Experimental Study of Fuzzy Hashing in Malware Clustering Analysis

Yuping Li
ypli@ksu.edu
Kansas State University

Sathya Chandran Sundaramurthy
sathya@ksu.edu
Kansas State University

Alexandru G. Bardas
bardasag@ksu.edu
Kansas State University

Xinming Ou
xou@ksu.edu
Kansas State University

Doina Caragea
dcaragea@ksu.edu
Kansas State University

Xin Hu
huxin@us.ibm.com
IBM Research

Jiyong Jang
jjang@us.ibm.com
IBM Research

Abstract

Malware triaging is the process of analyzing malicious software applications' behavior to develop detection signatures. This task is challenging, especially due to the enormous number of samples received by the vendors with limited amount of analyst time. Triaging usually starts with an analyst classifying samples into known and unknown malware. Recently, there have been various attempts to automate the process of grouping similar malware using a technique called fuzzy hashing – a type of compression functions for computing the similarity between individual digital files. Unfortunately, there has been no rigorous experimentation or evaluation of fuzzy hashing algorithms for malware similarity analysis in the research literature. In this paper, we perform extensive study of existing fuzzy hashing algorithms with the goal of understanding their applicability in clustering similar malware. Our experiments indicate that current popular fuzzy hashing algorithms suffer from serious limitations that preclude them from being used in similarity analysis. We identified novel ways to construct fuzzy hashing algorithms and experiments show that our algorithms have better performance than existing algorithms.

1 Introduction

Security companies receive a huge number of malware samples every day. For instance, Cisco [10] states in their annual security report that they are processing daily around 400,000 malware samples. Various limitations (*e.g.*, time and resources) call for systematic prioritization of incoming samples. Existing triage processes, which start with binary analysis, then move into dynamic sandbox analysis, and finally end up in the hands of a human researcher, can be viewed as having an associated cost in terms of time and resources. Thus the effectiveness of binary analysis techniques is vital to minimize the cost, as the overall cost function is heavily influenced by time and resource costs of both sandboxing and human analysis.

Fuzzy hashing algorithms, also known as fuzzy hash functions, similarity preserving hash functions or similarity digest, are a type of compression functions for computing the similarity between individual digital files. In this paper, for consistency purposes, we use the term *fingerprint* as the output of a fuzzy hash function. Different from traditional cryptographic hash algorithms such as MD5 or SHA1, which are designed to be sensitive to small input modifications and can only determine if the inputs are exactly the same or not, fuzzy hash functions hold a certain tolerance for changes and can tell how different two files are by comparing the similarity of their fingerprints. This property is desirable for clustering malware because campaigns often use multiple variants from the same family that perform the exact same set of behaviors and use the same command and control, but have different cryptographic hashes.

There are numerous attempts to apply fuzzy hashing to malware analysis. For instance, ShadowServer [1] started to use fuzzy hash functions for malware similarity analysis in 2007. Around 2012, VirusTotal [2] started to generate an *ssdeep* [18] hash value for each submitted malware sample. Unfortunately, these attempts have not been rigorously validated, even though several blog posts talk about failing to apply fuzzy hashing algorithms in detecting similar executable samples [12]. So far, there has not been a rigorous evaluation and experimentation effort on the applicability of using fuzzy hash functions for malware analysis.

Intuitively, since fuzzy hashing algorithms can be used for document similarity analysis, it could be applied for malware clustering analysis, *e.g.*, by using a fuzzy hash function as a customized distance function for a hierarchical clustering algorithm. This combination could potentially allow us to find a better malware clustering approach and validate it on different datasets. In this way we can evaluate the overall effectiveness of different fuzzy hash functions, and compare them with existing static malware clustering attempts, *e.g.*, BitShred [17].

Furthermore, existing static clustering algorithms may also provide certain guidances on how to design a more effective fuzzy hash function. For instance, according to our experiments the feature hashing technique leveraged in BitShred is more effective than using Bloom filters for feature encoding (utilized in some of the fuzzy hashing functions).

The contributions of our work are:

- We identify a key design flaw – *asymmetric distance computation* – within some of existing block-based fuzzy hash functions which prevents them from being used in malware clustering. To address this issue, we design a new block-based distance computation algorithm.
- We propose a novel way to build a fuzzy hash function inspired from current fuzzy hashing algorithms and the state-of-the-art malware clustering approaches. Moreover, we show that our function performs better than existing fuzzy hash functions.
- We design and implement a generic experimentation framework for evaluating the effectiveness of different fuzzy hash functions in malware clustering.

2 Background

Initial fuzzy hash functions (*e.g.*, ssdeep) are built on top of spam detection algorithms, which are used to determine if an email is "similar" to certain known spam emails. Fuzzy hashing algorithms are also utilized during forensic investigations when malware analysts want to compare unknown files with known malware samples. For instance, ModSecurity [4] introduces a new operator [3] that uses fuzzy hashing to identify web-based malware in web attachment uploads. Using fuzzy hash functions for malware clustering has a particular appeal due to its ability to calculate a similarity score between two different samples. Primary use cases involve identifying existing malware families with updated offensive capabilities, thus requiring a review and potential update of the countermeasures.

In general, a complete fuzzy hash function consists of two parts: (1) fingerprint generation and (2) fingerprint comparison. The fingerprint generation includes a feature extraction and a feature encoding procedure. All of the existing fuzzy hashing algorithms extract the features from binary input sequences. Based on how the fuzzy hash functions encode the features and compute the final fingerprint, one can categorize existing fuzzy hash algorithms into the following two types.

Context-triggered piecewise hashing (CTPH): This type of functions split the input sequence into pieces based on the existence of special contexts, called *trigger points*, within the data object. A context (small input sequence) is considered to be a trigger point if it matches a certain property, *e.g.*, a small byte sequence whose checksum is equal to a predefined value. The function computes a hash value (*e.g.*, cryptographic hash) for the individual split-pieces and concatenates them into a final fingerprint string. ssdeep [18], FKSum [9], and SimFD [24] belong to this category.

Block-based hashing (BBH): This category of fuzzy hash functions generate one small block of the final fingerprint after a certain amount of input has been processed. The final fingerprint results from the concatenation of all the generated small block results *e.g.*, , dcfldd [14], SimHash [23], sdhash [21], mvHash-B [5], *etc*.

In this paper, we analyze and evaluate the following three fuzzy hashing algorithms: ssdeep, sdhash, mvHash-B.

ssdeep is probably the most popular application for computing CTPH for individual files and is based on a spam detector called spamsum [26]. The basic idea behind it is to split an input into chunks, hash each chunk independently and concatenate the chunk hashes to a final fingerprint. Specifically, the algorithm identifies trigger points (chunk boundaries) via a rolling hash algorithm. Two similar files will have similar trigger points and similar fingerprints.

sdhash is a BBH function. It relies on entropy estimates and a sizeable empirical study to pick out the features that are most likely unique to a data object. It then hashes the selected features with the cryptographic hash function SHA-1 and inserts them into multiple Bloom filters. The concatenation of the resulting Bloom filters constitutes the final fingerprint.

mvHash-B also belongs to the BBH category. It is an improved version of mvhash and bbHash [6]. This algorithm is based on the idea of majority voting in conjunction with run length encoding used to compress the input data and leverages Bloom filters to produce the fingerprint. According to the feature inserting procedure as described in [5], the Bloom filters of mvHash-B are fixed-size bit vectors (*e.g.*, 2048 bits).

3 Limitations of Existing Fuzzy Hash Functions

Roussev [21] identified a limitation in *ssdeep* due to its fixed size fingerprint. A fixed size fingerprint means that a smaller file and a relatively larger file cannot be meaningfully compared as they would have different compression rates. We further identified two new problems in existing fuzzy hashing algorithms, which make them inefficient or even unsuitable for similarity analysis.

3.1 Asymmetric distance computation

We identified an asymmetric distance computation problem in sdhash and mvHash-B. For these two block-based fuzzy hash algorithms, the final distance score is non-

deterministic when the order of inputs changes. Specifically, the procedure to compute the distance between two fingerprints A and B in mvHash-B is as follows: (a) if A is longer than B swaps A and B, (b) compare each block A_i in A against each block B_i in B, (c) select the minimum distance for each A_i and name it as a candidate distance, and (d) average the selected distances as a final distance score. However, when two fingerprints have the same lengths, it may happen that $distance_{(A,B)} \neq distance_{(B,A)}$, because the selected candidate distances are the local minimum values for the *first* input.

Asymmetric distance computation is critical because if we are not aware of the problem within fuzzy hashing algorithm we will end up with inconsistent results. Consequently, we will not be able to conduct consistent evaluations with regard to algorithm accuracy.

3.1.1 New Distance Computation Algorithm

We propose a new generic block based distance computation algorithm to address the asymmetric distance computation problem and use it to evaluate the effectiveness of different block based fingerprint generation algorithms. It is noteworthy to mention that the new algorithm targets only distance computation and it assumes the fuzzy hash function fingerprint is a sequence of fixed size small blocks, such as the Bloom filters fingerprint (used in sdhash and mvHash-B), or the bitvector fingerprint generated by nextGen-hash-multiblock to be introduced later. Thus for the evaluation of sdhash and mvHash-B, we replace their finger print distance computation algorithm with our new one but still using the same fingerprint generation algorithms designed in sdhash and mvHash-B. Please refer to [21] and [5] for more details.

The new distance computation algorithm is shown in Algorithm 1. Assuming a single block in a multi-block fingerprint is a fixed-size bit vector (*e.g.*, 2048 bits), the new block based distance computation algorithm is designed as follows: 1) compute all the pairwise distances between the blocks of the two input fingerprints *BFS1* and *BFS2*, store the distance results in an m by n array; 2) greedily select m global minimum distance results with no overlapping fingerprints as final candidates; 3) compute the average value of the selected candidates to represent the best match between $BFS1$ and part of $BFS2$. To further account for the difference in the lengths of the two inputs, the final distance value is a weighted average of the m smallest distance values and a term $n-m$ which accounts for the difference in the two inputs' lengths. All the pair-wise block distances and the final fingerprint distance are values between 0 and 1.

3.2 Results interpretation

Apart from similarity analysis, many fuzzy hash algorithms (*e.g.*, sdhash and ssdeep) also specifically mention that the algorithm can be used for fragment identification, in which the algorithms try to compare two fingerprints with dramatically different sizes and check how much of the smaller size fingerprint is contained in the larger fingerprint. This usage scenario would be in conflict with similarity analysis. Suppose that there are two fingerprints "abc" and "abcdef." Apparently "abc" is 100% contained in fingerprint "abcdef" (first interpretation), but their overall similarity is definitely not 100% (second interpretation). Fuzzy hash functions can only choose one of the two interpretations. Therefore, it should be clearly stated which interpretation was chosen and users should be informed about whether it can be used for overall similarity analysis.

Data: Fingerprints: $BFS1 = bf_1, bf_2, ..., bf_m$ and Fingerprint $BFS2 = bf_1, bf_2, ..., bf_n$
Result: Distance score d

```
if m > n then
    Switch BFS1 and BFS2;
end
/* Initialize a m*n distance array D and
   a size m vector S                          */
for i ← 1 to m do
    for j ← 1 to n do
        D[i][j] = hamming_distance(bf_i, bf_j);
    end
end
/* Greedily find m smallest candidates
   from distance array D, require that
   each row and each column only has one
   value selected.                            */
for k ← 1 to m do
    Let D[x][y] be the current smallest value in D;
    S[k] = D[x][y];
    Remove row x and colum y from D
end
Temporary distance d' = sum(S)/m;
/* Adjust distance score with input
   lengths information                        */
Final distance d = (d' * m + (n - m))/n
```

Algorithm 1: New block based distance computation algorithm

4 Insights on Designing a Better Fuzzy Hashing Algorithm

Code section vs whole binary: We observed that current fuzzy hashing algorithms do not take into consideration the structure of the input data. They consider the whole binary file as a data unit. Prior work on malware clustering, such as BitShred [17], extracted features from the code section of malware samples yielding good results. This led us to consider whether fuzzy hashing algorithms should extract their features only from code section of the malware samples.

Semantic vs low level features: Current fuzzy hashing algorithms extract features from low level binary input sequences. This is partially because the algorithms have no knowledge about the input file structure. However, it is very hard to derive meaningful information from low level features. High level semantic features, on the other hand, such as *disassembled instruction sequences*, are more meaningful in comparing malware [15].

Bloom filters vs feature hashing: A Bloom filter is a bit array of *m* bits, all initialized to 0. Bloom filter feature encoding approach maps each feature to multiple bits in the bit array. *Jang et al.* showed in their work [17] that the Bloom filter feature encoding approach will bring in more noise and saturate the final fingerprint more quickly, thereby generating poor results. They proposed another feature encoding approach, called "feature hashing," which maps each feature into exactly one bit in the bit array.

Single vs multiple block fingerprint: Currently most of block-based fuzzy hashing algorithms are using multiple small blocks to represent the final fingerprint instead of one large block. It is referred to as "Bloom filters" in the algorithms. Each "Bloom filter" handles an input sequence and the results are concatenated together into the final fingerprint. We also observe that BitShred maps all the features into one large block - a 16KB bit array. Therefore, we also want to know which fingerprint representation would be more effective for comparing similarity. However, since the fingerprint formats are different, we have to also use different distance computation algorithms for this comparison.

5 New Fuzzy Hashing Algorithm Design

In order to verify our insights, we designed a new fuzzy hashing algorithm and implemented it in python. The algorithm extracts *5-grams* from the code section of a malware sample, uses *feature hashing* to encode them into a bit array as final fingerprint. We use 16KB as the size of the fingerprint for a given malware sample. Similarity between the fingerprints is computed using *bitwise Jaccard* distance function. We call this algorithm *nextGen-hash*. We then implemented the following variations of our algorithm to effectively test our insights.

nextGen-hash-imp is a fuzzy hashing algorithm similar to nextGen-hash. The main difference is that nextGen-hash-imp analyzes the executable structures, extracts "import table entries" from header part of each executable as features, and then applies feature hashing to generate a bitvector as fingerprint. It also uses bitwise Jaccard function to compute the similarity between two fingerprints. In particular, we set the fingerprint to be 16KB.

nextGen-hash-bf is a fuzzy hashing algorithm similar to nextGen-hash. The main difference between nextGen-hash-bf and nextGen-hash is how the features are encoded into the fingerprint. It also analyzes the executable structures, and uses 5-gram binary sequences from the code section as features but applies a Bloom filter approach (one feature is mapped to multiple bits in the fingerprint) to map features into a large bitvector. Specifically, we compute a SHA1 hash for each 5-gram feature. Next we split the hash output into 5 equal length smaller hash values to simulate 5 different hash functions so we can set the corresponding bits in the fingerprint. The algorithm uses bitwise Jaccard distance function to compute the similarity of two fingerprints, and the fingerprint size is set to 16KB.

nextGen-hash-multiblock is a fuzzy hashing algorithm which uses a different fingerprint format and distance computation function. The program also analyzes the executable structures, extracts 5-gram features from code section sequence, and then applies the feature hashing technique to map every 1500 features into a 2048-bit bitvector. The final fingerprint is the concatenation of all of the small bitvectors. Therefore the length of the final fingerprint is dynamic (multiple of 2048 bits), and the fingerprint consists of variable number of small blocks. The value of 2048 is chosen because both sdHash and mvHash-B set their single block to be 2048 bits. We use the new block based distance computation algorithm introduced in section 3.1.1 to compute the similarity of two fingerprints.

6 Evaluation Framework

In order to verify the previously identified fuzzy hashing limitations and evaluate their effectiveness for malware similarity analysis, we designed and implemented a modularized framework as shown in Figure 1. We combine fuzzy hashing and clustering analysis together, and use fuzzy hash function as customized distance function for agglomerative hierarchical clustering (with *average* linkage). In the beginning, each fingerprint is in a cluster of its own. The clusters with shortest distance are then sequentially combined into larger clusters until the distance between two clusters are greater than user specified threshold. For *average* linkage clustering, the distance between any two clusters is defined as the mean distance between all fingerprints of each cluster. The clustering algorithm we used is *fastcluster* [11].

As shown in Figure 1, fingerprint generation and distance computation are abstracted as public interfaces for clustering analysis, so that we can support arbitrary fuzzy hash functions. Optionally, certain fuzzy hash functions will need a pre-processing component, which may involve unpacking malware samples, decompiling and extracting code section sequences. The main advantage of the framework is that it can be used as a platform for ver-

ifying the applicability of fuzzy hashing algorithms in clustering analysis. We validated our insights and findings regarding the effectiveness of fuzzy hashing functions in malware analysis using the framework.

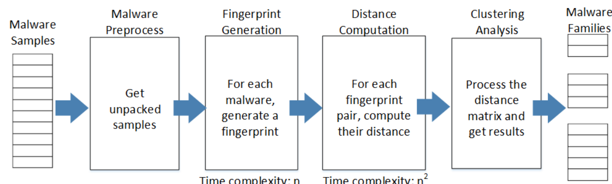


Figure 1: Fuzzy Hash Evaluation Overview

7 Data preparation

The accuracy of cluster analysis may depend on the data set used. To create reliable ground truth, it is desirable to prepare accurately labeled malware samples. In this research, we start with one archived data set from VirusShare [20].

7.1 Filter packed malware samples

Executable packing is to compress an executable file and combine the compressed data with decompression code into a single executable, which is commonly seen in most modern malware. Since there are hundreds of different packers available, it is almost impossible to precisely unpack all of the samples. Compressed executables may present a completely different binary sequence and bring in noise for the analysis. In our evaluation, we filter out the packed samples from the data set and conduct cluster analysis only on unpacked samples.

1. **Malware packers classification model:** By applying machine learning techniques, Perdisci *et al.* [19] proposed a malware classification model for automatically distinguishing between packed and unpacked samples with high accuracy. We adopt their feature extraction tool together with their training data set (5498 training points), and conduct the classification analysis on Weka [13] platform to get the unpacked samples.

2. **Static signature based matching:** We also use PEiD [25] with 28307 different signatures to detect common packers, cryptors and compilers for PE files. PEiD is one of the most widely used packer detectors among malware analysts. The drawbacks are that it is relatively easy to evade the static signatures used by the software, it is almost impossible to cover all of the new packers, and one PE sample can be matched with multiple signatures. PEiD is used as an auxiliary step to filter the packed samples.

3. **Other heuristic packing filters:** Since there is no guarantee that after the previous two steps all packed samples have been removed from the input, we also apply several heuristic filters. First we analyze the section names of the PE executables and remove those with non-standard section names. Second, we submit the file hash to VirusTotal and extract the corresponding labels returned by different anti-virus companies, and then remove those samples with "packers," "packed," "obfuscators," "themida," which can be viewed as strong indicators of packed samples.

7.2 Reliable malware family information

We make use of VirusTotal to obtain more reliable family label information. After extracting the keywords from the VirusTotal returned scan labels and removing the less informative ones (such as malicious, agent, malware, Trojan *etc.*), we collect the count information of all unique keywords. We then choose the majority keyword as the family name with the requirement that at least 40% of all anti-virus companies have flagged the sample as malicious, and at least 75% of those with positive flags contain the same label keyword in their scanning results.

The above unpacking and family name assigning processes are combined to create a highly reliable data set. To prevent potentially biased or skewed distribution of clusters, we limit the size of each malware family to be less than or equal to 300, and discard those families that only have less than 10 samples. Eventually, we end up with 1146 samples from 17 different malware families. The family statistics of the selected malware samples are shown in Table 1.

Family	Count	Family	Count
Viking	32	Vilsel	185
Fesber	57	Jeefo	36
Neshta	39	Turkojan	22
Skintrim	41	Bettersurf	300
Ramnit	38	Koutodoor	30
Zenosearch	99	Zbot	22
Hupigon	28	Fosniw	22
Domaiq	147	Wabot	27
Xpaj	22		

Table 1: Malware Family Statistics

8 Evaluation

Instead of comparing the performance of the algorithms with respect to computational and space complexity [8], we evaluate the "correctness" of algorithm results by leveraging clustering analysis. We believe evaluating correctness is more helpful as the computational performance depends highly on algorithm implementation specifics and the programming language used.

We use precision and recall to measure the accuracy of clustering results. Assume we have n malware samples from m different families; the ground truth reference set

can be represented as $F = F_1, F_2, ..., F_m$. For a specific threshold the algorithm separates the samples into t clusters. The results can be represented as $C = C_1, C_2, ..., C_t$. Formally, the precision P and recall R are defined as:

$$P = \frac{1}{n}\sum_{i=1}^{t} max(|C_i \cap F_1|, |C_i \cap F_2|, ..., |C_i \cap F_m|)$$

$$R = \frac{1}{n}\sum_{j=1}^{m} max(|F_j \cap C_1|, |F_j \cap C_2|, ..., |F_j \cap C_t|)$$

Precision is to measure the purity of clustering analysis results and check how well the clustering algorithm assigns samples of different families to different clusters. Recall is to check how well the malware samples from same family are grouped into the same cluster by the algorithm. Precision P would be 1 when each cluster in C is completely pure, that is, the members of each cluster belong to the same family; likewise, recall would be 1 if every family group of the reference set is a subset of one cluster in C, an extreme case is when C only contain one cluster. Usually, optimizing only one measurement will inevitably sacrifice the other one. Therefore, we take the intersection of the two metrics as a balance point between precision and recall, and use it as the measurement of clustering performance.

8.1 Effects of considering underlying data structure

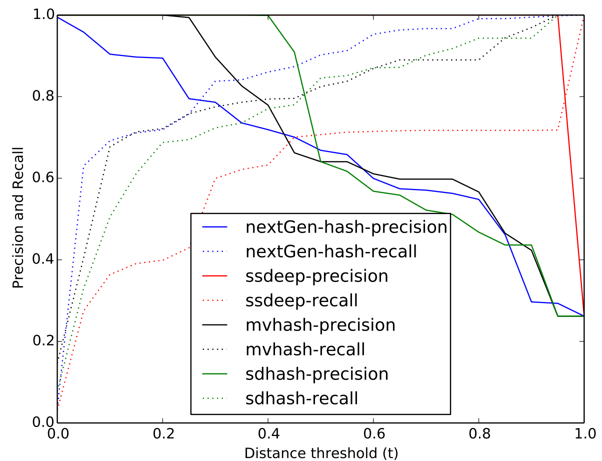
Algorithms: ssdeep, sdhash, mvHash-B

In this experiment, we use whole binary sample as one input set, then analyze each sample and extract the code section from the same samples as another input set, then apply three existing fuzzy hash functions and our newly designed function to generate fingerprints and compare their clustering results. As described previously in Section 3, sdhash and mvHash-B have the asymmetric distance computation problem, we use the new block based distance computation algorithm to evaluate the effectiveness of sdhash and mvHash-B fingerprint generation algorithm.

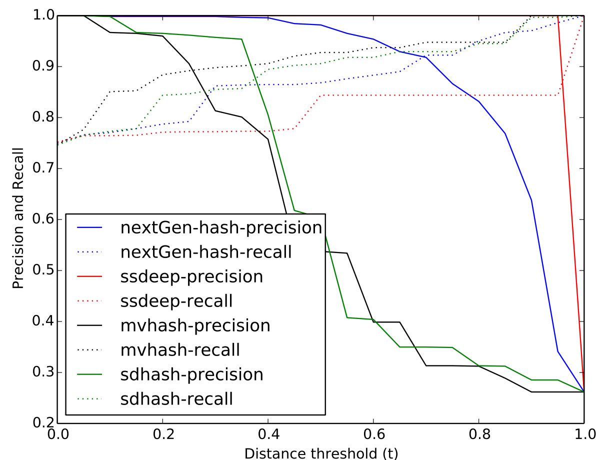
Figure 2 shows that for all fuzzy hash algorithms, code section as input can always generate more accurate clustering results, Table 2 is the detailed clustering results, each entry in the table is the intersection point between precision and recall. This verifies our first insight in section 4. This might be because: except for code section content, whole binary also includes almost similar PE header and other structural information which result in overestimated similarity. Table 2 also shows that our newly designed algorithm can generate better results if using code section sequence as input.

Algorithm	Whole sample	Code section
ssdeep	0.797	0.872
sdHash	0.807	0.877
mvHash-B	0.792	0.893
nextGen-hash	0.791	0.919

Table 2: Fuzzy Hash Clustering Performance



(a) Functions taking whole binary as input



(b) Functions taking code section as input

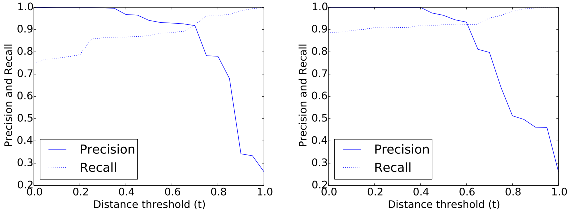
Figure 2: Effects of considering underlying data structure

8.2 Effects of higher level semantic features

Algorithms: nextGen-hash, nextGen-hash-imp

In this experiment, we evaluate the performance differences brought by different level of features of the same malware data set. Specifically, nextGen-hash firstly analyzes the executable structure and extracts n-gram sequences (from code section of samples) as features; while nextGen-hash-imp firstly analyzes the executable structure and extracts import table entries as features. By comparing their clustering performances on the same malware dataset, we could check the effects brought by higher level semantic features.

As presented in Figure 3, nextGen-hash-imp actually generates slightly better results than nextGen-hash, their intersection points between precision and recall were 0.919, 0.924, respectively. This verifies our second insight in Section 4. Usually, the number of “import table



(a) Clustering results with nextGen-hash (b) Clustering results with nextGen-hash-imp

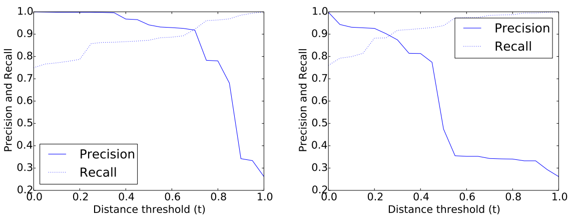
Figure 3: Effects of higher level semantic features

entries” is significantly smaller than the total number of n-gram sequences for a single malware sample. This experiment shows “import table entries” features are more representative and efficient compared with low level n-gram features when comparing overall similarity.

8.3 Effects of different feature encoding strategy

Algorithms: nextGen-hash, nextGen-hash-bf

We evaluate the performance differences brought by different feature encoding strategy in this experiment. The only difference between nextGen-hash and nextGen-hash-bf is feature encoding strategy: for nextGen-hash, each feature is mapped to a single bit in a 16KB fingerprint (feature hashing approach); while for nextGen-hash-bf, each feature is mapped to 5 bits in a 16KB fingerprint (Bloom filter approach). By comparing their clustering performances on the same malware dataset, we could check the effects introduced by different feature encoding strategy. As shown in Figure 4, nextGen-hash provided better results than nextGen-hash-bf, and their precision and recall intersection were 0.919, 0.894. It verifies the third insights described in Section 4. This observation has already been discussed and analyzed in [17].



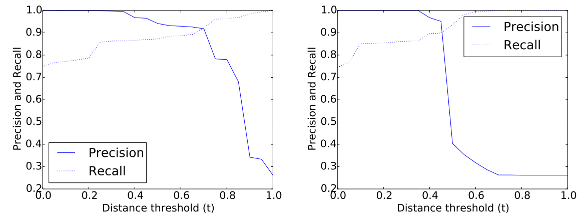
(a) Clustering results with nextGen-hash (b) Clustering results with nextGen-hash-bf

Figure 4: Effects of different feature encoding strategy

8.4 Effects of different fingerprint format

Algorithms: nextGen-hash, nextGen-hash-multiblock

To check the effects of different fingerprint format, we compared the clustering performances of nextGen-hash and nextGen-hash-multiblock. They are both using the same 5-gram features and the same feature encoding strategy, but with different fingerprint formats. The fingerprint of nextGen-hash is one big block (16KB bit vector), while the fingerprint of nextGen-hash-multiblock is the concatenation of multiple small size blocks (2048 bit each). We use the new block based distance computation algorithm to compare the similarity of nextGen-hash-multiblock fingerprints. As illustrated in Figure 5, the intersection of precision and recall for nextGen-hash (0.919) was higher than nextGen-hash-multiblock (0.907). One potential explanation for the results is that the fingerprint of nextGen-hash-multiblock is usually more compact and smaller in size compared with the fingerprint generated by nextGen-hash, thus for nextGen-hash-multiblock, features will have more chances to encounter collisions since they are mapped to smaller space.



(a) Clustering results with nextGen-hash (b) Clustering results with nextGen-hash-multiblock

Figure 5: Effects of different fingerprint format

9 Related work

Evaluation of fuzzy hash functions: Researchers have tried throughout the years to evaluate existing fuzzy hash tools. Roussev *et al.* [22] provided a baseline evaluation of the capabilities of fuzzy hash tools in a controlled environment and on real-world data. Breitinger *et al.* [8] proposed a framework to test the efficiency, sensitivity, and robustness of similarity hashing. Furthermore, Breitinger *et al.* [7] introduced an automated evaluation of fuzzy hash tools by using longest common substrings as ground truth results.

Static-based malware similarity analysis: Jacob *et al.* [16] proposed a static, packer-agnostic filter to detect similar malware samples. Researchers also explored static features for malware clustering. Based on static code reuse analysis, Jang *et al.* [17] built the BitShred system to extract direct n-gram features from the code section sequence of the malware samples, then applied the feature hashing & winnowing techniques to dramatically reduce feature spaces into fixed length bit-vector, and applied bit-wise Jaccard distance to compute pairwise distance. Hu *et al.* [15] designed and implemented a system, called MutantX-S, that can extract n-gram features from sequences of opcodes to represent the original binary program. It is noteworthy to mention that MutantX-S included a generic malware unpacking algorithm to handle usually obfuscated samples which made their system more applicable to practical usage. Applying fuzzy hashing algorithms for malware clustering is

more close to static-based clustering analysis, and malware packing is also considered as an orthogonal problem.

Our research is different from previous work in that: we are focused on the effectiveness of fuzzy hashing algorithms in malware clustering analysis. To the best of our knowledge, our work is the first work that systematically evaluates the effectiveness and limitations of existing fuzzy hashing algorithms in malware clustering.

10 Conclusion

In this research, we conduct a comprehensive evaluation of existing fuzzy hashing algorithms with the goal of studying their applicability in malware similarity analysis. Apart from problems that have been raised before, we find a asymmetric distance computation design flaw within several existing block-based fuzzy hashing algorithms, which make them unsuitable for practical use. We designed a new distance computation algorithm to address the asymmmetricity problem in block based fuzzy hashing algorithms. We also developed key insights based on literature study of existing malware similarity analysis algorithms and use them to develop a new fuzzy hashing algorithm. We designed and developed a modularized framework for evaluating different fuzzy hashing algorithms. Evaluation of our algorithm was performed on a public malware dataset using the framework. The results indicate that fuzzy hashing algorithms can indeed be used in malware similarity analysis provided they are designed based on our insights.

11 Acknowledgement

We would like to thank Marc Eisenbarth and Arbor Networks for providing the initial malware samples that started this research. This material is based upon work supported by U.S. National Science Foundation under award no. 0954138 and 1314925, and by the Air Force Office of Scientific Research award FA9550-12-1-0106. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the above agencies.

References

[1] ShadowServer - Fuzzy Clarity: Using Fuzzy Hashing Techniques to Identify Malicious Code. http://tinyurl.com/ol9ld8u, 2007.

[2] Virustotal. https://www.virustotal.com, 2014.

[3] ModSecurity Advanced Topic of the Week: Detecting Malware with Fuzzy Hashing. https://tinyurl.com/kd4twln, 2015.

[4] ModSecurity: Open Source Web Application Firewall. https://www.modsecurity.org/, 2015.

[5] F. Breitinger, K.P. Astebol, H. Baier, and C. Busch. mvHash-B - A New Approach for Similarity Preserving Hashing. In *IEEE IMF*, 2013.

[6] F. Breitinger and H. Baier. A fuzzy hashing approach based on random sequences and hamming distance. In *Proceedings of the Conf. on Digital Forensics, Security and Law*, 2012.

[7] F. Breitinger and V. Roussev. Automated Evaluation of Approximate Matching Algorithms on Real Data. In *Digital Investigation Journal*, 2014.

[8] F. Breitinger, G. Stivaktakis, and H. Baier. FRASH: A Framework to Test Algorithms of Similarity Hashing. In *In Digital Investigation Journal*, 2013.

[9] L. Chen and G. Wang. An Efficient Piecewise Hashing Method for Computer Forensics. In *IEEE WKDD*, 2008.

[10] Cisco. Annual Security Report. http://tinyurl.com/kzzs95a, 2014.

[11] M. Daniel. fastcluster: Fast hierarchical, agglomerative clustering routines for r and python. *J. Stat. Softw*, 2013.

[12] D. French. Fuzzy Hashing Against Different Types of Malware. http://tinyurl.com/6q97d6y, 2011.

[13] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I.H. Witten. The WEKA Data Mining Software: An Update. In *ACM SIGKDD Explorations Newsletter*, 2009.

[14] N. Harbour. Dcfldd - Defense Computer Forensics Lab. http://dcfldd.sourceforge.net/, 2002.

[15] X. Hu, K.G. Shin, S. Bhatkar, and K. Griffin. MutantX-S: Scalable Malware Clustering Based on Static Features. In *USENIX ATC*, 2013.

[16] G. Jacob, P.M. Comparetti, M. Neugschwandtner, C. Kruegel, and G. Vigna. A Static, Packer-agnostic Filter to Detect Similar Malware Samples. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2013.

[17] J. Jang, D. Brumley, and S. Venkataraman. Bitshred: Feature Hashing Malware for Scalable Triage and Semantic Analysis. In *ACM CCS*, 2011.

[18] J. Kornblum. Identifying Almost Identical Files Using Context Triggered Piecewise Hashing. In *Digital Investigation Journal*, 2006.

[19] R. Perdisci, A. Lanzi, and W. Lee. Classification of Packed Executables for Accurate Computer Virus Detection. In *Pattern Recognition Letters*. Elsevier, 2008.

[20] J.M. Roberts. VirusShare.com. http://virusshare.com/, 2014.

[21] V. Roussev. Data Fingerprinting with Similarity Digests. In *Advances in Digital Forensics VI*. Springer, 2010.

[22] V. Roussev. An Evaluation of Forensic Similarity Hashes. Elsevier, 2011.

[23] C. Sadowski and G. Levin. Simhash: Hash-based Similarity Detection. Technical report, Technical report, Google, 2007.

[24] K. Seo, KS. Lim, J. Choi, K. Chang, and S. Lee. Detecting Similar Files Based on Hash and Statistical Analysis for Digital Forensic Investigation. In *IEEE CSA*, 2009.

[25] Snaker, Qwerton, and Jibz. PE iDentifier (PEiD). https://tuts4you.com/download.php?view.398, 2008.

[26] A. Tridgell. Spamsum. http://tinyurl.com/qf89mgu, 2002.

[27] T. Wang, X. Hu, S. Meng, and R. Sailer. Reconciling Malware Labeling Discrepancy Via Consensus Learning. In *IEEE ICDEW*, 2014.