sets with different sizes is not. Our performance is enhanced significantly by this cache-friendliness as shown in Figure 3. Further, BitShred's fixed-size fingerprint size typically leads to a smaller memory footprint, as shown in § 8.

4 Distributed BitShred

BitShred's throughput, as well as any clustering algorithm, is bottlenecked by how quickly fingerprints can be compared. In addition to improved single-node performance, we have developed a distributed version of BitShred based upon Hadoop. The distributed versions performance is designed to improve linearly with the amount of additional hardware resources. In order to improve linearly, we address two challenges. First, can we design an algorithm that does not require cross-node communication? Second, can we develop an algorithm where no node is a bottleneck, i.e., all nodes do the same amount of work? In this section we describe how the BITSHRED-SCHEDULE algorithm optimally parallelizes BitShred to achieve both goals, as well as how the parallelization can be implemented in the MapReduce framework.

4.1 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-COMPARE in a resource and communication-efficient manner requires more thought. There are $s(s-1)/2$, and every comparison takes the same fixed time, so if every node does $s(s-1)/2r$ comparisons all nodes do equal work.

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.

Thus, BITSHRED-SCHEDULE ensures that the comparisons are evenly divided among the nodes, all nodes do equal work, and there is no inter-node communication required. Of course there may be other protocols that achieve these goals; we use this because it is a simple and optimal algorithm that meets our goals.

4.2 BitShred on Hadoop

Our distributed implementation uses the Hadoop implementation of MapReduce [1, 10]. 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. We also use DistributedCache feature to copy the necessary (read-only) files to the nodes prior to executing a job. This allows nodes to work from a local cached copy of data instead of continually fetching items over the network.

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

1. MAP: $\langle K_i, m_i\rangle$ list $\rightarrow \langle K_i, f_i\rangle$ list. Each MAP task is assigned the subset of malware samples m_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-COMPARE across all Hadoop nodes by defining the following functions:

1. MAP: $\langle K_i, f_i \rangle$ list $\rightarrow \langle \mathbb{R}, (m_a, m_b) \rangle$ list MAP tasks read fingerprint data files created during phase 1 and runs BITSHRED-COMPARE on each fingerprint pair, outputting the similarity $d \in \mathbb{R}$.

2. REDUCE: $\langle \mathbb{R}, (m_a, m_b) \rangle$ list $\rightarrow$ sorted $\langle \mathbb{R}, (m_a, m_b) \rangle$ 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 essentially the agglomerative single linkage clustering. In particular, malware m_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.

5 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 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 too 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 m_1 and m_3 are in one family, and m_2, m_4, and m_5 are in another family. The column clusters say the distinguishing features between the two families are features 2, 3, and 5.

5.1 Co-clustering Algorithm in Bitshred

We have adapted the cross-associations algorithm [8] redesigned for the Map-Reduce framework [21] to BitShred fingerprints. The basic steps are row iterations and

[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.

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

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

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 [21]).

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. To our knowledge, our results are the first to demonstrate that if co-clustering algorithms can be combined with the appropriate feature hashing functions, they can still extract the underlying structure of the data accurately.

6 Practical Data Reduction Techniques

In this paper so far we have explored techniques that provide an exact ranking between all pairs of malware. For example, we must compute the distance between all pairs since if we omitted a pair we would not know how they relate, and the overall result would be approximate. 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.

In practice there are two ways to reduce complexity in order to increase speed at potentially the cost of some precision. First, one can omit some pairwise comparisons of malware sample. Removing pairs reduces the value of s for the $\approx s^2$ comparisons during clustering. Second, one could remove some malware features so that each comparison is faster. Doing so makes each pairwise comparison faster. We have explored both data reduction techniques in BitShred.

One practical way to reduce the number of comparisons is to partition the malware data set, and then compute the clustering among the partitions. Suppose we are given s samples, and we can partition it into t groups. We then compute t independent clusterings, one for each partition. Assume for simplicity that each partition is approximately the same s/t size, then the total number of comparisons is $\approx t \cdot (\frac{s}{t})^2$ vs. $\approx s^2$ without partitioning. The overall speedup is: $s^2/(t(\frac{s}{t})^2) = t$. For example, if we partition the malware into 4 groups, we get a factor of 4 speedup. There are a number of ways to perform the rough partitioning, e.g., using locality-sensitive hashing [5], running k-means first with some small k value, etc.

We note that another complementary way to reduce the number of comparisons is to prune near duplicates from the data set first. Bayer *et al.* propose a technique to do this based upon locality sensitive hashing (LSH) (this should not be confused with feature hashing) [5]. In LSH, we compute a hash function $lsh(m_i)$ for all s malware, and throw away duplicates. The hash function is designed so that if m_i is "close" to m_j, $lsh(m_i) = lsh(m_j)$. The result is some set of malware $s' \leq s$ of unique-by-hash malware which is then passed to standard hierarchical clustering.

Directly reducing the number of features has traditionally been more tricky and required one to think about what features matter *a priori*. For example, suppose we use the amount of code shared as a similarity metric [24, 27]. Let w be a window of code measured in some way, e.g., w statements, w consecutive n-grams, etc. Schleimer *et al.* propose a winnowing algorithm (which is a type of fuzzy hashing) that guarantees that at least one shared unit of code will in any window of length at least $w + n - 1$ will be included in the feature set [24]. The result is we select a factor of w fewer features, which in turn means that each feature set comparison is much more efficient.

We can also perform both feature reduction and malware reduction simultaneously, to get even more substantial data reduction for the clustering. This insight has led us to perform a rough co-clustering to identify rough malware family partitioning, along with the feature set for that family. For example, in our experiments we co-cluster into 7 partitions, and then perform full hierarchical clustering on all of those individual partitions, we get a 7x speedup in comparison to clustering over the entire malware dataset. We have also implemented Winnowing, and found it gives us a 2x speedup. Although there is no guarantee that the results will be optimal compared to an exact clustering, we have performed empirical measurements that show we get the speedup with a negligible

loss of accuracy.

7 Implementation

We have implemented single-node BitShred in 2K 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 [16] for visualizing the clustering and family trees generated as shown in Figure 7, 8.

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 $(m_i, m_{i+1}, \cdots m_{i+2047})$ 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.

8 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.

8.1 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 [3, 17, 27]. 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). Using other features such as basic blocks, etc. are all possible 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. Pragmatically, n-gram analysis also has the advantage of not require 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. Clustering using Jaccard took about 9.5 hours, and required 644.13MB of memory. This works out to about 15 malware comparisons/sec and 1,593 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 303 times faster than with exact Jaccard. With 32KB fingerprints, BitShred runs about 2 times faster compared to 64KB fingerprints, and about 577 times faster than exact Jaccard.

Since BitShred is a probabilistic data structure created through feature hashing, hash collisions may impact the

	Size of fingerprints	Time to compare between 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	9h 26m 59s	-	-	15	1,593
BS64K	62.50MB	1m 50s	0.0199	0.0017	4,541	28,012
BS32K	31.25MB	58s	0.0403	0.0050	8,657	38,677
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	58s	0.0159	0.0009	8,657	38,677
BS32K (W12)	31.25MB	58s	0.0062	0.0039	8,657	38,677
BS8K (W4)	7.81MB	18s	0.0649	0.0086	27,750	69,247
BS8K (W12)	7.81MB	18s	0.0247	0.0016	27,750	69,247

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

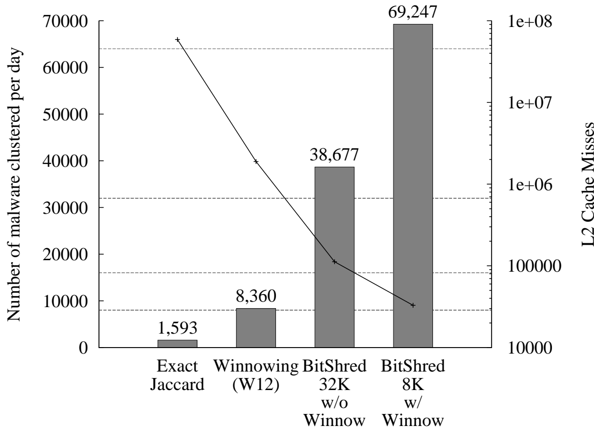


Figure 3: Overall malware cluster-per-day capabilities. We also report relative L1/L2 cache misses.

accuracy of the 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. 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.

We also applied the data reduction techniques described in Section 6. Winnowing is especially relevant because it is guaranteed to be within 33% of an upper bound on performance algorithm [24], and is currently the fastest from a theoretic sense we are aware of.[2] We compare 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 and 12.

BitShred beats straight Winnowing. BitShred is anywhere from 11-43 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 8K, allowing all 1,000 samples to be clustered in 18 seconds.

Figure 3 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 3 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 uses 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 4 shows the BITSHRED-GEN fingerprint generation time. In this experiment, we utilized 80 map

[2] Winnowing is perhaps better known as the Moss plagiarism detection tool.

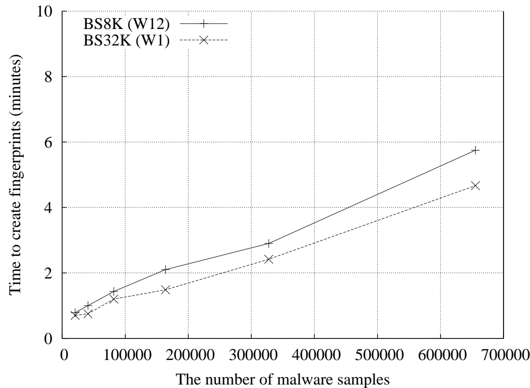


Figure 4: Performance of Distributed BitShred (Creating Fingerprints)

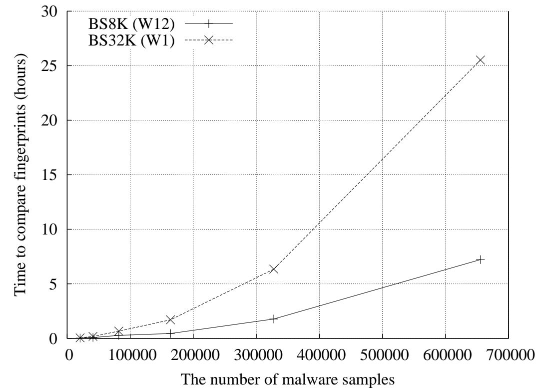


Figure 5: Performance of Distributed BitShred (Comparing Fingerprints)

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 5 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 26m 41s with BS8K (W12), which works out to 8,383,317 comparisons/second. 327,680 samples requires about 5.4×10^{10} comparisons, and takes 1h 46m 25s with BS8K (W12), which works out to a similar 8,408,289 comparisons/sec.

Overall, the distributed version achieved a pairwise comparison throughput of about 7.2×10^{11} per second. This works out to clustering about 1,205,000 malware clustered per day.

Data Reduction In addition to experimenting with Winnowing as a feature reduction technique, we also experimented with using the co-clustering data reduction technique from § 6. We performed the rough partitioning by running the co-clustering algorithm for 10 iterations on the 131,072 samples. The result was 7 partitions, for which we then measured the total number of comparisons for performing hierarchical clustering. The resulting number of comparisons dropped by 80% overall.

Triage Tasks Three common triage tasks are to automatically identify malware families via clustering [5], to identify the nearest neighbors to a particular malware sample, and to visualize malware by creating phylogenetic trees [15]. In this experiment we explore using BitShred for both using 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 Section 8.2 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.

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 unpacked 131,072 malware samples using off-the-shelf unpackers. We then clustered the malware based upon n-grams and compared the identified families to a reference clustering using ClamAV labels. Figure 6 shows the overall results with BitShred with Winnowing (BS32K (W12) where the window size is 12). Surprisingly, simple n-gram analysis did quite well. 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 less than 1.5 hour with 256 map tasks.

Nearest Neighbor. Hu *et al.* describe finding the nearest k-neighbors to a given sample as a common triage

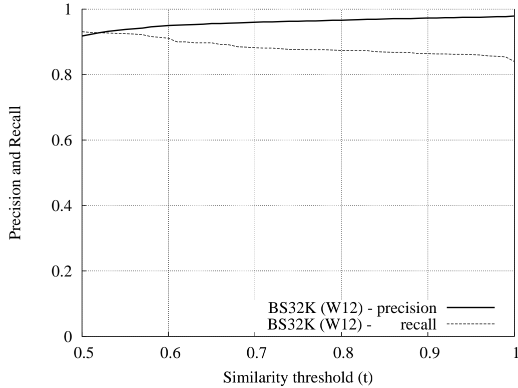


Figure 6: Precision and Recall.

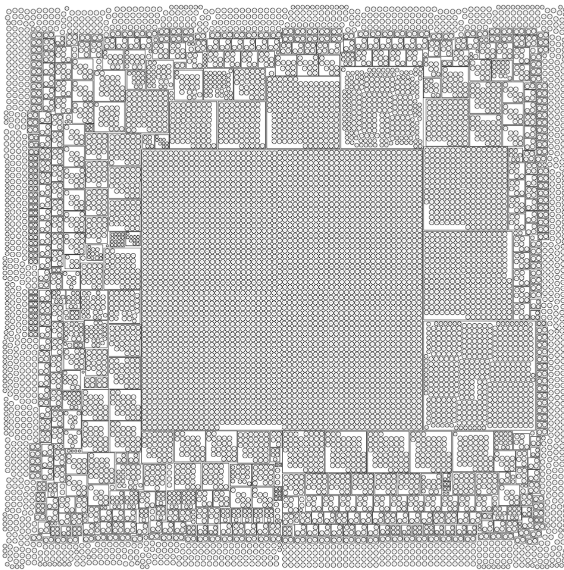


Figure 7: Clustering graph when $t = 0.85$

task [13]. 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% precison 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 graphs where each square represents a malware family, with circles representing individual samples. Figure 7 shows a clustering of 9,404 malware samples when $t = 0.85$. [3] In the figure we can see one large family 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.

[3] We pick 9,404 samples because larger numbers created graphs that hung our, and potentially the reviewers', PDF reader.

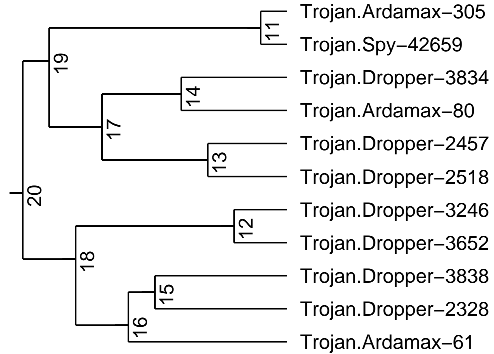


Figure 8: Lineage tree for a single malware family

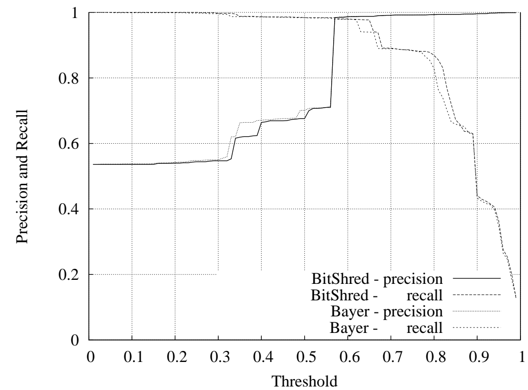


Figure 9: Clustering Quality based upon behavior profiles

Another way to visualize the results using BitShred is to create phylogenetic family trees based upon similarity [15]. The more alike two malware samples are, the closer they are on the tree. Figure 8 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.

8.2 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., [4, 5, 19, 20, 23, 25]. 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.

We have performed experiments using BitShred for

clustering using 2658 dynamic profiles available from Bayer *et al.* [5]. These are the same samples used to measure accuracy in that paper. In this data set, each behavior profile is a list of feature index numbers. The total number of features was 172260, which is relatively small compared to static analysis. In our experiments, we used only a 1K fingerprint size since the number of features was relatively small.

An exact clustering took 16s and 86MB of memory using code from Bayer *et al.* . BitShred took 8s (2x as fast) and used 12MB of memory (7x less memory). The average error was 2% using the 1KB fingerprint. Figure 9 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.

8.3 Semantic Feature Information

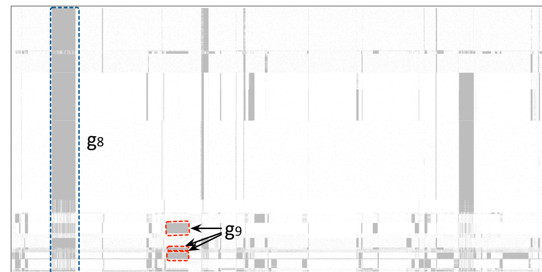
Finally, we used the co-clustering phase in BitShred to identify semantic distinguishing features among malware families. We performed a variety of experiments. Overall, we found that co-clustering automatically identified both inter-family and intra-family semantic features. Typical identified features 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 § 8.2. Figure 10a depicts the malware/feature matrix before co-clustering. We then co-clustered, which took 15 minutes.

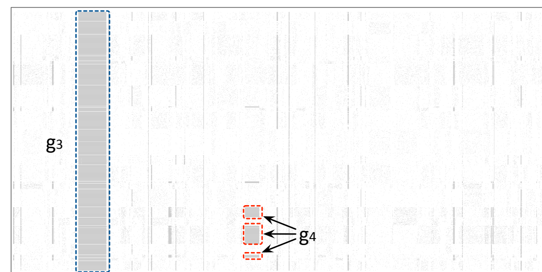
Figure 10b shows the complete results. The checkerboard pattern corresponds to the sub-matrices identified as being homogeneous, i.e., corresponding malware/feature pairs that are highly correlated. For example, the large dark sub-matrix labeled g_8 corresponds to the fact that most malware had the same 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 10c 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).

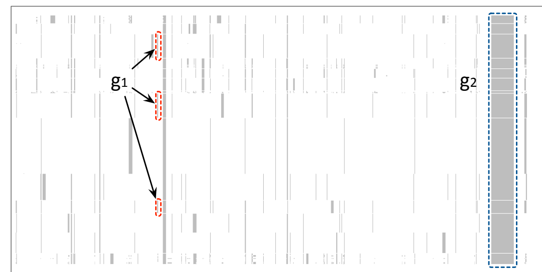
(a) A typical matrix before co-clustering.



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



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



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

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

Figure 10: Feature extraction by co-clustering. Grey dots represent 1 in the binary matrix, i.e., the presence of a feature.

Co-clustering of n-gram features We also experimented with co-clustering using the n-gram features. Figure 10d 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. Total co-clustering time 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 10e 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.

9 Discussion

Containment. BITSHRED-COMPARE 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:

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

when f_i is the fingerprint for malware m_i and $|m_a| \gg |m_b|$.

Additional applications. There are a number of other security applications for automatic binary code reuse detection. For example, BitShred can be used for plagiarism detection, similar to MOSS [24]. An immediate application is to find copyright violations, e.g., compile all GPL libraries and then use BitShred to check for GPL violations, as done in [12]. We leave exploring these scenarios as future work.

10 Related Work

We are not the first to propose the need for large-scale malware analysis and triage. Two recent examples are Hu *et al.* [13] and Bayer *et al.* [5]. BitShred's hash-based approach works well when the feature set is fixed, as shown in § 8 with Bayers dynamic analysis approach. Hu *et al.* use a different metric based upon the NP-complete problem of determining function call graph similarity. 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 (Section 6.3 in [13]). 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 our data set for finding nearest neighbors are reported in § 8.1.

Bayer *et al.* [5] use dynamic analysis, which is less vulnerable to some kinds of obfuscation, such as packing. They also perform locality sensitive hashing (LSH) on the features extracted by the analysis to reduce the number of comparisons. However, the $O(n^2)$ comparisons that may be required between the samples, together with variable-length feature representations, make their approach less scalable for large-scale malware analysis.

Li *et al.* [18] argue it is difficult to get ground truth for clustering accuracy. Our goal is different than addressed in that paper since our goal is to find a right data structure compactly representing even high-dimensional feature sets and to calculate similarity significantly fast, precisely. We stress that detecting non-trivial similarities between programs is equivalent to deciding whether two programs have a shared non-trivial property, thus undecidable in the general case. As a result, any automatic malware classification or clustering scheme is going to be a best effort.

Clustering, classification, and lineage tree generation have been studied in other work. BitShred fingerprints are motivated by using Bloom filters to difference sets (see [7]), but previous work has not shown that it can be used as an approximation for Jaccard like here. Kolter and Maloof [17] suggested a classification method based upon 4-grams. Karim *et al.* [15] proposed a malware phylogeny generation technique using n-perms to match

every possible permuted code. Perdisci *et al.* [22] presented a classification method between packed and non-packed PE files exploiting PE header information. Bailey *et al.* [4] proposed behavior-based malware classification and clustering technique. They define the behavior of malware in terms of system state changes, i.e., abstraction of system calls, and use normalized compress distance as a distance metric.

11 Conclusion

In this paper we have presented *BitShred*, a new approach to large-scale malware triage and similarity detection. At the core of BitShred is a probabilistic data structure based upon feature hashing. Our approach make inter-malware comparisons in typical large-scale triage tasks such as clustering and finding nearest neighbors up to three orders of magnitude faster than existing methods. 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 1,000,000 malware per day. In addition, we have develop 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.

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] 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*, pages 41–42, 2004.

[4] 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.

[5] 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*, February 2009.

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

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

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

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

[10] 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.

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

[12] A. Hemel and S. Coughlan. Binary analysis tool. http://www.binaryanalysis.org/.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

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

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

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

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

[23] 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.

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

[25] 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.

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

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

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

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 [14] 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 bloom filter obtained by computing the bitwise-or of the two fingerprints f_i and f_j is equivalent to the bloom filter 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 bloom filter 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 bloom filter 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:

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

As U is tightly concentrated around its expectation [7], 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:

$$E[I] = m\left(1 - 2\left(1 - \frac{1}{m}\right)^{kN} + \left(1 - \frac{1}{m}\right)^{k(2N-c)}\right). \quad (3)$$

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 2 and 3:

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

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.