Chapter 12

HASHING INCOMPLETE AND UNORDERED NETWORK STREAMS

Chao Zheng, Xiang Li, Qingyun Liu, Yong Sun and Binxing Fang

Abstract Deep packet inspection typically uses MD5 whitelists/blacklists or regular expressions to identify viruses, malware and certain internal files in network traffic. Fuzzy hashing, also referred to as context-triggered piecewise hashing, can be used to compare two files and determine their level of similarity. This chapter presents the stream fuzzy hash algorithm that can hash files on the fly regardless of whether the input is unordered, incomplete or has an initially-undetermined length. The algorithm, which can generate a signature of appropriate length using a one-way process, reduces the computational complexity from $O(n \log n)$ to $O(n)$. In a typical deep packet inspection scenario, the algorithm hashes files at the rate of 68 MB/s per CPU core and consumes no more than 5 KB of memory per file. The effectiveness of the stream fuzzy hash algorithm is evaluated using a publicly-available dataset. The results demonstrate that, unlike other fuzzy hash algorithms, the precision and recall of the stream fuzzy hash algorithm are not compromised when processing unordered and incomplete inputs.

Keywords: Fuzzy hashing, network traffic, approximate matching, file tracking

1. Introduction

Identifying content in network traffic is important in deep packet inspection applications such as malware detection, data leakage prevention and digital forensics. In these applications, a search is made for predefined signatures in packet payloads, which could be string patterns, cryptographic hash values, etc. For example, Snort inspects file content using regular expressions and Suricata (`suricata-ids.org`) computes the MD5 checksums of files.

In order to deal with new security threats, several researchers have focused on identifying similar files and file fragments in network traffic

G. Peterson and S. Shenoi (Eds.): Advances in Digital Forensics XIV, IFIP AICT 532, pp. 199–224, 2018.
https://doi.org/10.1007/978-3-319-99277-8_12

using techniques such as Bloom filters [1], machine learning [8] and fuzzy fingerprints [21]. However, increased network throughput and network transmission optimization technologies have rendered these approaches infeasible. Scarce computational resources prevent the execution of complex algorithms on high throughput traffic in deep packet inspection applications. Additionally, emerging technologies such as multi-thread downloading, P2P file sharing and cyberlocker file uploading make it impractical to acquire ordered file streams during or even after transmission.

Fuzzy hashing, also referred to as context-triggered piecewise hashing (CTPH), essentially slices an input file into pieces using a context-triggered algorithm and hashes each piece. Compared with cryptographic hash algorithms such as MD5 and SHA1, fuzzy hashing can recognize files that are changed in a subtle manner (e.g., by inserting just a single character in a document). This feature makes fuzzy hashing very appealing in deep packet inspection applications. However, current fuzzy hash algorithms only work on intact and stored files. Indeed, when applying fuzzy hashing to files in transmission, a number of challenges are encountered that are quite different from conventional scenarios. The challenges include incomplete capture, unordered fragments and very high buffering requirements.

This chapter proposes an improved fuzzy hash algorithm that can detect similar files in network traffic under real-time, incomplete input and limited memory constraints. This so-called stream fuzzy hash (SFH) algorithm can be applied to streamed and unordered data. The algorithm employs a compact structure to record the computed contexts and hashes unordered fragments individually with almost no buffering, generating a proper-length signature via a one-way process. Experiments demonstrate that the stream fuzzy hash algorithm can hash data at 68 MB/core/s in a typical multi-thread transfer scenario while consuming no more than 5 KB memory per file regardless of the file size. The ability of the stream fuzzy hash algorithm to track files transmitted in network traffic is very useful in network measurement, malicious software detection and data leakage protection applications.

2. Preliminaries

This section discusses the fuzzy hash algorithm, which is the primitive version of the stream fuzzy hash algorithm presented in this chapter. Next, it presents the Tillich-Zémor (TZ) hash that is used as a strong hash by the stream fuzzy hash algorithm due to its concatenation property, which saves memory.

2.1 Fuzzy Hashing

Cryptographic hash algorithms such as MD5 and SHA1 have good avalanche effects, which means that flipping a single bit of a file causes drastic changes to its hash value. While this is a highly desirable security property of cryptographic hash algorithms, digital forensic investigators are interested in a hash algorithm that can be used to compare two distinctly different items and determine their fundamental level of similarity. Context-triggered piecewise hashing, also referred to as fuzzy hashing, is one of many such options. Unlike a cryptographic hash, a fuzzy hash is not designed to be difficult to reverse by an adversary, making it unsuitable for cryptographic purposes. The concept of fuzzy hashing was pioneered in `spamsum` [23] and Nilsimsa [25]. Kornblum [14] formalized the concept and developed `ssdeep` for use in digital forensics.

Non-propagation is a property unique to fuzzy hashing. In a fuzzy hash, only the part of the signature that corresponds linearly to the changed part of the binary is changed. This means that a small change to any part of the plaintext will leave most of the signature intact.

Alignment robustness is another important property. Most hash algorithms are very alignment sensitive. Deleting or inserting a single byte in the plaintext generates a completely different hash value. The core of the fuzzy hash algorithm is a rolling hash that produces a series of reset points in the binary. Each reset point depends only on the immediate context.

Fuzzy hashing uses a block size variable b to trigger a reset point. The block size is computed using the following equation:

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

where b_{min} is the (constant) minimum block size, S is the (constant) expected fuzzy hash length and n is the input file size. The equation helps ensure that the fuzzy hash result of a given file is neither too long for efficient comparisons nor too short to avoid collisions. Note that the file size is not always available in network traffic; this issue, which poses some challenges, is discussed later.

In the case of a rolling hash function with window k, if an input sequence of k bytes $c_1 c_2 \ldots c_k$ satisfies the condition:

$$\text{rolling hash}\,(c_1 c_2 \ldots c_k) \mod b = b - 1 \tag{2}$$

then a reset point is positioned at c_k. Statistically, the smaller the value of b, the greater the number of reset points that are triggered.

The strong hash based on the Fowler-Noll-Vo (FNV) algorithm is used to produce hash values of the regions between two reset points. The resulting signature comes from the concatenation of a single character from the Fowler-Noll-Vo hash per reset point. A signature is produced if the length is not more than $S/2$ characters. Specifically, the fuzzy hash algorithm reduces the block size from b to $b/2$ and the algorithm is executed iteratively until a signature of at least $S/2$ characters is produced. Some researchers [2, 6] have proposed improvements to reduce the computations, but the iterative processing has not been eliminated. The string edit distance algorithm is used to measure the similarity percentage of fuzzy hash values computed for different files.

2.2 Tillich-Zémor Hash

The Tillich-Zémor [22] hash function maps a binary string to a matrix over a finite field of matrices with determinant one. Each element in the alphabet is first mapped to a matrix from a generator set. Next, the corresponding matrices are multiplied according to their order in the binary string. The security of the Tillich-Zémor hash to certain attacks has been proven to be equivalent to associating a Cayley graph with the hash function.

The Tillich-Zémor hash has: (i) a defining parameter; and (ii) a hash algorithm:

- **Defining Parameter:** This parameter is an irreducible polynomial $P_n(X)$ of degree n in the range 130 to 170.

- **Tillich-Zémor Hash Algorithm:** Let A and B be the matrices:

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

  and let $\pi = \{0,1\} \rightarrow \{A, B\}$ be a mapping where $0 \rightarrow A$ and $1 \rightarrow B$. Then, the hash code of a binary message $x_1 x_2 \ldots x_k$ is the matrix product:

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

  where the computations are performed in the quotient field $F_{2^n} = F_2[X]/P_n(X)$ of 2^n elements.

Since the Tillich-Zémor hash uses the group $SL_2(G)$ to present a bit of input data and multiply matrices to produce the hash result, it has the concatenation property. This feature is of interest to the stream fuzzy hash algorithm presented in this chapter.

Note that this presentation only provides an overview of Tillich-Zémor hashing, so mathematical details and proofs are omitted. However, it should be noted that several attacks have targeted the collision resistance and pre-image resistance properties of the Tillich-Zémor hash function [10, 17]. Fortunately, the vulnerabilities have been addressed in recent research [12, 13]. As a result, the Tillich-Zémor hash is still strong enough for non-cryptographic purposes.

3. Challenges

As mentioned above, fuzzy hashing has been applied in many domains to determine the fundamental level of similarity between a pair of files. However, several challenges are encountered when applying fuzzy hashing to network traffic in order to identify similar files in transmission.

One challenge is that the file size is not known initially; typically, it cannot be determined until the end of a transmission. For example, if an HTTP session is non-keep-alive or chunked, the content-length region is optional [9]. This is not a problem for hash algorithms such as MD5 and SHA-1 that do not require the file size. However, the file size is a crucial parameter in a fuzzy hash implementation. This is because it is used as the input to produce a trigger value (or block size) for generating pieces for hashing.

Another challenge is one-way processing. In order to generate a signature of the proper length, fuzzy hashing must adjust the rolling hash trigger value and calculate it iteratively. Because network traffic often has high throughput (e.g., 10 Gb Ethernet), it is impractical to store the file content, so there is no opportunity for recomputation.

Unordered input also poses a challenge. State-of-the-art networking technologies split a file into fragments for transmission efficiency and agility, such as multi-thread downloading, P2P file sharing and cyberlocker services. Figure 1 shows a typical multi-thread transfer scenario, where the grey block represents a file fragment. At time t_3, any file offset in the range 0–3 M may appear. The fuzzy hash algorithm can only perform its computations from the file header or a reset point, meaning that unordered fragments must be buffered until all the preceding data has been received. In the worst case, almost the entire file has to be buffered, which makes the memory consumption unacceptable.

Finally, incomplete capture is a challenge. Files captured from network traffic are often incomplete due to packet loss and processing errors. Packet loss is a common problem in intrusion detection and data leakage protection systems because they cannot deal with bursts of network traffic or attacks. A human user may also compromise the integrity of a captured file, for example, by dragging the progress bar on an on-line video or manually terminating a transfer session.

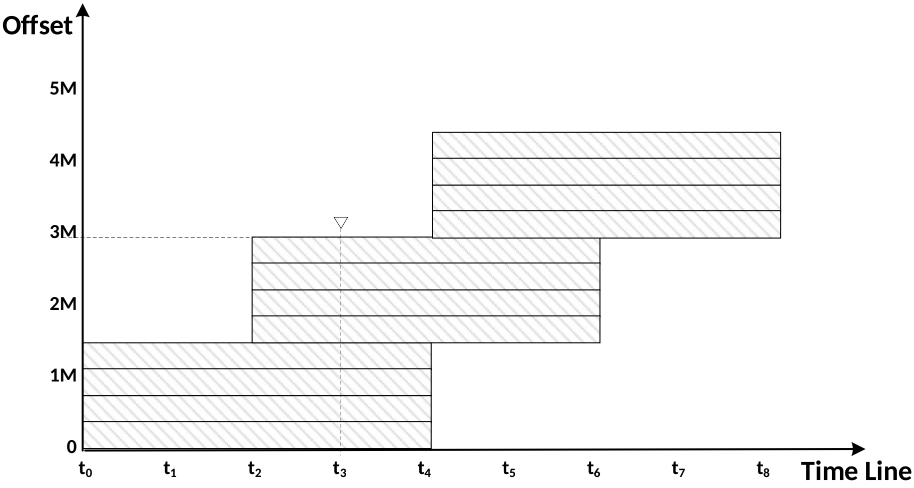


Figure 1. Example of an out-of-sequence transfer.

4. Stream Fuzzy Hashing Algorithm

This section describes the design and implementation of the stream fuzzy hash algorithm.

4.1 Overview

As mentioned above, a fuzzy hash algorithm uses a rolling hash to generate reset points and a strong hash technique to produce hash values for each piece between two reset points. The stream fuzzy hash algorithm uses a context to record the computational result of each discrete data segment. Since unordered data segments are common and each segment generates a segment context, an interval tree [7] is used to efficiently organize the file segment contexts. The stream fuzzy hash algorithm uses the Tillich-Zémor hash instead of the Fowler-Noll-Vo hash, enabling each discrete segment to be buffered using no more than six bytes. A fine-grained adaptive mechanism is employed to generate a fuzzy hash signature via a one-way process. If a file is confirmed to be incomplete after transmission, the missing parts of a file do not affect other intervals because each signature is generated individually.

Figure 2 illustrates the three basic operations in the stream fuzzy hash algorithm:

- **Updating:** This operation updates the segment context with an incoming data segment. There are no limitations on the data size and starting offset.

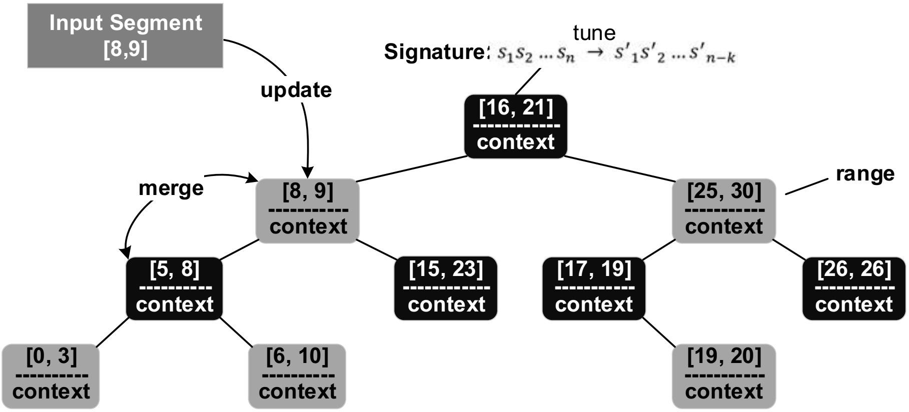


Figure 2. Stream fuzzy hash algorithm operations.

- **Merging:** This operation merges adjacent segment contexts when an input data fills the gap of the interval tree.
- **Tuning:** This operation tunes the block size during hashing if the current signature length exceeds the upper limit. This enables the generation of a signature with the proper length via a one-way process.

4.2 Segment Contexts

The stream fuzzy hash algorithm uses a context to represent the fuzzy hash computation of a data segment while it is being processed. The data segment, which may belong to any part of the original file, has no minimum length limit. As shown in Figure 3, a rolling hash is computed for each byte of a data segment with marginal data, sliced data or truncated data:

- Marginal data comprises data in the range before the first reset point to which a strong hash cannot be applied.
- Sliced data comprises data in the range between two reset points to which a strong hash can be applied.
- Truncated data comprises data in the range from the last reset point to the end to which a strong hash can be applied, but the result is an incomplete hash.

A hash algorithm must hash data from the beginning of a file. Since a strong hash cannot be applied to the marginal data portion before a reset

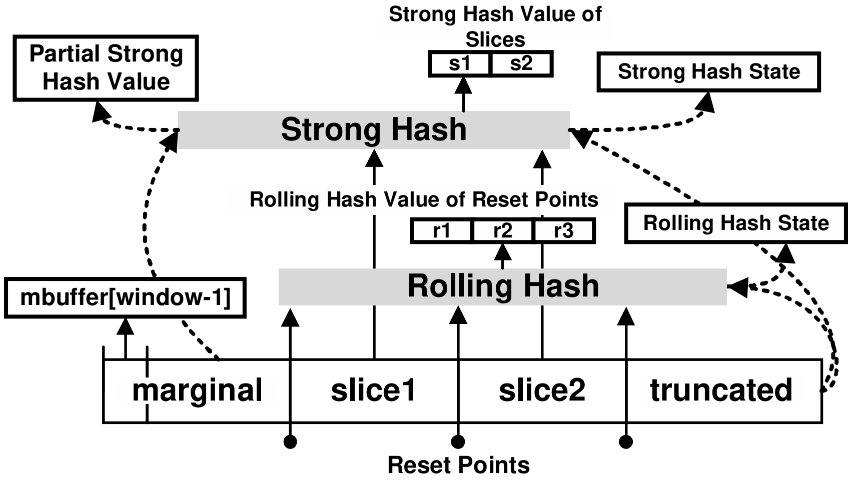


Figure 3. Data segment context.

point, the data is buffered in the context. However, buffering marginal data can cause a severe memory overload; in the worst case, the entire file has to be buffered. This is infeasible when hashing multi-gigabit files.

The Tillich-Zémor hash used in the stream fuzzy hash algorithm has two attractive features:

- **Concatenation Property:** Partial Tillich-Zémor hashing satisfies the concatenation property. This is because the Tillich-Zémor hash uses the group $SL_2(G)$ to present a bit of input data and multiplies matrices to produce the hash result. For example, the Tillich-Zémor hash of $d_1d_2d_3$ can be computed individually as follows:

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

  This design is conducive to parallel computing. In the stream fuzzy hash algorithm, the marginal part is separated into a partial Tillich-Zémor hash value and a buffer of size $rollingwindow - 1$ (six bytes for each discrete data segment in the algorithm implementation).

- **Computational Efficiency:** Tillich-Zémor hashing is computationally efficient. Matrix multiplications are performed in the quotient field F_{2^n}, which are easily computed using a few shifts and XORs of 150-bit quantities per message bit. Because the stream fuzzy hash is a non-cryptographic hash function, the strong hash

Table 1. Context components.

Symbol	Description
mbuff	Marginal data buffer of context, may have a reset point
msize	Size of *mbuff* buffer, up to six 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 in group $SL_2(G)$
$array_r$	Array storing rolling hash values of reset points
$array_s$	Array storing strong hash matrices between reset points
$backup_s$	Backup of $array_s$ before the last tuning operation

result is reduced by recording only a Base64 encoding of the six least significant bits (LS6B [23]) of each hash value; $n = 8$ is used instead of the range 130–170 to define the quotient field F_{2^n}. The following irreducible polynomial is employed:

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

Table 1 presents the context components produced after Tillich-Zémor hashing.

4.3 Context Updating

An interval tree is employed for context updating. It is a tree data structure that holds intervals such that all the intervals that overlap with any given interval or point can be determined efficiently. In this work, the interval tree implementation is based on a red-black tree. This dynamic data structure enables the efficient insertion and deletion of an interval in $O(\log n)$. Because intervals in the tree cannot overlap, the query time is also $O(\log n)$.

When a new data segment is received, the stream fuzzy hash algorithm finds the segment context by querying the interval tree with the starting offset and ending offset of the data segment. If the new data segment overlaps with a previous data segment (e.g., due to retransmission), for convenience of computing, the duplicate portion of the new segment is discarded. The remaining input data is used to update the context as in the original fuzzy hash algorithm, except that the stream fuzzy hash algorithm replaces the Fowler-Noll-Vo hash with the Tillich-Zémor hash.

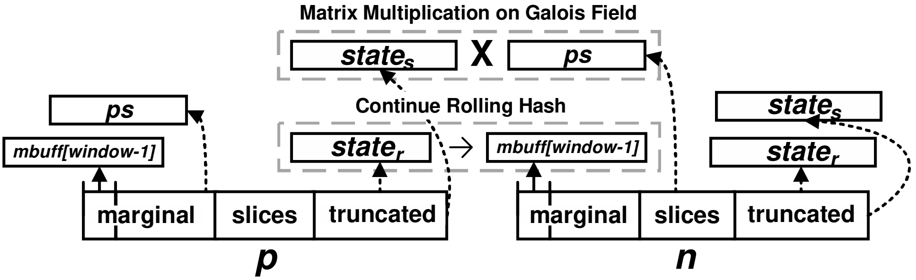


Figure 4. Merging two adjacent contexts.

4.4 Context Merging

Discrete segment contexts in the interval tree could become adjacent if an incoming segment fills the gap; this situation triggers the merging operation. Merging adjacent segment contexts decreases the number of nodes in the interval tree, thereby reducing the search time.

Figure 4 illustrates the process of merging two adjacent contexts p and q. It is based on the associative property of the Tillich-Zémor hash, where strong hash values of discrete fragments can be computed individually and concatenated when they are consecutive. Also, the rolling hash is continued with the six bytes in *mbuff*.

4.5 Block Size Tuning

The number of reset points generated by the rolling hash function is determined by three parameters: (i) file length; (ii) randomness (entropy) of file content; and (iii) block size. For ease of comparison, a fuzzy hash should limit its signature length to a specific range. The original fuzzy hash algorithm achieves this goal by iteratively adjusting the initial block size and recomputing the hash value until the desirable signature length is obtained. However, this method is infeasible for files in transmission. On one hand, the file length may not be known before the computation, which means that Equation (1) cannot be used to compute the initial block size. On the other hand, using a fixed block size renders the signature length unpredictable and there is no opportunity for recalculation in a one-way process.

The stream fuzzy hash algorithm employs an adaptive mechanism to tune the block size during hashing. In the tuning process, the rolling hash value of each reset point (stored in $array_r$) is tested by a new block size, which is the current block size multiplied by a tuning factor k that

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

Figure 5. Example signature.

satisfies the following equation:

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

Obviously, if an appropriate reset point exists for a new block size, then it must be one of the surviving reset points. This guarantees that the final signature is independent of the timing of the tuning operation. After a new reset point is selected, the temporary hash result is refined by multiplying each strong hash value between the reset points.

The tuning factor k also determines whether or not a partial file can be compared. As discussed later in this chapter, a larger tuning factor k generates a longer signature.

A tuning operation is chosen carefully. Let S be the expected signature length. Then, a tuning operation is deemed to be necessary when the current reset point number is larger than $k \times S$. In certain cases, the file entropy may be extremely low (e.g., a string of repeated characters); this yields rolling hash values with less diversity. Tuning the block size based on these hash values causes a dramatic reduction in the number of reset points. To avoid this, every rolling hash value is tested with a new block size before tuning is performed. The tuning operation is aborted when the number of eligible reset points is less than the expected signature length S.

4.6 Signature Generation

After all the transferred data is input, the stream fuzzy hash algorithm initiates an inorder traversal of the interval tree to visit every context. Each strong hash value is mapped to a Base64 space with LS6B and the data range is in the format [*left offset* − *right offset*] to mark gaps. To enhance partial file matching, the stream fuzzy hash signature for block size b is $k \times b$, where k is the tuning factor. Repeated computations are not necessary because the signature of a block of size b is a precedent result of $k \times b$, which is stored in $backup_s$ by the tuning operation. The format of the stream fuzzy hash signature is $blocksize{:}hash_b{:}\ hash_{\frac{b}{k}}[range_{start} - range_{end}]$. Figure 5 shows an example stream fuzzy hash signature with an expected length of 64, block size of 3,072 and tuning factor of 4.

Since the stream fuzzy hash algorithm uses the same rolling hash function as ssdeep, the two signatures should have the same length for the same input. Some files may not yield an appropriate number of pieces for any block size. The ssdeep algorithm addresses this problem by combining the last few pieces of a message into a single piece. In contrast, the stream fuzzy hash algorithm keeps all the pieces separate and generates a longer signature to preserve more details of the input.

5. Signature Comparison

The stream fuzzy hash signature length depends on the block size and entropy of the input. Since the file is determinate, it can be assumed that a missing number of signature characters is linearly correlated with the missing length. Since s_1 is generated by the transferred file, n_i spaces are used for each gap in s_1, where n_i is given by:

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

Let s_1' be the filled signature and s_2 be the signature of the intact target file. The Levenshtein distance (LE) of s_1 and s_2 is refined using the following equation:

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

As in [14], the final match score in the range 0 to 100 is computed using the equation:

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

A higher match score indicates a greater probability that the source files have blocks of values in common and in the same order.

5.1 Partial File Matching

Two stream fuzzy hash signatures are comparable if and only if they have been generated using the same block size. As described above, block size tuning is driven by the input data and missing data postpones the tuning operation. A partial file may have a different block size from its original file. In fact, although the stream fuzzy hash signature with block size b has length b/k, it is still possible that a partial file has a different block size.

Assume that the reset points are evenly distributed over the entire file and consider a partial file with integrity rate m and tuning factor

k. Then, the probability that the partial file can be compared with the original file is given by:

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

Note that the even distribution is used just to simplify the problem; in practice, the distribution of reset points is strongly correlated with the dataset. For the desired stream fuzzy hash signature length S, a complete file with final block size b has two reset point numbers, one is L_b and the other is $L_{\frac{b}{k}}$, which is the precedent result of final tuning. L_b satisfies the following inequality:

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

The partial file signature is generated with block size b' and its reset point number $L_{b'}$ satisfy the following inequality:

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

The partial file is comparable when:

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

Based on Equation (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 Equations (12) and (13) yields:

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

Since $kS \le \frac{kS}{m}$, $m \in (0, 1]$, this inequility can be written as:

$$\frac{S}{mk} \le L_b \le kS \tag{15}$$

Since the reset points are evenly distributed, if L_b satisfies Equation (10), then the probability p that L_b also satisfies Equation (15) is given by:

$$p = \frac{kS - \frac{S}{mk}}{kS - S} = \frac{k^2m - 1}{k^2m - km} \tag{16}$$

A larger tuning factor k can be used to obtain a better comparison probability. However, this generates a longer signature that increases the storage requirements and comparison overhead. Therefore, it is necessary to trade-off comparison efficiency versus the ability to perform partial matching. For example, for a tuning factor $k = 3$ and integrity rate $m = 0.3$, the comparison probability is 0.94. Based on Equation (16), a partial file is always comparable when $m \geq \frac{1}{k}$. Equation (16) is also confirmed later in this chapter when the application of the stream fuzzy hash algorithm on a real dataset is evaluated.

It is important to note that the ability of the stream fuzzy hash algorithm to perform partial file matching not only enables a deep packet inspection application to identify incomplete captured files, but also allows hazardous transmissions to be stopped.

5.2 Comparing Massive Numbers of Files

Deep packet inspection and intrusion prevention systems maintain signature sets of valuable files and malicious software that run into millions of elements. For example, NIST's National Software Reference Library [15] maintains a large public database of known content that covers more than 50 million files. A naive solution for dealing with massive numbers of signatures is to compare all pairs by brute force, which is obviously impractical. Fortunately, the large-scale approximate matching problem has been well studied in connection with string similarity search [24]. The proposed approach adopts a classical method from the string similarity search domain to speed up comparisons.

The method involves indexing the stream fuzzy hash signatures using n-grams:

- **Index Creation:** An n-gram is a contiguous sequence of n characters from a sequence of text. The original design of the fuzzy hash algorithm requires that similar hashes must have a common 7-gram. Thus, each signature in the set is split into many 7-grams. Each 7-gram is treated as a key and the signature itself as a value. The key-value pair is then inserted into a hash table.

- **Signature Querying:** The signature to be queried is also split into several 7-grams. Every 7-gram in the hash table is examined to find candidate signatures. The 7-gram implementation was selected because the original fuzzy hash algorithm required similar hashes to have a common 7-gram. A threshold c is selected for querying based on a predefined similarity baseline. When a candidate signature shares more than c 7-grams with a query, then Equation (8) is applied to determine their similarity.

Table 2. Normalized TLSH distances versus ssdeep scores.

TLSH Distance	< 60	< 50	< 30	< 20	< 10	< 1
ssdeep Score	> 0	> 30	> 70	> 80	> 90	100

6. Evaluation

This section evaluates the correctness of the stream fuzzy hash algorithm and its hashing speed and signature length using the t5 corpus [19]. The t5 corpus contains 4,457 files and 1.8 GB of data. In particular, the stream fuzzy hash algorithm is compared against ssdeep v2.13 (sourceforge.net/projects/ssdeep), sdhash v3.4 (roussev.net/sdhash/sdhash.html) and TLSH v3.7 (github.com/trendmicro/tlsh), which were the latest versions available when the experiments were conducted. ssdeep is the *de facto* standard for malware analysis domain; it is currently the only similarity digest supported by VirusTotal (www.virustotal.com). sdhash is a widely-applied fuzzy hash implementation. ssdeep and sdhash are both supported by NIST's National Software Reference Library [15]. TLSH [16] is an open-source fuzzy matching library developed by Trend Micro.

6.1 Correctness

The first set of experiments sought to evaluate the ability of the stream fuzzy hash algorithm to identify similarities between files compared with other fuzzy hash algorithms. Note that the results can contain false positives (non-similar pairs identified as similar) and false negatives (similar pairs not identified as similar).

The ssdeep, sdhash and stream fuzzy hash schemes score the similarity between two files in a range from 0 to 100, where 0 corresponds to a mismatch and 100 is a perfect match (or near-perfect match); a lower score means a lower confidence level. However, TLSH uses a different scheme to score the similarity between two digests – a distance score of zero means that the files are identical (or nearly identical) and increasing score values above zero represent greater distances between the files.

In order to compare the four schemes using a common basis, the TLSH distance was normalized to a range from 0 to 100 based on the results obtained by Oliver et al. [16]. Table 2 shows the TLSH distances and ssdeep scores with the proximate false positive and recall rates considered to be the same.

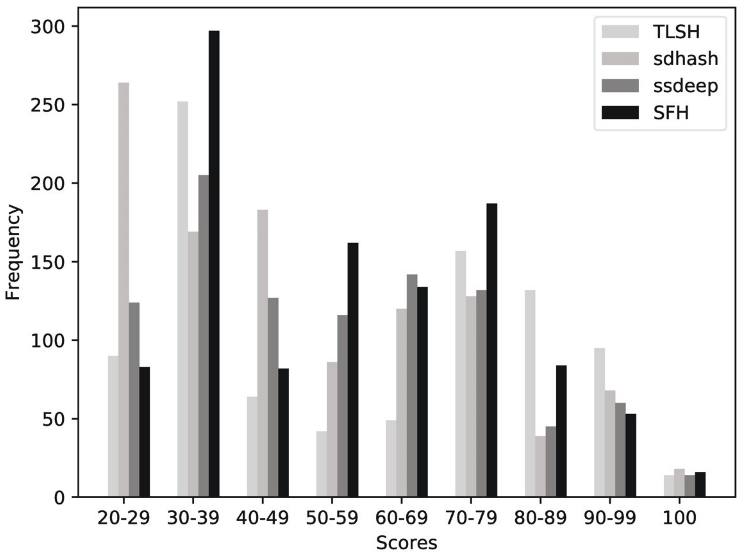


Figure 6. Distributions of the detected pairs in `t5`.

Figure 6 shows the distributions of the detected pairs obtained by the four hash algorithms.

Recall. Because there are $n(n-1)/2$ pairs in a set of n files, there are almost 10 million pairs in `t5` and it is not possible to determine all the pairs by hand. Therefore, an assumption was made that only (true and false) positives would be detected by an algorithm and, if a correlation was not discovered by an algorithm, then it did not exist. This is appropriate because the intent was to compare the performance of the stream fuzzy hash algorithm relative to the other algorithms instead of in absolute terms based on the ground truth.

Drawing on previous research [16, 19], a strict threshold value was set for each algorithm. Below the threshold, all positive results were ignored as the false positive rate rose to 10%. The four algorithms detected 387 similar pairs in total using the threshold values listed in Table 3. Note that the `TLSH` threshold is not a score, but a distance.

Precision. The 387 unique file pairs were reviewed manually. A total of 256 pairs were identified as true positives. The following definition was employed to determine a correct similar pair (true positive):

- TXT and HML files that use the same boilerplate or share more than 10% common content.

Table 3. Comparison of the precision and recall rates for `t5`.

	Threshold	True Positives	False Positives	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%
Total	–	256	131	–	–

- PDF, DOC, PPT and XLS files that are syntactically correlated beyond their formats.

- JPG and GIF files that have visual similarities.

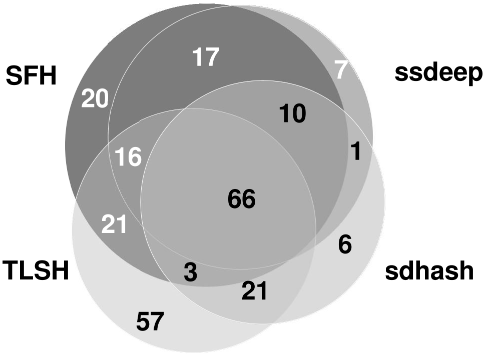


Figure 7. Intersections of the true positive sets for `t5`.

Note that the focus of the true positive definition is not on determining the percentage of similar pairs, but to compare the four algorithms on the same real-world dataset. Figure 7 shows the intersections of the true positive sets for the four algorithms. The overlaps vary because the thresholds are more rigorous than those in [19]. Note that, for readability, Figure 7 does not show all the intersections.

Table 3 compares the precision and recall rates of the four algorithms applied to `t5`. Note that the use of Tillich-Zémor hashing by the stream fuzzy algorithm does not decrease the precision and recall rates.

6.2 Hashing Speed for Sequential Inputs

This section evaluates the hashing speed of the stream fuzzy hash algorithm, which is a crucial property in deep packet inspection appli-

Table 4. Hashing speeds for 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

cations. MD5 was added to the four evaluated algorithms to provide readers with an intuitive understanding of the relative speeds of the algorithms.

The computer used in the experiments had a multicore Intel Xeon E5-2698 v3 CPU with a frequency of 2.30 GHz. All the algorithms were executed on one logic core with hyper-threading enabled. The operating system used was Linux RedHat 7.2 (kernel 3.10). The `t5` corpus was used as the input in the speed tests. The source code of all the algorithms was compiled with `gcc -O2` and configured as the default option. The MD5 results were generated by OpenSSL v1.0.0. Every algorithm processed the files sequentially and the chunk size was 4,096 bytes.

Table 4 shows the hashing speeds of the five algorithms. The performance gaps between the cryptographic hash algorithm (MD5) and the four fuzzy hash algorithms are evident. In the case of the stream fuzzy hash algorithm, the gap is mainly due to the fact that a block hash function was not used; instead, each bit was hashed individually. By querying the Galois multiplication table, it would have been possible to hash eight bits per call, but the invocation cost would still be more than a block hash. In the case of MD5 (and SHA1), an input message was broken up into 512-bit blocks, which `ssdeep` and the stream fuzzy hash algorithm processed byte-by-byte.

The `ssdeep` algorithm has to recalculate an n-byte input $O(\log n)$ times to find the proper block size and requires $O(n)$ time for each computation, making the total execution time $O(n \log n)$. For the stream fuzzy hash algorithm with block size tuning, no recalculation is needed after the block size is adjusted, so the total execution time is $O(n)$. However, the complexity of the Tillich-Zémor hash weakens this advantage.

6.3 Hashing Speed for Unordered Inputs

The stream fuzzy hash algorithm is designed to hash files in transmission. Therefore, the unordered input of a multi-thread download was simulated. The chunk size was 1,460 bytes in order to simulate a generic TCP payload. Sequential inputs are preconditions for all the other algo-

Table 5. Hashing speeds for `t5` multi-thread downloading.

	4 Threads	**8 Threads**	**16 Threads**	**Random Order**
Speed (MB/s)	67.8	67.8	67.6	61.3
Space (KB)	2.39	3.40	4.90	310.20

rithms evaluated in this research; therefore, they could not be compared with the stream fuzzy hash algorithm in this set of experiments.

Table 5 shows the hashing speeds of the stream fuzzy hash algorithm for different numbers of concurrent fragments. No significant slowdown was observed even when the input order was completely random. The memory consumption in the case of sixteen concurrent fragments was 4.90 KB, which is practical for deep packet inspection applications.

6.4 Comparison of Incomplete Files

As discussed earlier, files captured from network traffic may be incomplete for a number of reasons. It has also been shown that, if the product of the integrity rate m and the tuning factor k is greater than one, then the partial file has a same block size as the original file. Experiments were conducted to validate this theoretical result on real data.

In the experiments, for each file in `t5`, only a portion of m (0 to 0.5) from the first byte of the file was provided as input to the stream fuzzy hash algorithm. If the block size of an incomplete file was the same as that of the complete file, then, by design, the two files were deemed to be comparable. The choice of the threshold score may be left to the user.

Figure 8 shows the incomplete file comparison probabilities obtained for various tuning factors k. Note that the results mostly fit with the predictions made by Equation (9).

6.5 Stream Fuzzy Hash Algorithm Deployment

This section discusses a case study involving the deployment of the stream fuzzy hash algorithm to provide an intuitive understanding of the practicalities involved in handling real network traffic. In the case study, the stream fuzzy hash algorithm was integrated as a plug-in in a carrier-grade deep packet inspection system to identify malicious Android app installation packages in network traffic. The stream fuzzy hash algorithm proved to be very flexible because it does not require buffering

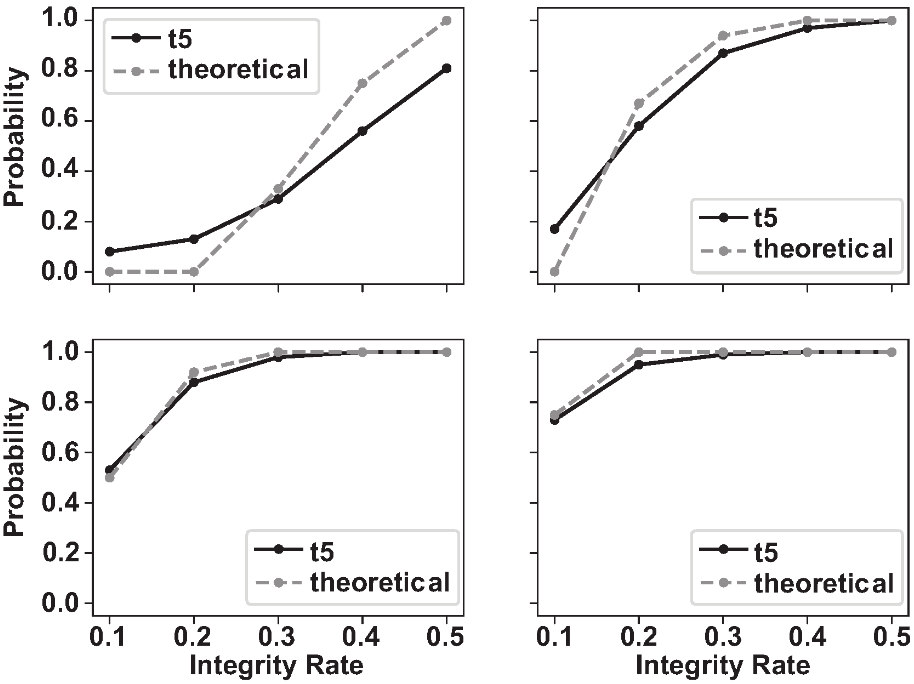


Figure 8. Incomplete file comparison probabilities for various tuning factors.

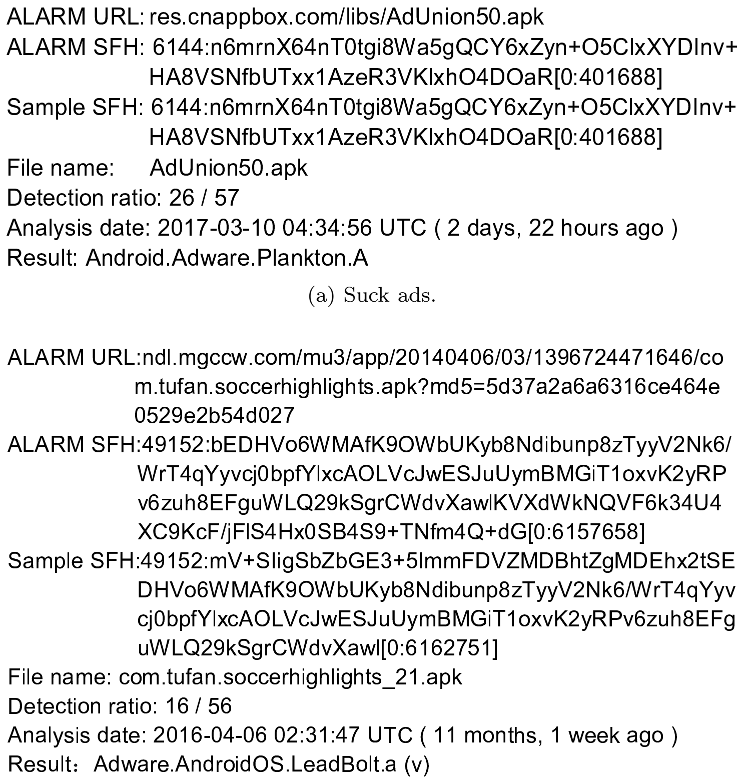
and segment rearrangement; as a result, it was readily integrated with the deep packet inspection system.

The following steps were involved in detecting malicious Android app installation packages:

- Malicious Android app samples were downloaded from VirusShare (`virusshare.com`). In all, 35,397 files with a total size of 52.82 GB were downloaded and a stream fuzzy hash algorithm signature was computed for each file.

- To enhance comparison efficiency, the 35,397 signatures were indexed as described in Section 5.2.

- The deep packet inspection system was deployed at an Internet service provider. It processed traffic at about 10 Gbps. The stream fuzzy hash algorithm plug-in identified app package transmissions based on their URLs. The plug-in provided the packets of the HTTP session as input to the stream fuzzy hash algorithm. Thus, a data structure containing the stream fuzzy hash algorithm signatures and URLs was created.



```
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 app.

Figure 9. Malicious app packages detected by the stream fuzzy hash algorithm.

- The signature of the app in the previously-constructed index was queried to measure the similarity. An alarm was raised when the distance as computed by Equation (7) was less than a fixed threshold.

In the case study, the deep packet inspection system processed 184,688 Android APK downloads and raised fourteen alarms associated with ten apps. To verify the results, the suspected app packages were retrieved using the recorded URLs and uploaded to VirusTotal for further examination.

Four app packages were downloaded successfully and two of them were identified as malicious (true positives). Figure 9 shows the two malicious app packages. The Football Highlights app was detected based on similarity, a capability that is not provided by conventional detection methods. Visual checks of the two false positive app binaries revealed that their file contents were quite similar to malicious samples. They shared 95% mutual content and presumably used the same development components. Thus, the false positive results are due to the concepts underlying approximate matching as opposed to a flaw in the proposed stream fuzzy hash algorithm.

7. Related Work

Fuzzy hash algorithms have evolved over the years. Kornblum [14] developed the `ssdeep` open-source fuzzy hash algorithm, which has been widely used to find similar files. Chen and Wang [6] and Breitinger and Baier [2] have proposed approaches for improving the performance of fuzzy hashing. However, these researchers and others have not considered the challenges involved in applying fuzzy hashing to network traffic.

The `sdhash` algorithm [18] has also been widely used for similar file detection. It attempts to find the features in each neighborhood that have the lowest empirical probability of being encountered by chance. Each of the selected features is hashed and placed in a Bloom filter. When a Bloom filter reaches its capacity, a new filter is created until all the features are accommodated. Thus, an `sdhash` similarity digest comprises a sequence of Bloom filters. The `sdhash` digest length is about 2 to 3% of the input length, which is different from the bounded digest of the fuzzy hash algorithm (64 to 128 bytes). Because it retains more details of the original file, `sdhash` is better at embedded object detection than `ssdeep` [19]. However, retaining these details increases the storage and comparson overhead. For example, `sdhash` generated digests totaling 101 GB for the 50 million files in the National Software Reference Library [15]; in contrast, `ssdeep` generated only 1.2 GB of digests for the same set of files.

MinHash [4] and SimHash [5] have been widely adopted in industry to identify potential duplicated text files. They belong to the family of locality-sensitive hash (LSH) algorithms. Shrivastava and Li [20] claim that MinHash outperforms SimHash on binary data, but it is curious that locality-sensitive hash algorithms are rarely employed in digital forensic applications. Harichandran et al. [11] note that a locality-sensitive hash algorithm attempts to map similar objects to the same bucket whereas approximate matching yields similarity digests that can be compared to a desired threshold. Several other open-source algorithms and tools, such as Nilsimsa [25], `TLSH` [16], MRSH-v2 [3], have also been developed to detect similar files.

8. Stream Fuzzy Hash Algorithm Limitations

Although the stream fuzzy hash algorithm has important security applications, it is by no means cryptographically secure. Additionally, the stream fuzzy hash algorithm cannot handle files that have been compressed or encrypted.

As a hash function, the stream fuzzy hash algorithm has two principal limitations:

- **Collisions:** Like the original fuzzy hash algorithm, the stream fuzzy hash algorithm maps a file chunk to a six-bit value. Therefore, the possibility always exists that two distinct chunks will map to the same hash. Moreover, two files that have identical stream fuzzy hash signatures could still be different files. Kornblum [14] has shown that the probability of a failure to detect a change is 2^{-12} to 2^{-6}. In the case of two completely-random files with signatures of length S, the probability of an exact match is $\left(2^{-6}\right)^S$. Of course, the expected signature length can be increased to reduce collisions, but this increases the time and space requirements.

- **Signature Comparison:** Meaningful signature comparisons can only be performed on files with the same block size. This is not a problem for different files that match approximately, but it is a useful feature because different block sizes means that two files are quite different in terms of size as well as content. In the case of partial file matching, as discussed in Section 5.1, the cut-off of $1/k$ of a file is always compared, where k is the tuning factor. Below this level, the block sizes are too different to make meaningful comparisons. Embedded file detection is similar because a small embedded object can be considered to be a portion of the larger object.

9. Conclusions

The stream fuzzy hash algorithm is specifically designed to hash files and file fragments captured from network traffic in real time. The algorithm leverages the context-triggered piecewise hashing concept, employs the Tillich-Zémor hash as a strong hash function and uses an interval tree to index the computed segment contexts, rendering it very effective at handling unordered and incomplete inputs. With block size tuning, the stream fuzzy hash algorithm can hash a data stream using one-way processing. Additionally, compared with the `ssdeep` algorithm, the stream fuzzy hash algorithm reduces the computational complexity from $O\left(n \log n\right)$ to $O\left(n\right)$.

Experimental results demonstrate that the stream fuzzy hash algorithm has a hashing speed of 68 MB/s per CPU core and consumes just 5 KB of memory per file. Moreover, compared with the other fuzzy hash algorithms, the precision and recall of the stream fuzzy hash algorithm are not compromised when processing unordered and incomplete inputs.

The integration of the stream fuzzy hash algorithm in a carrier-grade deep packet inspection system to identify malware in network traffic demonstrates the applications potential of the algorithm. A deep packet inspection system incorporating the stream fuzzy hash algorithm can identify valuable files (e.g., containing intellectual property) in egress traffic and malicious software in ingress traffic. Another significant benefit is the ability to perform partial file matching in the context of deep packet inspection – this makes it possible to identify files before transmission completes and to stop attacks before they can be realized.

Interested readers may access `github.com/mesasec/sfh` for the source code related to this project.

Acknowledgements

The authors wish to thank Vassil Roussev for his assistance with the evaluation conducted in this research. This research was supported in part by the National Key R&D Program of China under Grant No. 2016YFB0801304.

References

[1] F. Breitinger and I. Baggili, File detection on network traffic using approximate matching, *Journal of Digital Forensics, Security and Law*, vol. 9(2), pp. 23–35, 2014.

[2] F. Breitinger and H. Baier, Performance issues about context-triggered piecewise hashing, *Proceedings of the International Conference on Digital Forensics and Cyber Crime*, pp. 141–155, 2011.

[3] F. Breitinger and H. Baier, Similarity preserving hashing: Eligible properties and a new algorithm MRSH-v2, *Proceedings of the International Conference on Digital Forensics and Cyber Crime*, pp. 167–182, 2012.

[4] A. Broder, On the resemblance and containment of documents, *Proceedings of the Conference on Compression and Complexity of Sequences*, pp. 21–29, 1997.

[5] M. Charikar, Similarity estimation techniques from rounding algorithms, *Proceedings of the Thirty-Fourth Annual ACM Symposium on the Theory of Computing*, pp. 380–388, 2002.

[6] L. Chen and G. Wang, An efficient piecewise hashing method for computer forensics, *Proceedings of the First International Workshop on Knowledge Discovery and Data Mining*, pp. 635–638, 2008.

[7] T. Cormen, C. Leiserson, R. Rivest and C. Stein, *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts, 2009.

[8] Y. Elovici, A. Shabtai, R. Moskovitch, G. Tahan and C. Glezer, Applying machine learning techniques for detection of malicious code in network traffic, *Proceedings of the Annual Conference on Artificial Intelligence*, pp. 44–50, 2007.

[9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee, Hypertext Transfer Protocol – HTTP/1.1, RFC 2616, 1999.

[10] M. Grassl, I. Ilic, S. Magliveras and R. Steinwandt, Cryptanalysis of the Tillich-Zémor hash function, *Journal of Cryptology*, vol. 24(1), pp. 148–156, 2011.

[11] V. Harichandran, F. Breitinger and I. Baggili, Bytewise approximate matching: The good, the bad and the unknown, *Journal of Digital Forensics, Security and Law*, vol. 11(2), pp. 59–77, 2016.

[12] K. Joju and P. Lilly, Pre-image of Tillich-Zémor hash function with new generators, *Applied Mathematical Sciences*, vol. 7(85), pp. 4237–4248, 2013.

[13] K. Joju and P. Lilly, Improved form of Tillich-Zémor hash function, *International Journal of Theoretical Physics and Cryptography*, vol. 6, pp. 24–29, 2014.

[14] J. Kornblum, Identifying almost identical files using context-triggered piecewise hashing, *Digital Investigation*, vol. 3(S), pp. S91–S97, 2006.

[15] National Institute of Standards and Technology, National Software Reference Library (NSRL), Gaithersburg, Maryland (`www.nist.gov/software-quality-group/national-software-reference-library-nsrl`), 2018.

[16] J. Oliver, C. Cheng and Y. Chen, TLSH – A locality sensitive hash, *Proceedings of the Fourth Cybercrime and Trustworthy Computing Workshop*, pp. 7–13, 2013.

[17] C. Petit and J. Quisquater, Pre-images for the Tillich-Zémor hash function, *Proceedings of the International Workshop on Selected Areas in Cryptography*, pp. 282–301, 2010.

[18] V. Roussev, Data fingerprinting with similarity digests, in *Advances in Digital Forensics VI*, K. Chow and S. Shenoi (Eds.), Springer, Heidelberg, Germany, pp. 207–226, 2010.

[19] V. Roussev, An evaluation of forensic similarity hashes, *Digital Investigation*, vol. 8(S), pp. S34–S41, 2011.

[20] A. Shrivastava and P. Li, In defense of MinHash over SimHash, *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*, pp. 886–894, 2014.

[21] X. Shu and D. Yao, Data leak detection as a service, *Proceedings of the International Conference on Security and Privacy in Communications Systems*, pp. 222–240, 2012.

[22] J. Tillich and G. Zémor, Hashing with SL_2, *Proceedings of the International Cryptology Conference*, pp. 40–49, 1994.

[23] A. Tridgell, `spamsum` (`github.com/tridge/junkcode/tree/master/spamsum`), 2002.

[24] S. Wandelt, J. Wang, S. Gerdjikov, S. Mishra, P. Mitankin, M. Patil, E. Siragusa, A. Tiskin, W. Wang, J. Wang and U. Lesser, State-of-the-art in string similarity search and join, *ACM SIGMOD Record*, vol. 43(1), pp. 64–76, 2014.

[25] Wikipedia, Nilsimsa Hash (`en.wikipedia.org/wiki/Nilsimsa_Hash`), 2018.

[26] C. Winter, M. Schneider and Y. Yannikos, F2S2: Fast forensic similarity search through indexing piecewise hash signatures, *Digital Investigation*, vol. 10(4), pp. 361–371, 2013.