Characterizing the Effect of Feature Normalization on Malware Coclustering

How does the information loss due to different feature normalization methods affect the output and the running time of BitShred's coclustering algorithm?

Team (Alphabetical Order)

David Brumley <dbrumley@cmu.edu>, Dominick Direnzo <ddirenzo@andrew.cmu.edu>, Vincent Huang <vincenth@andrew.cmu.edu>, Owen Kahn <okahn@andrew.cmu.edu>, Sunny Nahar <anahar@andrew.cmu.edu>, Maverick Woo <pooh@cmu.edu>

Table of Contents

Background

Notation

We try to follow the notation in the BitShred paper [1] when possible.

$N = n$	number of malware samples (#rows in feature & fingerprint matrices)
M	number of *observable* features (#cols in feature matrix)
M^*	number of *observed* features in the given data set
m	fingerprint size (#cols in fingerprint matrix)
w	sliding window size ("n" in n-gram)
h	hash function mapping from *[M]* to *[m]*
L	number of normalization levels used in our experiments
s_i	malware sample i
F_i	feature (row) vector of sample i (in practice this is never computed)
f_i	fingerprint of sample i
$FM(k,l)$	fingerprint matrix with k row groups and l column groups
k^*, l^*	number of row and column groups when coclustering finishes
T	homogeneity threshold, measured in encoding length of submatrix

Rationale

A value before data reduction is set in uppercase, and its counterpart after data reduction is set in lowercase. Global constants and matrices are set in uppercase. All other scalar values are set in lowercase. Measurement outcomes are denoted in the starred notation.

BitShred Recap

The key idea behind BitShred is ***feature hashing***, which exploits the sparseness of feature vectors encountered in many real-world applications. To make this document more self-contained, let us briefly review the concepts behind BitShred.

Features

Given a set of malware sample { s_i }, BitShred starts by extracting a set of features from each s_i. This can be done by, for example, sliding a window of w (e.g., 16) bytes over the sample's text section, which is where the executable code is located in a binary. Suppose the current content of the sliding window is x. We say that s_i has feature x.

Feature Vector

Conceptually, we imagine a sample s_i is represented as a feature (row) vector F_i whose length M is the number of observable features. In the example above, M = 2^128 since there are 128 bits in a sliding window of 16 bytes and there are 2^128 different 128-bit strings. A key observation that enables BitShred to scale in many applications is that the feature vectors encountered in those applications tend to be sparse, i.e., a large fraction of the entries in a feature vector are zero[1]. This is because the corresponding feature space has been chosen to be so large that most of the observable features are not present in the samples. One possible reason behind such a feature space choice would be to ensure that the features are intuitive.

Fingerprint

To exploit the sparseness of feature vectors, BitShred employs *feature hashing*, a popular data reduction technique from machine learning. The idea of feature hashing is as follows. Instead of representing a sample s_i as its feature vector F_i, we represent s_i using what we call its "fingerprint" f_i by choosing an appropriate hash function h that maps from the feature space [1..M] to [1..m], where m is the size of the range of h. Once h is chosen, we produce f_i by iterating through the features of s_i. Suppose a feature x is present in s_i. Without feature hashing, we would have marked the x-th bit in the hypothetical feature vector F_i; with feature hashing, we mark the $h(x)$-th bit in the fingerprint f_i instead.

Clustering

BitShred can perform (i) clustering and (ii) coclustering using fingerprints. The goal of a clustering task is to identify *clusters*: a subset of samples that exhibit similar behavior (contains / lacks) over the entire set of observable features. BitShred performs bottom-up clustering, a.k.a. agglomerative clustering. Initially, each sample is in its own cluster. Two clusters that are deemed similar will be linked together to form a bigger cluster. This linking process continues until only one cluster is left. The result is a dendrogram, a tree depicting the dependency order of cluster linkages. In this report, we do not consider clustering.

[1] This is without loss of generality since one may logically negate the meaning of each feature when there are more ones than zeros.

Coclustering

The goal of a coclustering task is to identify *coclusters*: a subset of samples that exhibit similar behavior (contains / lacks) over a subset of observable features *and* vice versa. BitShred performs coclustering by first constructing an initial *n*-by-*m* fingerprint matrix *FM(1,1)* whose *i*-th row is f_i. The result of coclustering is a grouping of the rows of *FM(1,1)* and another grouping of the columns of *FM(1,1)*. Suppose there are *k** row groups and *l** column groups when the coclustering algorithm is terminated. Observe that each row of *FM(1,1)* will be assigned to a row group ranging from 1 to *k** and each column a column group ranging from 1 to *l**. By laying out the rows and columns in the order of their respective group to produce *FM(k*,l*)*, each submatrix induced by a row and a column group is considered a cocluster in the output. Note that the exact ordering of the rows/columns within a row/column group is unimportant.

The goal of a coclustering algorithm is to generate the row and column groups such that each induced submatrix is deemed homogeneous. For example, we can require that each submatrix be highly compressible by some coding algorithm. For this report, it would be sufficient to think of this as specified by a threshold *T*. BitShred's algorithm explicitly maintains *FM(k,l)*. At each iteration of an outer loop, it tries to improve the homogeneity by rearranging the rows (or columns, depending on the iteration) while keeping the column groups (resp. row groups) steady. Once an inner stopping criteria is met, the next iteration commences after increasing the number of column groups (resp. row groups). The algorithm terminates when it satisfies an outer stopping criteria that the homogeneity of each induced submatrix surpasses *T*. (We refer the reader to the BitShred publication [1] for more details.)

Tradeoff Between Coclustering Quality and Speed

When using the feature hashing technique to reduce data size, the key is to choose a right hash function for the task. Since high-performance and well-behaved hash functions are readily available (BitShred uses an algorithm known as djb2[2]), our choice essentially boils down to picking the range *m* of the hash function. As the feature vectors in our task tend to be very sparse in practice, *m* can be chosen to be much smaller than *M*. A reasonable first try would be to let *m* equal to a small multiple of the number of *observed* features *M**.

Tradeoff

Unfortunately, there is a difficult tradeoff in picking the exact value of *m* due to the presence of hash collisions.[3] Ideally, we would like to pick an *m* that is large enough for our data set such that the collision rate can be deemed to be "low enough" and then we can proceed as if collisions do not have any significant effect on our downstream analyses. However, larger

[2] djb2 is described in http://www.cse.yorku.ca/~oz/hash.html

[3] Another tradeoff which we do not discuss in this report is the selection of the actual feature set. Picking a feature set that results in features that are too generic can be detrimental to downstream analyses. As an extreme example, consider a sliding window of one bit: the two observable features will be present in virtually all samples and thus cannot be used to distinguish the samples in downstream analyses. For this this report, we assume that a small sliding window over assembly statements is good. This assumption will be evaluated in future work.

fingerprints are also slower to manipulate. Since coclustering is a computationally hard problem (certain variants are NP-hard) and thus coclustering algorithms tend to run for a long period of time (hours or even days are not uncommon), there is a real need to artfully balance between our two needs: coclustering quality and speed.

Coping Strategies

Over the years of this program, a number of strategies has been proposed to cope with the above tradeoff. For this report, we will merely list them and only discuss Feature Normalization. Note that some of these strategies may be combined.

Reduce Collision

- *Perfect hashing* completely eliminates collision, thus offering the best obtainable coclustering result.
- Explore other hash functions. While keeping the average load (#inputs/#bins) constant, some hash functions may give a more even load than others when handling features that do occur in the real world.

Reduce Sample (Row) Space

- Use *clustering* to produce row groups, each of which is smaller than the input data set, then run coclustering on each row group as a data set. This may speed up the overall process. (Merging the individual coclustering results would be challenging, but this step may not be needed in real-world applications.)

Reduce Feature (Column) Space

- Use *principal component analysis* to identify and keep only highly-distinguishing features.
- Use *deterministic sampling* to reduce the number of observed features. This method has the benefit of offering a highly-tunable degree of control on the expected *amount* of reduction; however, the selected features do not have any semantic relationship.
- Use *feature normalization* to reduce the number of observed features. **This report focuses on this strategy and we will explain its benefits below.**

Proposal: Feature Normalization

Definition

Feature normalization is (i) the grouping of observable features into a set of user-configurable equivalent classes and (ii) the replacement of each observed feature by a canonical representation of its class. As an illustration, suppose a feature is defined to be “two consecutive assembly statements from disassembly” and the particular feature under consideration is:

```
mov edx, [esi+4*ebx]
add edx, 42
```

One possible normalization would be to regard all immediate constants as equivalent, thus yielding

```
mov edx, [esi+IMM*ebx]
add edx, IMM
```

This can be strengthened (increased normalization power) to regard all registers as equivalent as well, yielding

```
mov REG, [REG+IMM*REG]
add REG, IMM
```

In this report, we consider 6 normalization levels, with the first level being *no normalization*. We describe them in Normalizations.

Benefits of Feature Normalization

We argue that feature normalization is a desirable method to control the amount of hash collision without increasing the hash range *m*.

First, observe that feature normalization can only reduce the number of unique inputs to the hash function. This is because multiple features can be normalized into the same representative. Holding *m* constant, fewer inputs to hash means potentially fewer hash collisions.

Second, and more importantly, feature normalization offers what we call "controlled collisions". Observe that without feature normalization, two *unrelated* features can be hashed into the same value just by sheer chance. Feature normalization, on the other hand, deliberately "collides" two *related* features into their representative and then hashes the representative instead. In other words, changing the hash domain from the original feature space to the normalized feature space allows us to group features at the earlier normalization stage, thus reducing collisions at the later hashing stage. Since we get to design our normalization rules, we have complete control over what features get "collided" into a equivalence class. With our intention of designing rules that group together *semantically similar* features, we therefore argue that our controlled collisions are more desirable for downstream similarity analysis.

(Although it is still possible for two representatives to collide in the hashing stage, we notice that we can combine feature normalization with perfect hashing, thus eliminating all collisions. We leave this combination for future work.)

Proposal: Cocluster Preservation

A key challenge in our malware coclustering research is how to automatically assess the quality of a coclustering result. The key word here is "automatic": while we do occasionally receive feedback from analysts on whether a coclustering result is deemed "good", such feedback requires manual effort and thus carries very high cost. Furthermore, such feedback is categorical (not numerical) in nature and is thus not suitable for evaluating the effect of

various data reduction strategies on coclustering quality. (On the other hand, it is easy to measure their effect on running time.)

Our insight in the last quarter is that if the application of a data reduction strategy speeds up the cocluster computation and also does not change a coclustering result "by much" in a quantifiable manner, then an analyst may be quite willing to accept the strategy. Not only does our insight allow us to sidestep the obstacle of how to measure the quality of a coclustering result, it also opens the door to a new line of inquiries.

1. Using a data reduction strategy with a reduction power that is tuned numerically, does increasing the power gradually give rise to coclustering results that change smoothly? Intuitively, for a strategy that aims to control semantic meaning reduction (such as feature normalization), this is a highly desirable property on both the strategy *and* the coclustering algorithm.[4] For the strategy, this shows that it indeed *preserves* semantic meaning; for the algorithm, this shows that it *produces* semantic meaning (some form of information captured from the input data set). We leave this as future work.
2. For a data reduction strategy with a reduction power that is tuned categorically, if we can identify a chain of settings that strictly increases data reduction (as is the case in feature normalization), then we can similarly check if the chain gives rise to smooth results.
3. In general, given a chain of increasing data reductions, we can study its improvement on the running time vs. its ability to preserve a coclustering result. A big win could be have if the amount of data reduction that yields a significant running time improvement changes a coclustering result only by a small amount. If so, we may choose to accept this tradeoff.
   (Notice that we specifically do not aim to study its effect on coclustering quality — indeed, it is possible that coclustering after a data reduction gives a result that actually has a higher quality. Our focus is on *preserving* the quality that is already present in the output.)
4. We can combine our data reduction with the original approach to data reduction: reduce the hash range *m*. Observe that data reduction reduces the number of inputs reaching the hash. If we also decrease *m* carefully, then we should be able to maintain (or we can even choose to improve) the hash collision rate. Since decreasing m improves the running time, if we can show that the coclustering result is preserved after the combined changes, then this is a net win. We leave this as future work.

The Feature Friendship Metric

Having identified that our task should be comparing the change between two coclustering outputs instead of assessing the quality of an individual output, what remains is to identify a

[4] On the contrary, for a strategy that does not aim to offer any control on semantic meaning reduction (such as deterministic sampling), it is less clear whether we should expect the coclustering results to change smoothly as data reduction is increased. If it is indeed smooth, then we may conclude that there is still enough semantics left in the selected data.

suitable metric to measure such changes. For this purpose, we propose the following quantity that we call "feature friendship", or "friendship" for short.

Let x and y be two distinct features (columns) observed from a dataset and let $x\sim$[5] and $y\sim$ be the representations of x and y in the coclustering output. If $x\sim$ and $y\sim$ are in the same column group, then we say that x and y are "friends". Since there are M^* observed features, there can be at most (M^* choose 2) friendships.

We can use this notion of friendships to model the change between two coclustering results and define a notion of precision and recall. Let U be the set of all observed features and consider two coclustering results A and B. To measure the changes from A (old) to B (new), define:

A^+ = { (x, y) in $U * U$ | x and y are friends in A }

A^- = { (x, y) in $U * U$ | x and y are not friends in A }

B^+ = { (x, y) in $U * U$ | x and y are friends in B }

B^- = { (x, y) in $U * U$ | x and y are not friends in B }

Viewing Coclustering as Information Retrieval

Our trick is to define coclustering A to be a kind of "ground truth" and coclustering B to be the result in an information retrieval task. To make sense of this, we draw the following analogy:

Information Retrieval Concepts	**Search Engine**	**Coclustering**
Document Space	the set of all possible documents	the set of all pairs of *observable* unique features
Corpus	the set of documents to be indexed	the set of all pairs of *observed* unique features
Relevant Documents (ground truth)	the set of documents that *should* be returned for a query	the set of friendships present in coclustering A (denoted A^+)
Retrieved Documents	the set of documents that are *actually* returned for a query	the set of friendships present in coclustering B (denoted B^+)

Given the above view, (A^+ intersect B^+) is the true positive set and (A^- intersect B^-) is the true negative set. The false positive in B is therefore the set (A^- intersect B^+) and the false negative is the set (A^+ intersect B^-). With the TP, TN, FP, and FN sets defined, we have also defined the notion of precision and recall. Just as in any information retrieval task, if B induces a high precision and recall, then we say that "B is close to A".

The notion of sample (row) friendships is defined analogously.

[5] By $x\sim$, we really mean x with a tilde on the top.

Metric Interpretation

Although our theory is grounded in information retrieval, we feel that the terms "precision" and "recall" can be misleading and thus we want to offer an interpretation based on friendships. From the perspective of a configuration change, recall represents the fraction of friendships that is ***preserved*** from before the change to after the change. This is because
Recall = TP / (TP + FN)
= $|A^+$ intersect $B^+|$ / ($|A^+$ intersect $B^+|$ + $|A^+$ intersect $B^-|$)
= $|A^+$ intersect $B^+|$ / $|A^+|$.

Similarly, precision represents the fraction of present friendships that ***have already existed*** before the change. This is because
Precision = TP / (TP + FP)
= $|A^+$ intersect $B^+|$ / ($|A^+$ intersect $B^+|$+ $|A^-$ intersect $B^+|$)
= $|A^+$ intersect $B^+|$ / $|B^+|$.

A subtle point that we must stress is that feature friendship is a ***quadratic function*** and thus precision and recall based on friendship should be interpreted accordingly. For example, suppose a column group has width *v* before a configuration change and has width *v / 2* after. The amount of friendships contributed this column group before the change is (*v* choose *2*) = *v * (v - 1) / 2 = v^2 / 2 - v / 2*. After the change, this quantity becomes *v / 2 * (v / 2 - 1) / 2 = (v^2 / 4 - v / 2) / 2 = v^2 / 8 - v / 4*. Notice the leading term has dropped by a multiplicative factor of ***one quarter***, which is exactly what we expect when the input to a quadratic function drops by ***one half***.

Experimental Setup

Overall Directory Structure

We strive to make sure our results are reproducible and we provide a tarball that contains all our code and data. Please see the README file in the tarball once it is uploaded to TeamForge.

Data Sets

Raw

Our raw data set comes from a data drop from upstream. The file names are: {01-08-2012_1.tar, 01-09-2012_1.tar, 01-10-2012_1.tar, 09-12-2011_1.tar, 10-10-2011_1.tar, 10-11-2011_1.tar}. There are a total of 22,772 files. By running the `file` command on them and filtering the output using `` `grep -e` ``, 9,678 files match the pattern "`PE32 executable (GUI) Intel 80386, for MS Windows$`". Starting from this raw data set, we iteratively apply *reservoir sampling* to obtain the following data sets:

Large

1000 files selected from Raw.

Medium

100 files selected from Large.

Small

10 files selected from Medium.

Features

We use IDA for disassembly and for part of normalization as well.

We extract features from the assembly dumps using a *w*-gram model[6]. In this report, we use *w* to denote the window size. This value is configurable but in practice we have been using $w = 2$. This corresponds to features of two consecutive assembly instructions.

IDA outputs basic blocks, sets of assembly instructions which are executed in their entirety (atomically). Therefore windows which cross over different basic blocks are not really capturing the function of the program, so those are not included as features.

The `countFeatures` script counts the number of distinct features over a set of IDA-parsed assembly files, which is detailed below. The input of the script is a folder containing these files. The script also takes in the parameter for window size, which is the number of consecutive assembly instructions used in a feature. The script prints the number of features to standard output. For example, `python countFeatures.py norm0/ 2` counts all the features in the folder `norm0/` using a window size of 2.

The `countAllNormFeatures` script extends the previous script for multiple folders of parsed assembly files. It takes in the window size, the source folder containing the subfolders of assembly files, and a destination folder where the output will be stored. For example, suppose that the folder small/ contains the folders `norm0/`, `norm1/`, and `norm2/`, which contain parsed assembly files. Then "`countAllNormFeatures 2 small/ smallout/`" runs `countFeatures` with window size 2 on the folders `norm0/`, `norm1/`, and `norm2/`, and puts the output in `smallout/`. The output for each folder is a text file which contains the individual output of the `countFeatures` script.

Normalizations

We use IDAPython to generate the disassembly, as well as other useful information such as the mnemonic (GetMnem), operands (GetOpnd), and operand types (GetOpType) for each assembly instruction. This is stored as a 6-tuple (instr, mnem, opnd0, opnd1, optype0, optype1) for each instruction. The output of this script is a list of lists, where each inner list is a basic block, and the element of each basic block list is the aforementioned 6-tuple. The output is then serialized using Python's pickle module and written to a file. If we run the script (`raw_asm.py`) on "sample", the output file will be "`raw/sample`". This file is used as input to the normalization script.

[6] More commonly known as *n*-gram model, but since we had already used *n* we decided to use *w* here instead.

We are then using a separate script to do the normalization (`normalize.py`). There are four types of normalization: (i) instruction normalization, (ii) register normalization, (iii) memory normalization, and (iv) immediate normalization. By combining these types at different settings, we are capable of achieving 32 different levels of normalization. The script takes an integer (0-31) and uses that integer as 5 bit flags. The first 2 bits (least significant) represent the level of instruction normalization (0 = none, 1 = light, 2 = medium, 3 = heavy). The next three bits (in order of significance) are on/off switches for immediate normalization, memory normalization, and register normalization respectively. This is detailed below:

Instruction Normalization

- None: does nothing
- Light: removes additional info from instruction, leaving just the "base" instruction
  e.g., `movsw -> mov`
- Medium: does everything in light, and removes condition codes
  e.g., `jnz -> jcc`, `cmovge -> cmovcc`
- Heavy: does everything in medium, and collapses arithmetic and logical instructions into arith/log respectively

Operand Normalization

- Immediate: uses IDAPython's `GetOpType()` to convert operand to:
  5 = Immediate
  6 = Immediate Far Address (with a Segment Selector)
  7 = Immediate Near Address
- Memory: uses IDAPython's `GetOpType()` to convert operand to:
  2 = Memory Reference
  3 = Base + Index
  4 = Base + Index + Displacement
- Register: collapses registers into caller/callee save registers
  ```
  eax, ecx, edx -> caller
  ebx, esi, edi -> callee
  esp/ebp -> esp/ebp
  ```

Depending on the version of IDA, "`raw_asm.py`" is run using

```
idaw64 -A -OIDAPython:raw_asm.py sample      (Windows)
idal64 -A -OIDAPython:raw_asm.py sample      (Mac/Linux)
```

If you want to run raw_asm.py on each file in "`dir`", then the "`batchida`" bash script is provided. Simply run:

```
batchida dir
```

The normalization script is run using `python normalize.py infile outdir normlevel`. The details are included in `normalize.py`.

There is a bash script `batchnorm` that calls `normalize.py` on every file in a directory

`batchnorm indir outdir`

The script `batchnorm` uses the 6 normalization levels described below. If "sample" is normalized with normlevel 19, it will reside in `outdir/norm19/sample`; this file will be the normalized assembly. It is a Python `pickle` dump of a list of lists, where each inner list is a basic block, and each element of the inner list is the normalized assembly instruction, as a string.

In this report, we consider the following list of 6 feature normalization levels, ordered by increasing power. Each normalization contains all the optimizations from the previous normalizations. Reiterating, the normalization level corresponds to the component normalizations included, so for example normlevel = 18 = 10010_b corresponds to using medium instruction normalization and register normalization.

1. **NO:** (normlevel = 0) No normalization
2. **REG:** (normlevel = 16) Normalize registers based on caller save, callee save, and esp/ebp

   Example: `eax -> caller, ebx -> callee, esp -> esp, ebp -> ebp`
3. **CC:** (normlevel = 18) Normalize condition codes

   Example: `jge -> jcc, cmovle -> cmovcc`
4. **ARITH:** (normlevel = 19) Normalize arithmetic/logic instructions

   Example: `add -> arith, imul -> arith, xor -> log`
5. **IMM:** (normlevel = 23) Normalize immediates

   Example: `add eax, 10 -> add eax, 5` where 5 comes from IDAPython's `GetOpType()`
6. **MEM:** (normlevel = 31) Normalize memory

   Example: `lea edi, [ebx+4*esi] -> lea edi, 4 ; mov ebx [ecx] -> mov ebx 3,` where 3 and 4 come from IDAPython's `GetOpType()`

Number of Unique Features Before Hashing at Various Normalization Levels

Using the above normalizations, the number of unique features for each data set are:

Normalization \ Data Set	Small (10 files)	Medium (100 files)	Large(1000 files)
1 - NO	35280	599029	4575338
2 - REG	32845	550363	4192030
3 - CC	32732	547605	4159199
4 - ARITH	32385	543017	4124975
5 - IMM	20569	227282	1539933

6 - MEM	4565	14090	55189

Hash Quality

In our application, the hash inputs are the unique observed features, and the hash bins are the bits in the fingerprint. The maximum load of the hash table is the number of unique hash inputs that get hashed to the fullest bin. This number depends on the hash function and the data set. The average load of the hash table is the number of unique hash inputs divided by the number of hash bins. This number depends on the data set only and does not depend on the hash function.

Maximum Load at Normalization Level 1 - NO

Fingerprint Size	Small (10 files)	Medium (100 files)	Large (1000 files)
16KB / 2^17 bits	5	16	61
32KB / 2^18 bits	4	12	37
64KB / 2^19 bits	4	9	22
128KB / 2^20 bits	4	7	18
256KB / 2^21 bits	2	5	11
512KB / 2^22 bits	2	5	9
1MB / 2^23 bits	2	4	7
2MB / 2^24 bits	2	4	6
4MB / 2^25 bits	2	3	5

Maximum Load at Normalization Level 2 - REG

Fingerprint Size	Small (10 files)	Medium (100 files)	Large (1000 files)
16KB / 2^17 bits	5	16	53
32KB / 2^18 bits	4	11	32
64KB / 2^19 bits	4	9	22
128KB / 2^20 bits	3	6	17
256KB / 2^21 bits	3	6	12
512KB / 2^22 bits	2	4	9
1MB / 2^23 bits	2	4	7

2MB / 2^24 bits	2	4	6
4MB / 2^25 bits	2	3	5

Maximum Load at Normalization Level 3 - CC

Fingerprint Size	Small (10 files)	Medium (100 files)	Large (1000 files)
16KB / 2^17 bits	5	16	53
32KB / 2^18 bits	4	12	34
64KB / 2^19 bits	4	8	21
128KB / 2^20 bits	3	6	17
256KB / 2^21 bits	2	6	11
512KB / 2^22 bits	2	5	8
1MB / 2^23 bits	2	5	7
2MB / 2^24 bits	2	3	6
4MB / 2^25 bits	2	3	5

Maximum Load at Normalization Level 4 - ARITH

Fingerprint Size	Small (10 files)	Medium (100 files)	Large (1000 files)
16KB / 2^17 bits	5	15	57
32KB / 2^18 bits	4	11	33
64KB / 2^19 bits	3	10	22
128KB / 2^20 bits	3	7	14
256KB / 2^21 bits	2	6	10
512KB / 2^22 bits	2	4	8
1MB / 2^23 bits	2	4	7
2MB / 2^24 bits	2	3	5
4MB / 2^25 bits	2	3	5

Maximum Load at Normalization Level 5 - IMM

Fingerprint Size	Small (10 files)	Medium (100 files)	Large (1000 files)

16KB / 2^17 bits	4	10	25
32KB / 2^18 bits	3	8	19
64KB / 2^19 bits	3	6	12
128KB / 2^20 bits	3	5	9
256KB / 2^21 bits	2	4	7
512KB / 2^22 bits	2	4	6
1MB / 2^23 bits	2	3	5
2MB / 2^24 bits	2	3	5
4MB / 2^25 bits	2	3	4

Maximum Load at Normalization Level 6 - MEM

Fingerprint Size	Small (10 files)	Medium (100 files)	Large (1000 files)
16KB / 2^17 bits	3	4	6
32KB / 2^18 bits	2	3	5
64KB / 2^19 bits	2	3	4
128KB / 2^20 bits	2	2	3
256KB / 2^21 bits	2	2	3
512KB / 2^22 bits	2	2	2
1MB / 2^23 bits	2	2	2
2MB / 2^24 bits	1	2	2
4MB / 2^25 bits	1	2	2

Average Load - Fingerprint size: 16KB / 2^17 bits

Normalization \ Data Set	Small (10 files)	Medium (100 files)	Large (1000 files)
1 - NO	0.26917	4.57023	34.9071
2 - REG	0.25059	4.19894	31.9827
3 - CC	0.24973	4.17790	31.7322

4 - ARITH	0.24708	4.14289	31.4711
5 - IMM	0.15693	1.73402	11.7488
6 - MEM	0.03483	0.10750	0.42106

Average Load - Fingerprint size: 32KB / 2^18 bits

Normalization \ Data Set	**Small (10 files)**	**Medium (100 files)**	**Large (1000 files)**
1 - NO	0.13458	2.28511	17.45353
2 - REG	0.12529	2.09947	15.99133
3 - CC	0.12486	2.08895	15.86609
4 - ARITH	0.12354	2.07145	15.73553
5 - IMM	0.07846	0.86701	5.87438
6 - MEM	0.01741	0.05375	0.21053

Average Load - Fingerprint size: 64KB / 2^19 bits

Normalization \ Data Set	**Small (10 files)**	**Medium (100 files)**	**Large (1000 files)**
1 - NO	0.06729	1.14256	8.72677
2 - REG	0.06265	1.04973	7.99566
3 - CC	0.06243	1.04447	7.93304
4 - ARITH	0.06177	1.03572	7.86777
5 - IMM	0.03923	0.43351	2.93719
6 - MEM	0.00871	0.02688	0.10527

Average Load - Fingerprint size: 128KB / 2^20 bits

Normalization \ Data Set	**Small (10 files)**	**Medium (100 files)**	**Large (1000 files)**
1 - NO	0.03365	0.57128	4.36338
2 - REG	0.03132	0.52487	3.99783
3 - CC	0.03122	0.52224	3.96652

4 - ARITH	0.03088	0.51786	3.93388
5 - IMM	0.01962	0.21675	1.46859
6 - MEM	0.00435	0.01344	0.05263

Average Load - Fingerprint size: 512KB / 2^22 bits

Normalization \ Data Set	**Small (10 files)**	**Medium (100 files)**	**Large (1000 files)**
1 - NO	0.00841	0.14282	1.09085
2 - REG	0.00783	0.13122	0.99946
3 - CC	0.0078	0.13056	0.99163
4 - ARITH	0.00772	0.12947	0.98347
5 - IMM	0.0049	0.05419	0.36715
6 - MEM	0.00109	0.00336	0.01316

Average Load - Fingerprint size: 2MB / 2^24 bits

Normalization \ Data Set	**Small (10 files)**	**Medium (100 files)**	**Large (1000 files)**
1 - NO	0.0021	0.0357	0.27271
2 - REG	0.00196	0.0328	0.24986
3 - CC	0.00195	0.03264	0.24791
4 - ARITH	0.00193	0.03237	0.24587
5 - IMM	0.00123	0.01355	0.09179
6 - MEM	0.00027	0.00084	0.00329

Average Load - Fingerprint size: 4MB / 2^25 bits

Normalization \ Data Set	**Small (10 files)**	**Medium (100 files)**	**Large (1000 files)**
1 - NO	0.00105	0.01785	0.13636
2 - REG	0.00098	0.0164	0.12493
3 - CC	0.00098	0.01632	0.12395

4 - ARITH	0.00097	0.01618	0.12293
5 - IMM	0.00061	0.00677	0.04589
6 - MEM	0.00014	0.00042	0.00164

Research Questions

In this report we focus on the following questions. Each of them will be discussed in its own section below. To ensure that our experiments are run at a realistic fingerprint size, we select for each dataset the smallest fingerprint size such that the maximum load is strictly less than 10 ***before*** normalization. (We have not attempted to specifically optimize our choice of the maximum load.) This resulted in a fingerprint size of 16KB (2^17 bits) and 64KB (2^19 bits) for the small and medium data sets respectively[7]. The corresponding average loads are respectively 0.27 and 1.1 and the corresponding maximum loads are respectively 5 and 9. We justify this choice by observing that 32% of the bins are empty and 36% of the bins have exactly one item.[8] This matches a realistic input to a coclustering run.

How do different levels of feature normalization change BitShred's coclustering output?

The pipeline is structured as follows:

1. Prepare the fingerprints. This is done by editing `pipeline_prepare.py` to use the appropriate `DATA_DIR`, and then running `python pipeline_prepare.py norm2`, providing the normalization level you wish to produce fingerprints as an argument (in this case `norm2`).
2. Cocluster the malware. This is done by running (assuming you're working on normalization level `norm6` in this case):
   `$ /usr/bin/time -v ./bitshred -r 1 -c 4 -d norm6_db -t norm6_data -o log6.out`
   where we use `/usr/bin/time` to get statistics like time and memory usage (relative paths should be substituted as appropriate).
3. Interpret the result of running `pipeline_interpret.py`, which lists the row grouping precision and recall for each of norm2 through norm6.

The BitShred code as written provides no way to control the number of row and column groups in its final output. We considered modifying it to add this feature, but after some consideration of the coclustering algorithm it uses, we decided that this is not a good idea. Our justification is that there could be a drastic decrease in the "score" (lower is better) that BitShred assigned to a coclustering after a small variation in the number of row/column groups it produced. As such,

[7] At the time of this report, our experiments on the Large data set are still running. Therefore, we have chosen to report on the Small and Medium data sets only.
[8] The actual distribution: 0: 31.93%, 1: 36.45%, 2: 20.72%, 3: 7.97%, 4: 2.27%, 5: 0.51%, 6: 0.10%, 7: 0.02%, 8: <0.01%, 9: <0.01%

instead of controlling the exact number of rows and columns, we record here the number of groups in the output. Future work should consider how to best obtain control on the number of row and column groups.

Number of row and column groups in BitShred output

Normalization \ Data Set	Small (10 files) 16KB fingerprints	Medium (100 files) 64KB fingerprints
1 - NO	7 rows, 44 columns	33 rows, 2108 columns
2 - REG	7 rows, 39 columns	36 rows, 2242 columns
3 - CC	7 rows, 47 columns	23 rows, 2228 columns
4 - ARITH	7 rows, 35 columns	24 rows, 2117 columns
5 - IMM	7 rows, 37 columns	38 rows, 1063 columns
6 - MEM	8 rows, 26 columns	39 rows, 576 columns

Precision/Recall

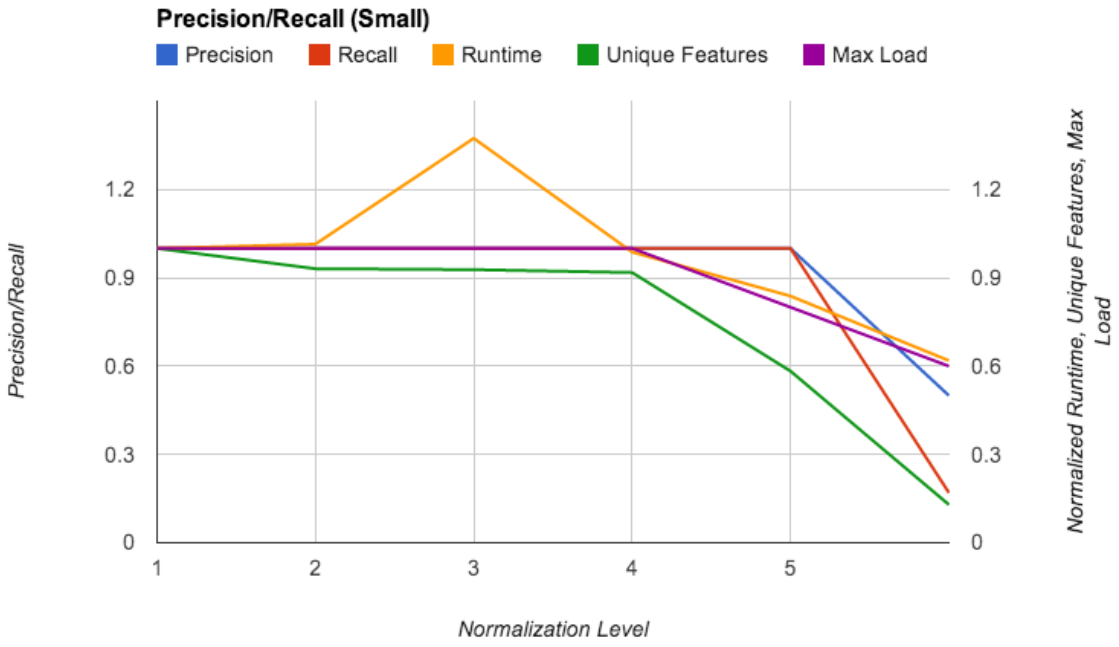
Normalization \ Data Set	Small (10 files) Fingerprint size 16KB	Medium (100 files) Fingerprint size 64KB
1 - NO	1.00/1.00	1.00/1.00
2 - REG	1.00/1.00	0.92/0.80
3 - CC	1.00/1.00	0.70/0.82
4 - ARITH	1.00/1.00	0.87/0.90
5 - IMM	1.00/1.00	0.78/0.56
6 - MEM	0.50/0.17	0.66/0.58

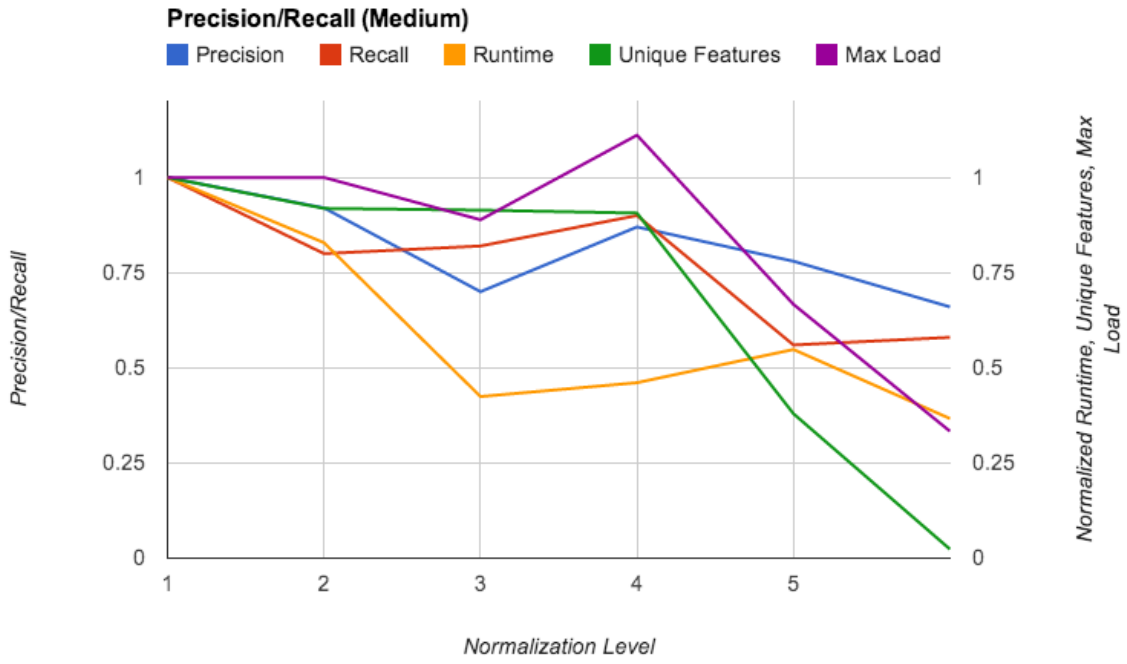
How do different levels of feature normalization affect the running time of BitShred's coclustering algorithm?

Normalization \ Data Set	Small (10 files) Fingerprint size 16KB	Medium (100 files) Fingerprint size 64KB
1 - NO	10.113s	1868m 44s
2 - REG	10.261s	1548m 28s
3 - CC	13.896s	793m 09s

4 - ARITH	9.985s	861m 29s
5 - IMM	8.475s	1024m 21s
6 - MEM	6.259s	685m 18s

Without reducing the hash range, is there a feature normalization that approximately preserves BitShred's coclustering output while being significantly faster?





Conclusions

As we can see from the plots above, feature normalization does appear to be rewarding. For example, with the Medium dataset, the running time drops by around 50% while respectable precision and recall of around 75% is maintained.

One important future work would be to control the number of row and column groups in the final output. Another would be to investigate the combined effect of reducing the range of the hash function while increasing normalization. We may also investigate the normalization levels. At present, their ordering is based on not making the number of unique features too low. We also have not investigated more creative combinations of different normalization types. (We enforced a strict inclusion order to ease result interpretation. Ideally, we can try all possible combinations.)

References

[1] J. Jang, D. Brumley, and S. Venkataraman, "BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2011, pp. 309–320.

We consulted the following links on using IDAPython:

http://www.offensivecomputing.net/papers/IDAPythonIntro.pdf
https://www.hex-rays.com/products/ida/support/idadoc/417.shtml

https://code.google.com/p/idapython/wiki/UsageInstructions