BitShred: Fast, Scalable Code Reuse Detection in Binary Code

Jiyong Jang, David Brumley

November 16, 2009

CMU-CyLab-10-006

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

BitShred: Fast, Scalable Code Reuse Detection in Binary Code

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

Abstract

Many experts believe that new malware is created at a rate faster than legitimate software. For example, in 2007 over one million new malware samples were collected by a major security solution vendor. However, it is often speculated, though to the best of our knowledge unproven, that new malware is produced by modifying existing malware, either through simple tweaks, code composition, or a variety of other techniques. Moreover, when buggy code is copied from one program to another program, both original and new programs have to be patched. However, code copying is typically not recorded. Such code reuse is a recurring problem in security.

In this paper we propose a fast, scalable algorithm for automatic code reuse detection in binary code, *BitShred*. BitShred can be used for identifying the amount of shared code based upon the ability to calculate the similarity among binary code. BitShred can be applied to many security problems, such as malware clustering and bug finding. We developed a prototype implementation to evaluate our algorithm. The experimental results show that BitShred is able to detect plagiarism among malware samples and cluster them efficiently.

1 Introduction

Automatic code reuse detection, finding common code sequences between two sets of programs, is a recurring problem in security. Automatic code reuse detection is a core component in many security scenarios, including:

- **Malware Clustering.** More malware (malicious software) was created in 2007 than the previous 20 years combined [6]. At least one large security vendor suggests that the rate at which new malware is created may exceed the rate at which legitimate software is created [5].
  It does not seem likely that such new malware programs are written from the scratch each day. Instead, it is much more likely that malware authors are modifying existing code slightly to produce new malware samples. Unfortunately, there is little information available that provides algorithms and metrics for scientifically demonstrating such copying.
  One straight-forward application of code reuse detection is to identify copied or related code. Once copied code (and unique code) is identified, there are many subsequent uses, such as using such code as feature vectors in machine learning algorithms that recognize new malware. Similarly, code reuse detection can be used to identify copied or related code and to cluster malware: if two malware samples contain shared code, they are likely to be of the same overall family.
- **Bug Finding.** Recent research shows that there is extensive code reuse in open source software [9]. When buggy code from one project is copied to a new project, both the original and new project need to eventually patched. It is possible that a bug already fixed in one project still exists in other projects when those projects share their code.
  Code reuse detection can be used to identify code similar to a known buggy version. For example, CP-Minder [8], a source-code copy-paste related bug detection tool, found 49 bugs in Linux that were due to developers not fixing copied buggy code.

53 8a 5c 24 08 56 24 08 56 24 08 8a 5c 24 08 56

(a) Given byte sequence

538a5c2408	8a5c240856	5c24085624	2408562408
0856240856	5624085624	2408562408	085624088a
5624088a5c	24088a5c24	088a5c2408	8a5c240856

(b) Derived shreds with size 5

Figure 1: Shredding a byte sequence when $n = 5$

In order to enable these scenarios, we need an algorithm that can automatically identify code reuse in a scalable manner, and even when source code is not available. In many cases we do not have access to the source code for programs we wish to perform automatic code reuse detection on. Malware authors are unlikely to provide source code in order to aid malware classification and categorization. Most people do not have source code to the programs they run, thus cannot check for bugs on source-code level analysis. Almost everyone, however, has access to binary (i.e., executable) code. Automatic code reuse detection in many of the above scenarios has the most value when used on very large datasets. For example, security companies may have over a million malware samples they would like to analyze to determine how much malware is really unique.

In this paper we propose a light-weight and scalable algorithm for code reuse detection in binary code. Our algorithm can be used to identify the amount of shared code based upon the ability to calculate the similarity among binary code. We demonstrate the effectiveness of our algorithm by applying it to clustering malware samples collected by CWSandbox.

Contributions. We make the following contributions:

- We develop an efficient algorithm for automatic code reuse detection in binary code. At the heart of our approach is to use Bloom filters as fingerprints. The fixed small size of fingerprints enables fast and scalable code reuse detection.
- We show how our algorithm can be applied to code reuse detection problems like malware clustering.
- We develop a prototype implementation to evaluate our algorithm. We apply our tool to real malware samples. Our experiments show that our tool is able to quickly cluster malware samples based on the similarity between the samples.

This paper is organized as follows: Section 2 depicts the design of our system. Section 3 provides experimental results, and Section 4 presents future work. Section 5 describes related work and concludes the paper.

2 BitShred

In this section, we propose *BitShred*, our algorithm for code reuse detection. BitShred consists of 3 phases: shredding a file, creating a fingerprint, and comparing fingerprints.

2.1 Shredding

For each given binary, BitShred first parses the binary and identifies all executable code sections. BitShred then divides them into fragments called *shreds*. A shred is a contiguous byte sequence of length n, often called n-gram, where n is chosen by a user. Figure 1 shows an example when $n = 5$.

If the size of a shred is too small, then it is hard to catch meaningful byte sequences. On the other hand, if the size of shred is too big, then it is not resilient to code reordering.

2.2 Creating a Fingerprint

A naive approach to detecting code reuse between two programs is to pairwise compare all shreds. However, pairwise matching is not storage-efficient or scalable in that it requires much larger space than the original

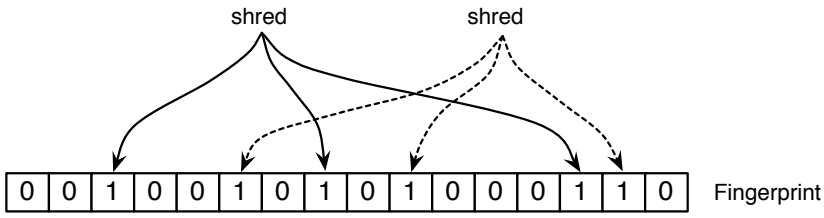

Figure 2: Fingerprinting when $k = 3$

programs to store all shreds (Section 3.1).

To improve scalability, we use Bloom filters [4] in BitShred. A Bloom filter is a data structure used for set membership tests. Suppose there is a data set S. A Bloom filter can tell whether an element x is a member of S in a storage-efficient way. A Bloom filter may have one-sided errors. A false positive occurs when the Bloom filter membership test returns $x \in S$ when x is not really in S. Bloom filters have no false negatives, i.e., membership tests never return $x \notin S$ when x is really a member of S.

A Bloom filter consists of a bit array with m bits and k different hash functions. Initially, all bits of the bit array are set to 0. To add an element, k hash functions are applied to the element, and the bits indexed by the resulting hash values are set to 1.

To test whether an element x is in the set S, x is hashed with k hash functions. If the bits indexed by the resulting k hash values are all set, x is deemed in the set S. Otherwise, x is deemed not a member of the set S.

Figure 2 shows how to create a fingerprint. All shreds of a given file are hashed using k hash functions, and the corresponding bits of a Bloom filter is set to 1. After adding all shreds, the Bloom filter is considered the fingerprint of the file.

2.3 Comparing Fingerprints

Resemblance. We use the Jaccard index to measure similarity between two files A and B. The Jaccard index is defined as the intersection size of two samples divided by the union size of two samples:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}.$$

We use the Bloom filter fingerprints as A and B to calculate the Jaccard index. In other words, if a specific bit of A is set, then the corresponding attribute of A is 1; otherwise, it is 0. Thus, the Jaccard index can be calculated as follows:

$$J(A, B) = \frac{F_{11}}{F_{01} + F_{10} + F_{11}},$$

where F_{11} means the total number of bits that both A and B set, F_{01} means the total number of bits that only B set, F_{10} means the total number of bits that only A set.

We can easily calculate F_{11} by counting the number of set bits of bitwise AND between the fingerprint of A and the fingerprint of B. $F_{01} + F_{10} + F_{11}$ can be derived by counting the number of set bits of bitwise OR between the fingerprint of A and the fingerprint of B. Therefore, the similarity between file A and B is calculated as

$$J_R(A, B) = \frac{S(BF_A \wedge BF_B)}{S(BF_A \vee BF_B)}, \tag{1}$$

where $S(BF)$ means the number of set bits of the Bloom filter BF.

Containment. In some cases, file A can include file B. For example, file B is a library object file and file A statically links in B. The similarity between file A and file B would be very low since the size of file A is much bigger than the size of file B.

	naive	winnow_w4	winnow_w8	winnow_w12	bitshred_64K	bitshred_32K
Size of fingerprints	16.78GB	3.14GB	1.86GB	1.35GB	1.71GB	0.85GB
Average Error *(0∼1)*	-	0.003	0.006	0.007	0.052	0.105
Average Error *(over 0.5)*	-	0.002	0.004	0.005	0.003	0.007
Time to compare	51m 13s	5m 22s	2m 59s	2m 5s	24s	15s

Table 1: Analysis of Fingerprinting ($n = 16bytes$ and 64-bits hash function was used for winnowing.)

If we need to consider the *containment* case, we can calculate the similarity using J_C instead of J_R.

$$J_C(A, B) = \frac{S(BF_A \wedge BF_B)}{S(BF_B)}, \tag{2}$$

when $S(BF_A) > S(BF_B)$.

Clustering. We can measure similarity between two files using J_R and J_C. For clustering, with defined threshold value t and the calculated similarity between files, we can group two files that have the higher similarity than t into the same cluster.

3 Evaluation

3.1 Analysis of Fingerprinting

We compared our fingerprinting algorithm to winnowing by creating fingerprints for 28,001 files and calculating the similarity between a randomly picked test file and the other 28,000 files. We performed our experiments on a Linux 2.6.28-11 machine (Intel Core2 6600 / 4GB memory).

Size of Fingerprints. The total size of the 28,001 files and their executable code sections is 7.90GB and 1.05GB, respectively.

Table 1 shows the total size of fingerprints. Note that the size of the fingerprints in a naive pairwise shred matching was 16.78GB, which is even larger than the total size of the target files.

In winnowing, the number of selected fingerprints depends on the window size w [10]. We ran experiments with w=4, 8, and 12. As w increased, winnowing selected less number of fingerprints from a hash sequence.

The size of fingerprints in BitShred obviously relies on the size of the Bloom filter. We used different sizes of Bloom filters, 64KB and 32KB, and 2 hash functions for our experiments.

The result shows that BitShred can be more scalable than the naive pairwise matching and winnnowing in that BitShred requires less space for storing fingerprints and less time to compute similarity. While BitShred generates a fixed size of fingerprint for each file, the size of fingerprints in winnowing is variable depending on the size of files. This fixed size of fingerprints allows us to manage the fingerprint database more efficiently.

Similarity Measure. We used the similarity derived by the naive pairwise matching as the reference, which compares every shred one by one, and then calculated the average error between the reference and winnowing as follows:

$$\mathsf{error}_{win} = \frac{1}{m}\sum_{i=1}^{m}(|\mathsf{similarity}^i_{naive} - \mathsf{similarity}^i_{win}|),$$

where m is the total number of elements. Similarily, we calculated the error between the reference and Bitshred.

Table 1 provides the calculated average error. The error of BitShred increased as the size of a Bloom filter shrunk. The error of BitShred came from the false positive in Bloom filters. For example, quite different two samples have extremely low similarity by the naive pairwise matching; however, false positives in Bloom filters may cause to share some bits between two fingerprints.

threshold (t)	# of clusters	threshold (t)	# of clusters
0.60	1,564	0.80	2,061
0.65	1,798	0.85	2,113
0.70	1,921	0.90	2,190
0.75	1,999	0.95	2,242
		1.00	2,895

Table 2: BitShred clustering results with various threshold

In addition, for clustering similar files, we are interested in only files with high similarity. Therefore, we calculated the average error with files over 0.5 similarity to see if BitShred can calculate similarity for similar files correctly. As shown in Table 1, both winnowing and Bitshred recorded a high accuracy with an average error of 0.007.

Comparing Performance. To calculate similarity between two files, winnowing needs to compare all hashes from both files one by one since the fingerprints in winnowing are hashes sequences. Suppose that the number of hashes in fingerprints of two different files are p and q, respectively. The complexity for comparing two fingerprints is $O(pq)$. BitShred, on the other hand, only needs to perform two operations: bitwise OR and bitwise AND. This makes the comparison between two files simpler and faster. The time to perform bitwise operations is proportional to the number of bits of a Bloom filter, m. Therefore, the complexity is calculated as $O(m)$.

Since we could not use the original code for winnowing, we excluded the time for creating fingerprints and only recorded the time for comparing between fingerprints to assess performance.

Table 1 shows the required time to compare the fingerprint of the tested file with the fingerprints of other 28,000 files. The results demonstrate that BitShred can calculate similarity among files 7.4 times faster at `bitshred_64K` than winnowing at `winnow_w8` on similar size of fingerprints database.

3.2 Malware Clustering

We evaluated BitShred on 9,404 malware samples collected by CWSandbox [1]. For our experiments, we used *unpacked* malware samples. We verified that malware samples are unpacked using PEiD [7].

Experiments The total file size of the malware samples is 2.20GB, and the total executable code section size of the malware samples is 0.39GB. It took 2min 9sec to fingerprint the 9,404 malware samples. After building the fingerprint database of malware samples, BitShred clustered malware samples with $0.6 \sim 1.0$ threshold values. It took average 61 minutes to cluster 9,404 malware samples [1]. Table 2 shows the number of resulting clusters according to the threshold values. As the threshold increases, the samples are divided into more clusters.

Clustering Quality Since there is no existing reference malware clustering data, we used ClamAV scanning result as a reference to evaluate clustering accuracy. To quantify the difference between the reference clustering $(R_1, R_2, ..., R_r)$ and BitShred clustering $(C_1, C_2, ..., C_c)$, we calculated Precision and Recall [3].

Precision measures the exactness of clustering, i.e., how well the clustering technique can separate different elements into different groups. Precision can be formalized as follows:

$$\mathsf{Precision}(C_i) = \max(|C_i \cap R_1|, |C_i \cap R_2|, ..., |C_i \cap R_r|)$$

$$\mathsf{Precision} = \frac{1}{n} \sum_{i=1}^{c} \mathsf{Precision}(C_i),$$

where n is the total number of samples.

[1] We perform the clustering in two steps: we first group the malware samples whose similarity is 1, and then calculate the similarity between every pair of the remaining samples.

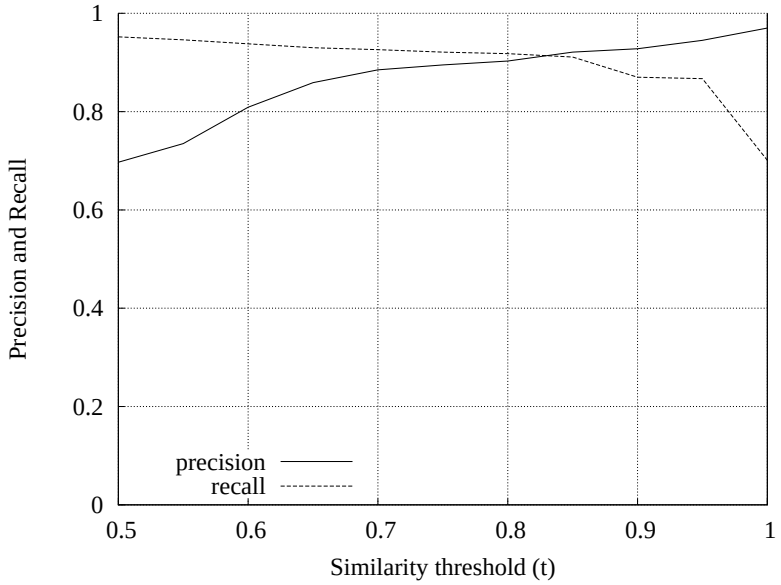

Figure 3: Precision and Recall

Recall measures the completeness of clustering, i.e., how well the clustering technique can group similar elements into the same group. Recall can be derived as follows:

$$\mathsf{Recall}(R_i) = \max(|C_1 \cap R_i|, |C_2 \cap R_i|, ..., |C_n \cap R_i|)$$

$$\mathsf{Recall} = \frac{1}{n}\sum_{i=1}^{r} \mathsf{Recall}(R_i).$$

We created a reference clustering based on the ClamAV virus names. The reference clustering has 1,958 clusters. Figure 3 depicts the Bitshred clustering quality compared to the ClamAV reference clustering with various threshold values. When $t = 0.85$, BitShred clustering produced 2,113 clusters with a precision of 0.921 and a recall of 0.911.

4 Future Work

4.1 Malware Clustering

To assess the quality of BitShred clustering, we compared the results to ClamAV scanning results. Different anti-virus tools, however, might report different virus names for the same suspicious file. To create a better reference clustering data, we might be able to use several anti-virus tools. In other words, after gathering scanning results from several anti-virus tools, we can create a reference clustering only with the samples reported as the same virus family by those tools.

We can further reduce the required storage space by combining both winnnowing and our algorithm. Winnowing decreases the number of hashes to be stored, which allows us to use smaller size of Bloom filters without increasing false positives.

4.2 Bug Finding

To verify that BitShred can find copy-pasted bugs at the binary level, we need to find the test cases such that several programs have the same buggy code. For example, in [2] a buffer overflow bug is reported in `xine-lib` library code, which is used in Xine, MPlayer, and VLC. Our initial results in finding bugs has been unsuccessful. We plan to investigate extensions in the future.

5 Related Work

Schlimer et al. [10] presented a new document fingerprinting algorithm, winnowing, to detect local matches. Winnowing first divides a document into shreds with size n. All shreds are hashed, and the selected subset of the hashes is the fingerprint to represent the corresponding document. Our approach can get the similar accuracy with less storage compared to winnowing. Moreover, our approach enables comparison between fingerprints to be done much faster.

Bayer et al. [3] proposed a scalable method to cluster a large set of malware samples. Their approach consists of three phases. At first, the analysis system collects execution traces of malware samples using dynamic analysis. Based on the execution traces collected at the first phase, a behavioral profile is extracted and then, malware samples that have similar behavioral profiles are grouped into the same cluster. Their approach has an additional overhead for executing programs to build behavioral profiles. On the other hand, our approach can create fingerprints for approximate 73 files per a second.

Li et al. [8] proposes a technique for detecting copy-paste related bugs in large software. For example, one main cause of copy-paste related bugs is that programmers may forget to change identifiers which should have been modified consistently after copy-pasting code fragments. Their tool, CP-Miner, found 49 unknown copy-paste related bugs in Linux. While CP-Miner is applied to source code to find copy-paste related bugs on source-code level using data mining, BitShred detects copy-pasted buggy code on binary level based on code plagiarism.

6 Conclusion

In this paper we present an algorithm for fast and scalable code reuse detection in binary code. Based on the ability to identify shared code segments, BitShred can be used in many security areas such as malware analysis and bug detection. We applied BitShred to cluster real malware samples and our experimental results show that BitShred can detect plagiarism among different malware samples and cluster them efficiently.

References

[1] CWSandbox. `http://www.cwsandbox.org/`.

[2] SecurityFocus. `http://www.securityfocus.com/bid/28312`.

[3] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. In *Proceedings of the Network and Distributed System Security Symposium*, February 2009.

[4] Burton H. Bloom. Space/Time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[5] Symantec Corporation. Symantec internet security threat report. `http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_internet_security_threat_report_xiii_04-2008.en-us.pdf`, April 2008.

[6] F-Secure. F-secure reports amount of malware grew by 100% during 2007. `http://www.f-secure.com/en_EMEA/about-us/pressroom/news/2007/fs_news_20071204_1_eng.html`.

[7] Jibz, Qwerton, snaker, and xineohP. PEiD. `http://www.peid.info/`.

[8] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32:176–192, 2006.

[9] Audris Mockus. Large-scale code reuse in open source software. In *Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development*, May 2007.

[10] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85, June 2003.