SFH: Hashing Unordered and Incomplete Network Streams on the Fly

Chao Zheng[1,2,3], Xiang Li[1,2], Qingyun Liu[1,2], Yong Sun[1], and Binxing Fang[4]

[1] Institute of Information Engineering, Chinese Academy of Sciences
[2] School of Cyber Security, University of Chinese Academy of Sciences
[3] National Engineering Laboratory for Information Security Technologies
[4] Institute of Electronic and Information Engineering of UESTC in Guangdong
`zhengchao@iie.ac.cn,liuqingyun@iie.ac.cn`

Abstract. The Deep Packet Inspection system usually uses a white/black MD5 list or regular expressions to identify viruses, malicious software or certain internal files from network traffic. Fuzzy hash, namely context triggered piecewise hash has the ability to compare two different files, and determine the similarity level. We focus on applying fuzzy hash on network traffic to identify files with real-time constraints, in which input is concurrency and stream based. In this paper, we present the SFH algorithm, which has the ability of hashing files on the fly, no matter whether the input is unordered, incomplete or of an initially undetermined length. SFH can generate a signature of appropriate length in a one-way process, and reduce the computational complexity from O(nlogn) to O(n). In particular, on a typical DPI scenario, SFH can hash files in a 68MB/s per CPU core, and consume no more than 5KB memory per file. To justify the effectiveness of SFH, we evaluated its performance on a public dataset. Compare to other fuzzy hash algorithms, SFH's precision and recall are not compromised for processing unordered and incomplete input.

Keywords: fuzzy hash, network traffic, approximate matching, file tracking

1 Introduction

Identifying content in transmission from network traffic is valuable for deep packet inspection (DPI) applications, such as malware detection, data leakage prevention (DLP), and Digital Forensics. These applications basically search for predefined signatures in the packet payload, which could be string patterns, cryptographic hash values, etc. For example, SNORT [23] inspects file content with regular expressions, and Suricata [10] computes MD5 checksum of the file. Furthermore, in order to deal with new security threats, many efforts have been devoted to identifying both similar and fragment files from network traffic, e.g. bloom filter [7], machine learning [8] and fuzzy fingerprint [30]. However, with the increasing amount of network throughput and new transmission optimization technologies, these solutions have become unfeasible. The reason is twofold.

First, DPI's scarcity computation resource is not allowed to run complicated algorithms on high throughput traffic. Second, emerging technologies such as multi-thread downloading, P2P file sharing and cyberlocker file upload make middlebox impractical to acquire an ordered file stream, neither during nor after the transmission.

Fuzzy hash is also known as context triggered piecewise hashes (CTPH), it basically slices input file to pieces by a context triggered algorithm and then hash each pieces individually. Compared to cryptographic hash algorithms such as MD5 and SHA1, fuzzy hash can still recognize a subtly changed file; for instance, inserting a character to a document. This feature makes fuzzy hash very appealing to DPI applications. But unfortunately, current fuzzy hash algorithms can only work on intact and stored files. On applying fuzzy hash to files in transmission, we found that the challenges were quite different from the conventional scenario, including incomplete capture, unordered fragments and a much higher memory consumption to buffer them.

In this paper, we propose an improved fuzzy hash algorithm, which can detect similar files from network traffic with constrains including real-time, incomplete input and memory efficient. For its feasible to apply on streamed and unordered data, we call it SFH (for **S**tream **F**uzzy **H**ash). SFH uses a compact structure to record computation context, and can hash unordered fragments individually with almost no buffer, and ultimately generate a proper length of signature in a one-way process. Our test shows SFH can hash data in 68MB/core/s in a typical multi-thread transfer scenario, and consume no more than 5KB memory per file which is irrelevant to file size. In addition, we applied SFH in practice to evaluate its effectiveness. The innovations of our work are:

- We discussed the difficulties of tracking files in real network traffic, including limited computation resource, an initially undetermined length, one-way processing, unordered input and incomplete capture.
- We tackled above challenges by proposing SFH, a fast and memory-efficient fuzzy hash algorithm that can deal with unordered file fragments individually with merely 6 bytes of buffer. In addition, comparing to original fuzzy hash, the time complexity of SFH is reduced from $O(n \log n)$ to $O(n)$.
- We evaluated SFH and three other fuzzy hash implementations, and our result shows that SFH achieved above goals without compromise of its ability on identifying similar files.

The SFH's ability of tracking files in transmission from network traffic could be very useful in network measurement, malicious software detection and data leakage protection, etc.

2 Preliminaries

In this section, we first introduce fuzzy hash algorithm, which is the primitive version of SFH. Then the Tillich-Zémor Hash, which is used as a strong hash in SFH for its concatenation property thus saving memory.

2.1 Fuzzy Hash Algorithm

Cryptographic hash algorithms like MD5 and SHA1 have good avalanche effect, which means flipping a single bit of a file should cause a drastic change in the cipher text. It's a desirable property of cryptographic algorithms for security purpose. But computer forensic investigators endeavor to find a hash algorithm that has the ability to compare two distinctly different items and determine a fundamental level of similarity. Context triggered piecewise hashes (CTPH), aka fuzzy hash, is one of the many options. Unlike cryptographic hash functions, it is not specifically designed to be difficult to reverse by an adversary, making it unsuitable for cryptographic purposes. The concept of fuzzy hash was invented by "spamsum" [33], or Nilsimsa [19] in another version, and then [17] formalized the method and developed `ssdeep` [14] to apply the algorithm to digital forensics.

The unique properties of fuzzy hash are described as below.

Non-propagation : In fuzzy hash, only the part of the signature that corresponds linearly with the changed part of the binary will be changed. This means that small changes in any part of the plain text will leave most of the signature the same.

Alignment robustness : Most hash algorithms are very alignment sensitive. Deleting or inserting a single byte to the plain text will generate a completely different hash value. The core of the fuzzy hash algorithm is a rolling hash used to produce a series of *reset points* in the binary. The *reset point* depends only on the immediate context. Fuzzy hash uses a variable named block size b to trigger reset point. The block size can be calculated with eq.1, that could ensure that the fuzzy hash result length of a certain file is neither too long to compare nor too short to avoid collisions. b_{min} is a constant minimum block size, S is a constant expected fuzzy hash length, n is an input file size. Note that file size is not always available in network traffic. We'll discuss this in Section 3.

$$b_{init} = b_{min} 2^{\left\lfloor log_2\left(\frac{n}{S b_{min}}\right)\right\rfloor} \quad (1)$$

With a rolling hash function of k window, given an input sequence of k bytes: $c_1c_2 \ldots c_k$ when

$$rolling\ hash\,(c_1c_2 \ldots c_k)\ \ mod\ b = b - 1 \quad (2)$$

is satisfied, a reset point will be positioned at c_k. Statistically, the smaller b is, the more reset points are triggered.

The stronger hash based on the *FNV* algorithm is then used to produce hash values of the areas between two reset points. The resulting signature comes from the concatenation of a single character from the FNV hash per reset point. Once the signature is produced, if its length is less than *S/2* characters. The fuzzy hash algorithm reduces the block size $b \leftarrow$ *b/2* and the algorithm is executed one more time until a signature of at least *S/2* characters is produced. Some researchers have proposed improvements to reduce such recalculation, but still couldn't eliminate the iterative processing [6] [2].

String edit distance algorithm will be used to measure different files' fuzzy hash value's similarity percentage.

2.2 Tillich-Zémor Hash

The Tillich-Zémor [31] hash function is defined by mapping each binary string to a matrix over a finite field of matrices with determinant 1. Each element in the alphabet is first mapped to a matrix from a generator set. The next step is to multiply corresponding matrices according to their order in the binary string. Its security under certain attacks is proven to be equivalent to associating to such a function a Cayley graph.

The TZ hash can be described as follows.

Defining parameter. An irreducible polynomial $P_n(X)$ of degree n in the range 130-170.

TZ hash algorithm. Let A and B be the following matrices.

$$A = \begin{pmatrix} X & 1 \\ 1 & 0 \end{pmatrix} \quad B = \begin{pmatrix} X & X+1 \\ 1 & 1 \end{pmatrix}$$

Define the mapping

$$\begin{gathered} \pi = \{0,1\} \to \{A,B\} \\ 0 \to A \\ 1 \to B \end{gathered}$$

The hash code of binary message $x_1 x_2 \dots x_k$ is just the matrix product

$$\pi(x_1)\,\pi(x_2)\dots\pi(x_k)$$

Where computations are made in the quotient field $F_{2^n} = F_2[X] \diagup P_n(X)$ of 2^n elements. Since Tillich-Zémor Hash uses group $SL_2(G)$ to present a bit of input data and multiply these matrices as the hash result, it has a concatenation property. This feature is fascinating to SFH, which will be detailed in Section 4.2. This is intended as an overview of the Tillich-Zémor hash, so some level of proving detail will remain outside the scope of this paper. It should be noted that there have been successful attacks on TZ hash function's collision resistance and pre-image resistance property [12] [22], but the vulnerability could be fixed by recent research [16] [15]. Thus, TZ hash is still strong enough for non-cryptographic purposes.

3 Challenges

As we mentioned before, fuzzy hash has been applied on many domains for its ability to compare two different files and determine a fundamental level of similarity. On applying fuzzy hash on network traffic to identify similar files in transmission, we encounter some new difficulties.

An initially undetermined size : File size may be undetermined until the end of a transmission. For example, if an HTTP session is non-keep-alive or chunked, content-length region is optional [9]. That's not a problem for hash algorithms like MD5 and SHA-1. However, file size is a crucial parameter to a fuzzy hash implementation, for it is as the input to produce a trigger value, that will be referred to as the *block size*, which was used to generate pieces.

One-way processing : In order to generate a signature with proper length, a fuzzy hash needs to adjust the rolling hash trigger value and calculate iteratively. As network traffic is high throughput, e.g. 10Gigabit Ethernet, in which it's impracticable to store the file content on the hard disk, so there is no second chance for recalculation.

Unordered input : The state-of-the-art transmission technologies split a file into fragments for transmission efficiency and agility, such as multi-thread downloading, P2P file sharing and cyberlocker service (e.g. Mega Upload and Baidu Cloud). Fig 1 is a typical multi-thread transfer scenario, where the grey block represents a file fragment. At time t_3, any file offset in range 0-3M may appear. Fuzzy hash algorithm can only calculate from the file head or a reset point, so that unordered fragments must be buffered until all preceding data is received. In the worst case, almost the whole file is buffered, which makes the memory consumption unacceptable.

Incomplete capture : It is a common observation that files captured from network traffic are incomplete due to packets loss or processing error. Packet loss is a common problem in middleboxes like Intrusion Detection System (IDS) or Data Leakage Protection (DLP) system, for they can't deal with a burst of network traffic or an attack. Meanwhile, there are also human factors that could compromise the integrity of the captured file. For example, dragging progress bar of an on-line video, or manually terminating the transfer session.

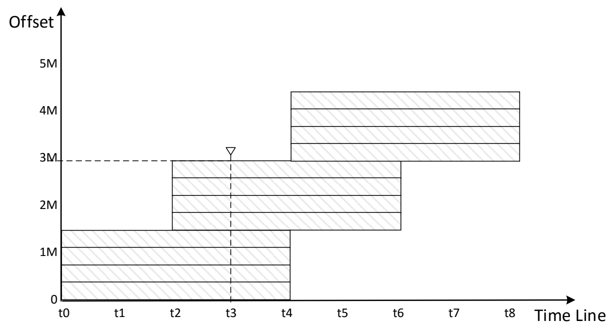


Fig. 1: Example of an out-of-sequence transfer.

4 Stream Fuzzy Hash

Based on the previous discussion, we now describe our design and implementation of Stream Fuzzy Hash, and the rationale for the design approach we adopt.

4.1 Overview

As we've introduced in Section 2.1, a fuzzy hash algorithm is composed of a rolling hash to generate reset points, and a stronger hash to produce hash values for each pieces between two reset points.

SFH uses a context to record each discrete data segment's calculation result. Considering that an unordered data segment is common, and each one will generate a segment contexts, we use an interval tree [1] to organize one file's several segment contexts efficiently. SFH uses the Tillich-Zémor hash instead of FNV as the strong hash function , which enables SFH buffer each discrete segment with no more than 6 bytes. Moreover, through a fine-grained adaptive mechanism, SFH could generate a fuzzy hash signature in a one-way process. If a file is confirmed incomplete after transmission, the missing parts of a file will not affect other intervals, for each signature is generated individually. As shown in Fig 2, we defined three basic operations on above data structure which include:

Updating the SFH segment context with an incoming data segment. There is no limitation on data size and beginning offset.

Merging the adjacent SFH segment contexts when an input data fills the gap of the interval tree.

Tuning the block size during hash if the current signature length exceeds the upper limit. That intend to generate a signature with proper length in a one-way process.

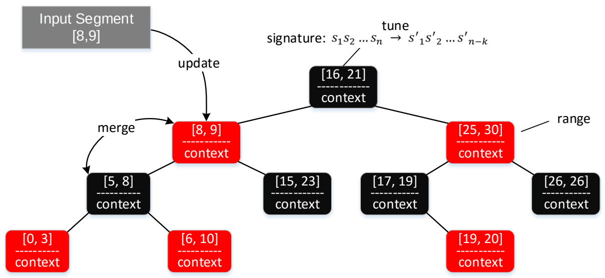


Fig. 2: Update, merge and tune operation in SFH.

4.2 Segment context

SFH uses a context to represent an in-progress fuzzy hash calculation of a data segment. This data segment may belong to any range of the original file, and have no minimum length limit. As shown in Fig 3, by running rolling hash on each byte of the segment, a data segment is sliced to marginal data, sliced data and truncated data accordingly.

1. Marginal data is the range before the first reset point that strong hash cannot be applied on.

2. Sliced data is the range between two reset points that are eligible for strong hash.
3. Truncated data is the range from the last reset point to the end that is also eligible for strong hash, but has an unfinished hash result.

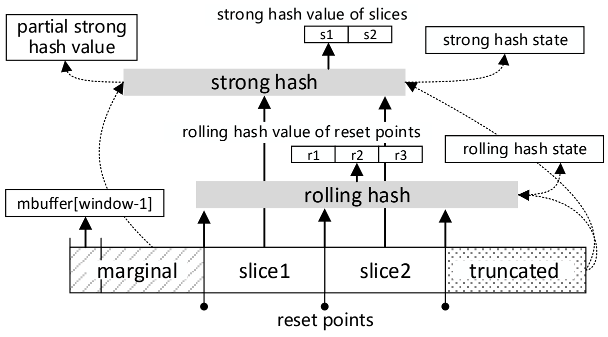


Fig. 3: A data segment's SFH segment context.

Most of the hash algorithm must hash the data from the beginning, so the marginal part before a reset point is not ready to make a strong hash, that's why we buffer it in the context. However, buffering the marginal data may cause severe memory overload; in the worst case, the whole file is buffered in the calculation contexts. This is unaffordable for a middle box when hashing multiple gigabit files.

The TZ(short for Tillich-Zémor) hash has two attractive features:

Concatenation property of partial TZ hash result is our favorite. This nature is benefited by TZ hash using group $SL_2(G)$ to present a bit of input data and multiply these matrices as the hash result. For example, the TZ hash process of $d_1d_2d_3$ in (3) can be computed individually.

$$hash_{TZ}(d_1d_2d_3) = hash_{TZ}(d_1d_2)\, hash_{TZ}(d_3) \tag{3}$$

This design is conducive to parallel computing. In SFH, the marginal part is separated to a partial TZ hash value and a buffer of size *rolling window-1*. In our implementation, it is 6 bytes for each discrete data segment.

Computational efficiency of TZ hash is another advantage we prefer. Matrix multipliers are made in the quotient field F_{2^n}, which can be easily computed in a few shifts and XOR's of 150-bit quantities per message bit. Because SFH is a non-cryptographic hash function and will reduce the strong hash result by recording only a base64 encoding of the six least significant bits (LS6B [33]) of each hash value, we choose n=8 instead of the range 130-170 to define the quotient field F_{2^n}. The irreducible polynomial we choose is eq.4.

$$F_{2^n} = x^8 + x^4 + x^3 + x^2 + 1 \tag{4}$$

Table 1: Notation of SFH context member

Symbol	Meaning
mbuff	marginal data buffer of context, may have reset point.
msize	buffered size of mbuff, up to 6 bytes
ps	partial strong hash value of unbuffered marginal data
$state_r$	rolling hash state of truncated data
$state_s$	strong hash state of truncated data, a matrix within group $SL_2(G)$
$array_r$	an array to store rolling hash values of reset points
$array_s$	an array to store strong hash matrixes between reset points
$backup_s$	a backup of $array_s$ before last tune operation

By looking up a Galois multiplication table, the computation could be efficient, which we'll discuss in Section 6. After invoking TZ hash, the members in an SFH context are described as in Table 1.

4.3 Context Updating

Before describing the update algorithm, we prefer to give a brief introduction to interval tree. An interval tree is a tree data structure to hold intervals. Specifically, it allows one to efficiently find all intervals that overlap with any given interval or point. We implement the interval tree based on red-black tree. Interval tree is dynamic data structure, that allows efficient insertion and deletion of an interval in $O(\log n)$. As we cannot allow intervals overlapped in the tree, the query time is $O(\log n)$ too.

When a new data segment comes, we first find its SFH segment context by querying the interval tree with the incoming data's start offset and end offset. If it's overlapped with any previous data segments, e.g. retransmission, for the convenience of computing, the new segment's duplicate part will be discarded. And then, the rest of input data is used to update the context as the orignal fuzzy hash, except the *FNV* was replaced by Tillich-Zémor hash.

4.4 Contexts Merging

The discrete calculation contexts in interval tree may be adjacent if an incoming segment filled the gap, then a merge operation was triggered. Merging the adjacent SFH segment contexts could decrease the node number of interval tree, and shorten the searching time. Fig 4 shows the process of merging two adjacent contexts, namely p and n. The basic idea lies in the associative law of TZ hash, that strong hash value of discrete fragments can be computed individually and concatenated when they are consecutive. Also, the rolling hash is continued with the 6 bytes in *mbuff*.

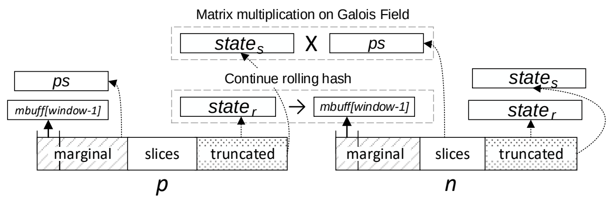


Fig. 4: Merging of two adjacent contexts.

4.5 Block size Tuning

The number of reset points generated by the rolling hash function is decided by three parameters, namely file length, randomness of file content and the block size. The randomness of file could be measured by the file's entropy. For comparing convenience, fuzzy hash should limit the signature length to a definite range. Original fuzzy hash achieves this goal by iteratively adjusting the initial block size and recalculating the hash until it gets a desirable one. But this method is infeasible for files in transmission as we've discussed in Section 3. On the one hand, the file length may be undetermined before the calculation, so we can't get an initial block size with eq.1. On the other hand, using a fixed block size makes signature length unpredictable and there's no second chance for recalculation in a one-way process.

SFH adopts an adaptive mechanism to tune the block size during the process of hashing. In the tuning process, each reset point's rolling hash value (stored in $array_r$) is tested by a new block size which is current block size times k.

It's easy to prove that:

$$r \bmod (k \times b) = k \times b - 1 \Rightarrow r \bmod b = b - 1 \tag{5}$$

Obviously, if there is a reset point fit for a new block size, then it must be one of the survived reset points. This guarantees that the final signature is irrelevant to the timing of tune operation. After a new reset point election, the temporary hash result is refined by multiplying each strong hash value between new reset points. We named parameter k in eq.5 as tune factor. Tune factor is also the parameter that determines whether a partial file could be compared or not. Bigger k will generate a longer signature, which we will discuss in Section 5.1.

The opportunity of tune operation is chosen carefully. The expected signature length is referred to as S. If the current reset point number is larger than $k \times S$, then a tune operation is necessary. But some files' entropy may be extremely low, like a string of repeated characters, thus producing less diversity of rolling hash values. Tuning block size on these hash values will cause a dramatic fall out of reset point number. To avoid this, we test every rolling hash value with a new block size before actual tuning. If the eligible reset point number is less than S, the tune operation will be aborted.

4.6 Generate SFH signatures

When we input all transferred data, SFH initiates an inorder traversal of the interval tree to visit every context. Each strong hash value is mapped to a base64 space with LS6B, the data range is printed in format [left offset - right offset] to mark gaps in passing. To enhance SFH's ability of partial file matching, we generate SFH signature with block size b and $k \times b$ (k is the tune factor) as the final output. Since the signature of block size b is a precedent result of $k \times b$ which is stored in $backup_s$ by tune operation, there is no double calculation. The format of SFH signature is $blocksize : hash_b : hash_{\frac{b}{k}}[range_{start} - range_{end}]$. Fig 5 is an example of SFH signature.

```
3072:Xk/maCm4yLYtRIFDFnVfHHqx1Jl+[0:432501]
7wr6Es3+TaKxONfbN[6130147:1160163]#12288:XCht
bFS6pHp9GZ[0:432501]lZ1hze2[6130147:1160163]
```

Fig. 5: SFH signature example, expected length=64 , block size=3072, tune factor=4.

Since SFH uses the same rolling hash function as `ssdeep`, for a given input, the two signatures should be the same length. Some files may not trigger proper amount of file pieces with any block size. `ssdeep` deal with this problem by combining the last few pieces of the message into a single piece. On the contrary, SFH chooses to keep all the pieces in separate and generate a longer signature to preserve more details of the input.

5 Signature Comparing

Since the SFH signature length depends on block size and the entropy of input, as a determinate file, we simply presume that the missing signature character number is in linear correlation to the missing length. s_1 is generated by the transferred file, so we use n_i spaces for each gap in s_1, where n_i is defined in eq.6. s'_1 represents the filled signature, and s_2 is the signature of intact target file. We use eq.7 to refine the Levenshtein distance result of s_1 and s_2. Similar to [17], the final match score in a range from 0 to 100 is calculated by eq.8. A higher match score indicates a greater probability that the source files have blocks of values in common and in the same order.

$$n_i = \left\lfloor \frac{GapBytes|s_1|}{ComputeBytes} \right\rfloor \tag{6}$$

$$e(s_1, s_2) = LE(s'_1, s_2) - \frac{|s_1|(|s'_1| - |s_1|)}{|s'_1|} \tag{7}$$

$$M = 100 - \frac{100e(s_1, s_2)}{|s'_1| + |s_2|} \tag{8}$$

5.1 Partial file matching

In SFH, two signatures are comparable if and only if they are generated by the same block size. As we've described in Section 4.5, tuning a block size is driven by input data, while data missing will postpone the tune operation. Apparently, a partial file may have a different block size from its origin. Although SFH generates a signature with block size b and b/k, it's still possible that a partial file has a different block size.

Theorem 1. *Assume the reset points are evenly distributed in the complete file, consider a partial file with a integrity rate of m, and a SFH tune factor k, the probability that partial file could be compared with its origin is*

$$p = \frac{k^2 m - 1}{k^2 m - km}, m \in \ (0, 1], k \in \mathbb{N}. \tag{9}$$

It might also be noted that the even distribution is just to simplify the problem; however, in practice, the distribution of reset points is strongly correlated to data set.

Proof. For an SFH desired length S, a complete file with a final block size b has two reset point numbers, one is L_b, and the other one is $L_{\frac{b}{k}}$ which is the precedent result of final tuning. L_b satisfied

$$S \le L_b \le kS \tag{10}$$

The partial file's signature is generated with block size *b'* and it's reset point number $L_{b'}$ satisfied

$$S \le L_{b'} \le kS \tag{11}$$

The partial file is comparable, when

$$b' = b \text{ or } b' = \frac{b}{k}$$

Based on eq.11, $b' = b$ is satisfied if and only if

$$S \le mL_b \le kS \Leftrightarrow \frac{S}{m} \le L_S \le \frac{kS}{m} \tag{12}$$

Similarly, $b' = \frac{b}{k}$ is satisfied if and only if

$$S \le mL_{\frac{b}{k}} \le kS \Leftrightarrow S \le mkL_b \le kS \Leftrightarrow \frac{S}{mk} \le L_b \le \frac{S}{m} \tag{13}$$

The concatenation of eq.12 and eq.13 is

$$\frac{S}{mk} \le L_b \le \frac{kS}{m}$$

As $kS \le \frac{kS}{m}, m \in \ (0, 1]$,above inequility is refined as

$$\frac{S}{mk} \le L_b \le kS \tag{14}$$

As we've assumed that the reset point is evenly distributed, in the precondition that L_b saisfies eq.10,the probability p that L_b also saisfies eq.14 is

$$p = \frac{kS - \frac{S}{mk}}{kS - S} = \frac{k^2m - 1}{k^2m - km} \tag{15}$$

We can use a bigger k to get a better comparable probability, and generate a longer signature, which will cause overhead on storage and comparison. Applications should make a trade-off between comparing efficiency and ability of partial matching. For example, with tune factor k=3 and integrity rate m=0.3,the comparable probability is 0.94. Based on theorem 1, when $m \geq \frac{1}{k}$, partial file is always comparable. Theorem 1 is also confirmed by our evaluation on real dataset in Section 6.4.

SFH's ability on partial file matching not only enables DPI to identify incomplete captured files, but also makes stopping the hazardous transmission possible.

5.2 Comparing with Massive Files

In practice, DLP and IPS systems may maintain a signature set of valuable files or malicious softwares, and the volume of the set could be millions. For example, NIST maintains a large public database of known content–the NSRL [20], which contains more than 50 million files. A naive solution to deal with such massive signatures is to compare all pairs by brute force, which is obviously impracticable and time consuming. Fortunately, the large scale approximate matching problem has been well studied in string similarity search area for many years [34]. We adopt a classical method of that area to speed up the comparison–index the SFH signatures with n-gram.

Building Index : n-gram is a contiguous sequence of n characters from a given sequence of text.The original design of fuzzy hash requires that similar hashes must have a common 7-gram. Thus each signature in the set is split into many 7-grams, treating each 7-gram as key and signature itself as value, then inserting the key-value pair to a hash table. To save space and speed up the querying on a multi-core system, techniques such as pruning and partition are also adopted. Due to space limitation, we will not go into those details here.

Querying Signature : The signature to be queried is also split into many 7-grams, which then looks up every 7-gram in the hash table to find candidate signatures. We chose 7-gram because the orignial fuzzy hash deliberately requires that similar hashes must have a common 7-gram. If any candidate signature sharing more than c 7-gram with the query, c is a threshold depending on the predefined similarity baseline, then eq. 8 is performed to determine their similarity.

6 Evaluation

In this section, we'll evaluate SFH's correctness, hash speed and signature length on `t5` corpus [11], which is an open data set built by [26]. `t5` contains 4,457 files

Table 2: Normalize TLSH distance to score range from 0 to 100 by related to `ssdeep` proximate false positive and recall rate.

`TLSH` distance	< 60	< 50	< 30	< 20	< 10	< 1
`ssdeep` score	> 0	> 30	> 70	> 80	> 90	100

and 1.8GB of data. SFH's performance was compared with `ssdeep v2.13` [14], `sdhash v3.4`[5] [24] and `TLSH v3.7.0` [18], which were the current latest versions at the time of the experiments. `ssdeep` is one of the de facto standard in the malware analysis area as it is currently the only similarity digest supported by Virus-Total [32]. `sdhash` is also a fuzzy hash implementation that has been widely applied. They are both supported by US NIST NSRL [20]. `TLSH` [21] is an open source fuzzy matching library that was developed by Trend Micro.

6.1 Correctness

We want to show SFH's capability to capture similarity between different files among different fuzzy hash algorithms. For similarity preserved hash algorithms, there can be false positives (non similar pairs returned as similar) as well as false negatives (similar pairs not returned as similar).

`ssdeep`, `sdhash` and SFH schemes provide a similarity score between two digests which ranges from 0 to 100, where 0 is a mismatch and 100 is a perfect match (or a near perfect match). A lower number means lower confidence level. However, the `TLSH` uses a different scheme to score the similarity between two digests - a distance score of 0 represents that the files are identical (or nearly identical) and scores above that represent greater distance between the documents. To compare these algorithms via a common basis, we normalized `TLSH` distance to a range from 0 to 100 based on the evaluation result by [21]. As shown in Table 2, `TLSH` distance and `ssdeep` score with proximate false positive and recall rate were considered the same.

Figure 6 shows the distribution of detected pairs on different scores in all scores.

Recall: As there are *n(n-1)/2* different pairs in a set containing *n* files, almost 10 million pairs in `t5`, it is not possible to determine all pairs by hand. Therefore, we make an assumption that the only positives are the ones discovered by either of the tools and that, if a correlation is not discovered by a tool, then it is non-existent. This works for our purposes as we are trying to describe the performance of SFH relative to other algorithms, rather than in absolute terms ground truth. Based on previous research [26] [21], we set a strict threshold value for each algorithm, below which we should ignore any positive results as the false positives rise to 10%. Using the threshold values in Table 3(`TLSH` threshold is not a score but a distance), the 4 algorithms detected 387 similar pairs in total.

[5] `sdhash v4.0` is the latest version, but Vassil Roussev(sdhash author) recommends `sdhash v3.4` to us.

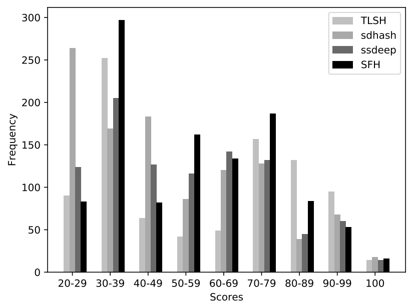


Fig. 6: Distribution of detected pairs on `t5`.

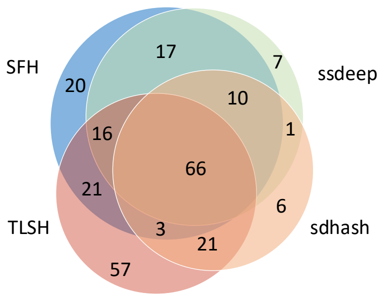


Fig. 7: The intersections of true positives sets on `t5`.

Table 3: Comparing 4 algorithm's precision and recall rate on `t5`.

	threshold	TP	FP	Precision	Recall
`TLSH`	20	146	95	60.6%	57.0%
`sdhash`	80	109	16	87.2%	42.6%
`ssdeep`	80	126	14	90.0%	49.2%
SFH	80	155	17	90.1%	60.5%
sum	-	256	131	-	-

Precision: We manually reviewed a total of 387 unique file pairs with the assitance of Beyond Compare [27], and 256 pairs are true positive.

We simply used the following definition for a correct similar pair:

1. text and html files that use a same boilerplate or share more than 10% common content.
2. pdf, doc, ppt and xls files are syntactic correlations beyond format.
3. jpg and gif files are similar in visual.

Note that the focus of this definition is not on determining the percentage of similar pairs on the `t5`, but to compare 4 algorithms on the same real-world data set. As shown in figure 7, the overlaps of the 4 algorithm varies, that's because the threshold we set is more rigorous than [26].For the readability, figure 7 does not show all the intersections.

To summarize,as shown in Table 3, the adoption of the TZ hash in SFH, does not decrease the recall and precision rate as a fuzzy hash implementation.

6.2 Hash speed on sequential input

In this section, we want to evaluate the hash speed of SFH, which is crucial in a DPI scenario. Besides above 4 algorithms, MD5 was also compared for giving readers an intuitive understanding of the speeds.

Table 4: Hash speed on sequential `t5`

	MD5	TLSH	sdhash	ssdeep	**SFH**
time(s)	2.62	149.53	60.70	31.03	27.01
speed(MB/s)	703	12	30	59	68

Table 5: SFH hash speed on `t5` multi-thread downloading

threads	4	8	16	random order
speed(MB/s)	67.8	67.8	67.6	61.3
space(KB)	2.39	3.40	4.90	310.2

The CPU is a multicore Intel Xeon E5-2698 v3 whose frequency is 2.30GHz. In our experiment, all the algorithms were executed on one logic core (Hyper-Threading enabled). The operating system is a Linux RedHat 7.2(kernel 3.10). We also use `t5` as the input of speed test. All the source codes of test algorithms are compiled with `gcc` "-O2" and configured as the default option. MD5 result was generated by OpenSSL v1.0.0. Every algorithm processed the files sequentially and the read chunk size is 4096 bytes. Table 4 shows their speeds on the test data.

We've noted the performance gap between cryptographic hash algorithm and fuzzy hash algorithm. We believe this was mainly because the SFH is not a block hash function, but rather each bit is hashed individually. By querying Galois multiplication table, we can hash 8-bit per call, but there is still more invocation cost than block hash. In MD5 and SHA1, the input message is broken up into chunks of 512-bit blocks, which `ssdeep` and SFH processes byte by byte.

Compared to `ssdeep`, it needs to recalculate an n-bytes input for $O(\log n)$ times to find a proper block size and $O(n)$ time on each calculation, making the total running time $O(n \log n)$. For SFH with block size tuning, no recalculation is needed after adjusting the block size, so the total running time is $O(n)$. But the complexity of the Tillich-Zémor hash weakens this advantage.

6.3 Hash speed on unordered input

As SFH is designed for hash files in transmission, we simulate the unordered input of multi-thread download. The chunk size is 1460 bytes so as to simulate a general TCP payload size. Sequential input is a precondition for original fuzzy hash and other cryptographic hash algorithm, so they are not comparable with SFH on this scenario. Table 5 shows the speed of SFH on different concurrent fragment numbers. We could sense no significant slowdown, even if the input order is completely random. The memory consumption on a 16 concurrent fragments is 4.90KB, which is affordable for a DPI middlebox.

6.4 Comparability of incomplete file

We've discussed that files captured from network traffic maybe incomplete due to many reasons. And we've proven that, if the product of integrity rate m and tune factor k is bigger than 1, partial file has a same block size as its origin. We want to evaluate this theory on real data.

For each file in `t5`, we only input a protion of $m(0{\sim}0.5)$ from the frist byte of the file to SFH. If the block size is same as the complete one, by design, they are comparable. As to the threshold of the score, the choice is user's. As shown in figure 8, the results mostly fit the theorem 1 speculation.

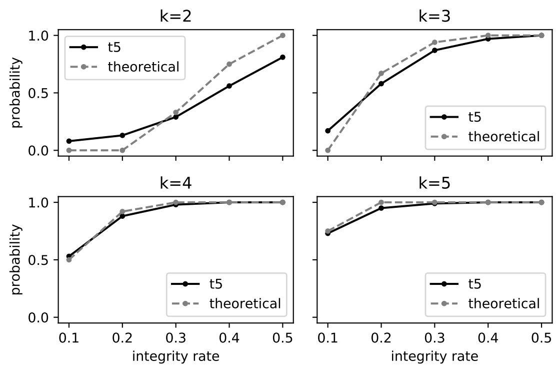


Fig. 8: Incomplete file comparable probability on `t5` in different tune factor k, the theoretical results were sampled with eq 9.

6.5 Deploying SFH in Practice

Having an intuitive understanding of the practicalities of SFH on network traffic, we present a deployment case. As a case study, we integrated SFH as a plugin to a carrier-grade DPI system to inspect malicious Android APP installation package in network traffic. It is worth mentioning that SFH is really flexible to utilize on a DPI system for no more buffering and rearranging.

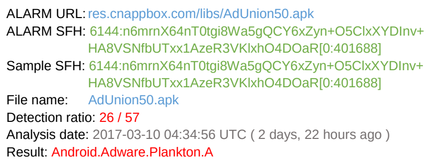
The detection proceeded as follows:

1. We first downloaded malicious Android APP samples from Virus Share [28], including 35397 files with a total size 52.82GB, and ran SFH signature on each file.
2. For comparing efficiency, we indexed the 35397 signatures with the approach in Section 5.2.
3. The DPI system was deployed in an anonymous ISP and processed about 10Gbps traffic. The plugin identifies an APP package transmission with its URL. Then the plugin input the subsequent packets this HTTP session to SFH. Thus, a data structure contains SFH signature and URL was generated.

4. The last step is querying the APP's signature in the previously built index to measure similarity. If the distance (eq 7) is less than a threshold, an alarm log is generated.

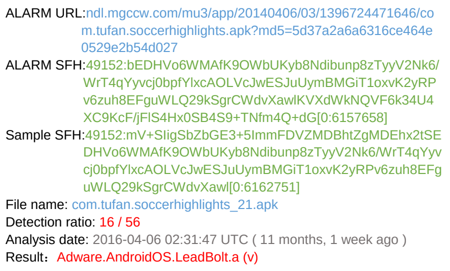
In our experiment, the DPI system processed 184,688 Android APK downloads, and got 14 alarm logs which related to 10 apps. To verify the result, we tried to retrieve suspect APP packages with the recorded URLs, and uploaded to VirusTotal [32] for further examination.

After all, among four successful downloaded apps, two were identified as malicious. The results are shown in Fig 9. We are gratified to see that *Football Highlights* is detected by similarity, which conventional methods are incapable of. The visual check on two false positive app binaries shows that, both of their file contents were quite similar to the malicious sample. More precisely, they shared 95% mutual content and presumably used same development components. This defect inherits from the basic idea of approximate matching but not SFH.



```
ALARM URL: res.cnappbox.com/libs/AdUnion50.apk
ALARM SFH: 6144:n6mrnX64nT0tgi8Wa5gQCY6xZyn+O5ClxXYDInv+
           HA8VSNfbUTxx1AzeR3VKlxhO4DOaR[0:401688]
Sample SFH: 6144:n6mrnX64nT0tgi8Wa5gQCY6xZyn+O5ClxXYDInv+
           HA8VSNfbUTxx1AzeR3VKlxhO4DOaR[0:401688]
File name:  AdUnion50.apk
Detection ratio: 26 / 57
Analysis date: 2017-03-10 04:34:56 UTC ( 2 days, 22 hours ago )
Result: Android.Adware.Plankton.A
```

(a) Suck Ads



```
ALARM URL:ndl.mgccw.com/mu3/app/20140406/03/1396724471646/co
          m.tufan.soccerhighlights.apk?md5=5d37a2a6a6316ce464e
          0529e2b54d027
ALARM SFH:49152:bEDHVo6WMAfK9OWbUKyb8Ndibunp8zTyyV2Nk6/
          WrT4qYyvcj0bpfYlxcAOLVcJwESJuUymBMGiT1oxvK2yRP
          v6zuh8EFguWLQ29kSgrCWdvXawlKVXdWkNQVF6k34U4
          XC9KcF/jFlS4Hx0SB4S9+TNfm4Q+dG[0:6157658]
Sample SFH:49152:mV+SIigSbZbGE3+5ImmFDVZMDBhtZgMDEhx2tSE
          DHVo6WMAfK9OWbUKyb8Ndibunp8zTyyV2Nk6/WrT4qYyv
          cj0bpfYlxcAOLVcJwESJuUymBMGiT1oxvK2yRPv6zuh8EFg
          uWLQ29kSgrCWdvXawl[0:6162751]
File name: com.tufan.soccerhighlights_21.apk
Detection ratio: 16 / 56
Analysis date: 2016-04-06 02:31:47 UTC ( 11 months, 1 week ago )
Result： Adware.AndroidOS.LeadBolt.a (v)
```

(b) Football Highlights

Fig. 9: Two SFH positive APP packages that examined by *Virus Total*.

7 Related Work

Fuzzy hash algorithm has evolved for many years. [17] developed an open source fuzzy hash tool named `ssdeep` [14], which was used to find similarity files in specified sets. [6] and [2] proposed approaches to improve the performance of fuzzy hash. So far, the challenges of applying fuzzy hash on network traffic are not considered.

`sdhash` [25] is another widely applied approach for similar file detection. It tries to find from every neighborhood the features that have the lowest empirical probability of being encountered by chance. Each of the selected features is hashed and placed into a Bloom filter. When a filter reaches its capacity, a new filter is created until all the features are accommodated. Thus, a similarity digest consists of a sequence of Bloom filters. `sdhash` digest length is about 2∼3%

of the length of the input, which is different from fuzzy hash's bounded digest length(64∼128 bytes). For it retains more details of the original file, `sdhash` is better at embedded object detection than `ssdeep` [26]. However, more details bring overheads on storage and comparing, e.g. `sdhash` generated a 101GB digests for 50 million NSRL [20] files, by contrast, same digests that was generated by `ssdeep` is 1.2 GB.

MinHash [4] and SimHash [5] have been widely adopted in industry to find out near duplicated text files. They all belong to locality sensitive hash (LSH) algorithms. [29] claimed that MinHash outperforms SimHash in binary data, but curiously, researchers rarely use LSH in digital forensic area. In fact, [13] pointed out that LSH aims at mapping similar objects into the same bucket, while approximate matching outputs from a similarity digest that is comparable. There are also some open source tools for similar file detection, such as Nilsimsa [19], TLSH [21],mrsh-v2 [3] etc.

8 Limitations

Although SFH is used in security applications, it's definitely not cryptographic secure. Moreover, SFH is incapable of identifying files that were compressed or encrypted. As a hash functions, SFH's limitations are twofold.

Possibility of Collision : As the original fuzzy hash algorithm, SFH mapped a file piece to a 6-bit value, so it is possible that two distinct pieces map to the same character. Moreover, it is possible that two files can have identical SFH signatures but still be different files. As Kornblum [17] had discussed, the probability that it will fail to detect a change is 2^{-12} to 2^{-6}. And for two completely random files with a signature length S, the odds of an exact match are $\left(2^{-6}\right)^{S}$. We can increase the expected signature length to reduce such collision.

Signature Comparability : A meaningful comparison can only be performed on files with same block size. For different files approximately matching, that's not a problem but a useful feature because different block size means two files are quite different both in size and content. For partial file matching, as we've discussed in Section 5.1, the cut off of *1/k* file is always compared, where k is the tune factor introduced in Section 4.5. Below this level, the block sizes are too different to make a meaningful comparison. Embedded object detection is similar for the small embedded object could be considered as a partial part of the bigger file.

9 Conclusion

In this paper, we have presented the SFH algorithm, which is used for hash files from network traffic in real-time. Based on context triggered piecewise hash concept, SFH uses the Tillich-Zémor hash as a strong hash function and interval tree to index calculation contexts, which make it perfectly suitable for unordered and incomplete input. With block size tuning, SFH can hash data stream on a one-way processing. Besides, compared to `ssdeep`, SFH reduces computation complexity from $O(n \log n)$ to $O(n)$. Our evaluation shows that SFH's speed is 68MB/s and it consumes 5KB memory. And compared to other fuzzy hash algorithms, SFH's precision and recall are not compromised for processing unordered and incomplete input. To demonstrate its deployment, we integrate SFH into a DPI system to inspect malicious Android APP packages, and it showed a good usability in practice.

Additionaly, DPI system equiped with SFH can identify valuable files from egress traffic and malicious software from ingress traffic. Another benfit from SFH's ability of partial file matching on DPI systems is that it is even possible to identify files before transmission completes and stop the attack in action.

References

1. *Introduction to Algorithms (3rd ed.).* MIT Press and McGraw-Hill, 2009.
2. Frank Breitinger and Harald Baier. Performance issues about context-triggered piecewise hashing. In *International Conference on Digital Forensics and Cyber Crime*, pages 141–155. Springer, 2011.
3. Frank Breitinger and Harald Baier. Similarity preserving hashing: Eligible properties and a new algorithm mrsh-v2. In *International Conference on Digital Forensics and Cyber Crime*, pages 167–182. Springer, 2012.
4. Andrei Z Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings*, pages 21–29. IEEE, 1997.
5. Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388. ACM, 2002.
6. Long Chen and Guoyin Wang. An efficient piecewise hashing method for computer forensics. In *Knowledge Discovery and Data Mining, 2008. WKDD 2008. First International Workshop on*, pages 635–638. IEEE, 2008.
7. Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, and John Lockwood. Deep packet inspection using parallel bloom filters. In *High performance interconnects, 2003. proceedings. 11th symposium on*, pages 44–51. IEEE, 2003.
8. Yuval Elovici, Asaf Shabtai, Robert Moskovitch, Gil Tahan, and Chanan Glezer. Applying machine learning techniques for detection of malicious code in network traffic. In *Annual Conference on Artificial Intelligence*, pages 44–50. Springer, 2007.
9. R Fielding, James Gettys, Jeffrey C Mogul, Henrik Frystyk Nielsen, and Larry Masinter. Hypertext transfer protocol http /1.1, rfc2616, 1999.
10. Open Information Security Foundation. Suricata, 2017.
11. GovDocs. t5 corpus code, 2011.
12. Markus Grassl, Ivana Ilić, Spyros Magliveras, and Rainer Steinwandt. Cryptanalysis of the tillich–zémor hash function. *Journal of Cryptology*, 24(1):148–156, 2011.
13. Vikram S Harichandran, Frank Breitinger, and Ibrahim Baggili. Bytewise approximate matching: The good, the bad, and the unknown. *The Journal of Digital Forensics, Security and Law: JDFSL*, 11(2):59, 2016.
14. jessekornblum. ssdeep source code, 2015.
15. KT Joju and PL Lilly. Preimage of tillich–zemor hash function with new generators. *International Journal of Applied Mathematical Sciences, ISSN*, pages 4237–4248, 2013.

16. KT Joju and PL Lilly. Improved form of tillich-zemor hash function. *International Journal of Theoretical Physics and Cryptography*, 6, 2014.
17. Jesse Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital investigation*, 3:91–97, 2006.
18. Trend Micro. Tlsh code, 2017.
19. Nilsimsa. Nilsimsa v.0.2.4, 2001.
20. US NIST. National software reference library, 2013.
21. Jonathan Oliver, Chun Cheng, and Yanggui Chen. Tlsh–a locality sensitive hash. In *Cybercrime and Trustworthy Computing Workshop (CTC), 2013 Fourth*, pages 7–13. IEEE, 2013.
22. Christophe Petit and Jean-Jacques Quisquater. Preimages for the tillich-zémor hash function. In *International Workshop on Selected Areas in Cryptography*, pages 282–301. Springer, 2010.
23. Martin Roesch et al. Snort: Lightweight intrusion detection for networks. In *Lisa*, volume 99, pages 229–238, 1999.
24. V Roussev. Sdhash version 3.4, 2013.
25. Vassil Roussev. Data fingerprinting with similarity digests. In *IFIP International Conference on Digital Forensics*, pages 207–226. Springer, 2010.
26. Vassil Roussev. An evaluation of forensic similarity hashes. *digital investigation*, 8:S34–S41, 2011.
27. Inc Scooter Software. Beyond compare, 2017.
28. Virus Share. https://virusshare.com/, 2017.
29. Anshumali Shrivastava and Ping Li. In defense of minhash over simhash. In *AISTATS*, pages 886–894, 2014.
30. Xiaokui Shu and Danfeng Daphne Yao. Data leak detection as a service. In *International Conference on Security and Privacy in Communication Systems*, pages 222–240. Springer, 2012.
31. Jean-Pierre Tillich and Gilles Zémor. Hashing with sl 2. In *Advances in Cryptology CRYPTO 94*, pages 40–49. Springer, 1994.
32. Virus Total. https://www.virustotal.com/, 2017.
33. Andrew Tridgell. Spamsum readme, 2002.
34. Sebastian Wandelt, Dong Deng, Stefan Gerdjikov, Shashwat Mishra, Petar Mitankin, Manish Patil, Enrico Siragusa, Alexander Tiskin, Wei Wang, Jiaying Wang, et al. State-of-the-art in string similarity search and join. *ACM SIGMOD Record*, 43(1):64–76, 2014.