Chapter 5

EVALUATION OF NETWORK TRAFFIC ANALYSIS USING APPROXIMATE MATCHING ALGORITHMS

Thomas Göbel, Frieder Uhlig and Harald Baier

Abstract Approximate matching has become indispensable in digital forensics as practitioners often have to search for relevant files in massive digital corpora. The research community has developed a variety of approximate matching algorithms. However, not only data at rest, but also data in motion can benefit from approximate matching. Examining network traffic flows in modern networks, firewalls and data loss prevention systems are key to preventing security compromises.

This chapter discusses the current state of research, use cases, validations and optimizations related to applications of approximate matching algorithms to network traffic analysis. For the first time, the efficacy of prominent approximate matching algorithms at detecting files in network packet payloads is evaluated, and the best candidates, namely `TLSH`, `ssdeep`, `mrsh-net` and `mrsh-cf`, are adapted to this task. The individual algorithms are compared, strengths and weaknesses highlighted, and detection rates evaluated in gigabit-range, real-world scenarios. The results are very promising, including a detection rate of 97% while maintaining a throughput of 4 Gbps when processing a large forensic file corpus. An additional contribution is the public sharing of optimized prototypes of the most promising algorithms.

Keywords: Network traffic analysis, approximate matching, similarity hashing

1. Introduction

Data loss prevention systems protect enterprises from intellectual property and sensitive data theft. They have become almost indispensable as legal requirements like the General Data Protection Regulation (GDPR) in the European Union levy high fines when consumer data is accessed by unauthorized parties.

In October 2020, the German software company Software AG was hit with a ransom demand of 23 million euros in exchange for the decryption password and the promise not to disclose 1 TB of internal company documents and customer data [19]. Software AG products are used by 70% of Fortune 1,000 companies. In addition to the ransom, recovery costs and loss of reputation incurred by Software AG, the damage extended to the enterprise environments of hundreds of its customers.

In June 2016, Russian hackers using the Guccifer 2.0 pseudonym stole tens of thousands of email and documents, including 8,000 attachments from the U.S. Democratic National Committee. The public release of the stolen documents on WikiLeaks constituted an attempt to actively influence the 2016 U.S. presidential election [16].

The two incidents demonstrate the costs of data protection failures. Also, they underscore the importance of checking data flows at transition points in networks. Whether this is accomplished using intrusion detection/prevention systems or data loss prevention systems, transparency at network transition points is vital to protecting sensitive data. Enterprise networks host a variety of platforms for data and file exchange. Emerging technologies, such as cloud platforms and workstream collaboration platforms (e.g., Slack and Mattermost), simplify data exchange in enterprises. However, they increase the risk of sensitive data leaving the premises of controlled networks and falling into the wrong hands.

To reduce the risk of data leakage, data loss prevention solutions must be implemented in networks, at network endpoints and in the cloud. A single data loss protection solution is ineffective; rather, a multi-tiered approach is needed to protect an enterprise infrastructure from unauthorized data access. Indeed, an orchestrated solution involving endpoints, storage and networks is advised [1]. Such a solution filters data at rest, data in use and data in motion to prevent the data from leaving the virtual premises of an enterprise. Since most data loss prevention systems are closed-source solutions, without internal knowledge of these products, it is difficult to judge their protection scopes and effectiveness.

Most commercial data loss protection solutions fall into one or combinations of three categories:

- **Endpoint Protection:** These solutions involve desktop and/or server agents that enable security teams to monitor data at endpoints. Predefined rules and blacklists prevent users from copying data (in use) to removable devices or transferring data to unauthorized web destinations.

- **Storage Protection:** These solutions focus on securing data at rest. Several cloud-based solutions fall in this category. Many of them protect intellectual property on Amazon AWS or Google Cloud platforms. They achieve their goals in part through access restrictions and blacklisting.

- **Network Protection:** These solutions protect data in motion, specifically at the ingress and egress points of enterprise networks. Many solutions prevent leakage by analyzing network packet metadata. Although network packet content analysis is hindered by encryption (approximately 80% of traffic is currently encrypted and the trend is towards total encryption), it is feasible for enterprises to place these solutions in-line with their network appliances. Products such as Sophos XG Firewall, Symantec Data Loss Prevention and Fortinet's Fortigate DLP integrate decryption functionality of TLS 1.3 communications. These are the network points at which the research discussed in this chapter is directly applicable. Decryption is necessary because encrypted packets are uniformly distributed and lack the recognizable features needed to leverage approximate matching.

This research focuses on preventing data loss in network environments. Approximate matching can be used to identify files in unencrypted network packet payloads. The goal is to transform theoretical concepts into reality and craft feasible solutions for identifying files with high accuracy while maintaining low failure rates at gigabit throughputs. Note that the approach does not involve prior inspection of unencrypted payloads to determine relationships between packets via string matching. Deep packet inspection methods can detect the transmission of sensitive content hidden in files. Since the majority of data loss occurs not via complete files but through portions of files being reformatted within another file or context (e.g., the Open Document Format compresses the actual contents), deep packet inspection aids data loss prevention via pattern matching and may find compressed portions of mixed files. However, these measures are not employed in this work.

This research demonstrates how file recognition in network traffic can be improved via approximate matching. The performance of prominent approximate matching algorithms is evaluated. First, file detection rates in an idealized setting are assessed without added noise from live traffic. The detection rates and maximum throughputs of the algorithms are measured. The best-performing algorithm is subsequently applied to live traffic and its performance is examined in detail. All the algorithms were adapted for packet filtering in that they were connected to a network interface and invoked whenever a packet was picked up by the interface.

Reasons for the variations in the true positive and false positive detection rates between the approximate matching algorithms are discussed. In fact, this work is the first to use prominent approximate matching algorithms to match files in real-world network traffic. Additionally, the two best approximate matching algorithms are adapted for efficient and effective real-time filtering of network traffic.

2. Foundations and Related Work

Approximate matching was first employed in digital forensics in the mid-2000s. The U.S. National Institute of Standards and Technology (NIST) [8] defines approximate matching as "a promising technology designed to identify similarities between two digital artifacts ... to find objects that resemble each other or to find objects that are contained in another object." Approximate matching algorithms achieve this goal using three approaches [23]:

- **Bytewise Matching:** This type of matching, referred to as fuzzy hashing or similarity hashing, operates at the byte level and only takes byte sequences as inputs.
- **Syntactic Matching:** This type of matching takes bytes as inputs but also relies on internal structure information of the subjects that are intended to be matched (e.g., header information in packets may be ignored).
- **Semantic Matching:** This type of matching, which focuses on content-visual differences, resembles human recognition. For example, JPG and PNG images may have similar content (i.e., pictures), but their filetypes and byte streams are different.

Four types of approximate matching approaches are relevant to this work [25]:

- **Context-Triggered Piecewise Hashing (CTPH):** This approach locates content markers (contexts) in binary data. It computes the hash of each document fragment delimited by contexts and stores the resulting sequence of hashes. In 2006, Kornblum [24] created `ssdeep`, one of the first algorithms that computed context-triggered piecewise signatures. The algorithm produces a match score from 0 to 100 that is interpreted as a weighted measure of file similarity, where a higher score implies greater similarity.
- **Block-Based Hashing (BBH):** This approach generates and stores cryptographic hashes of fixed-size blocks (e.g., 512 bytes).

The block-level hashes of two inputs are compared by counting the number of common blocks and computing a measure of similarity. An example implementation is `dcfldd` [22], which divides input data into blocks and computes their cryptographic hash values. The approach is computationally efficient but it is highly unstable because adding or deleting even a single byte at the beginning of a file changes all the block hashes.

- **Statistically-Improbable Features (SIF):** This approach identifies a set of features in each examined object and compares the features; a feature in this context is a sequence of consecutive bytes selected according to some criteria from the file in which the object is stored. Roussev [30] used entropy to find statistically-improbable features. His `sdhash` algorithm [31] generates a score between 0 and 100 to express the confidence that two data objects have commonalities.

- **Block-Based Rebuilding (BBR):** This approach uses external auxiliary data corresponding to randomly-, uniformly- or fixed-selected blocks (e.g., binary blocks) of a file, to reconstruct the file. It compares the bytes in the original file with those in the selected blocks and computes the differences between them using the Hamming distance or another metric. The differences are used to find similar data objects. Two well-known block-based rebuilding algorithms are `bbHash` [5] and `SimHash` [34].

- **Locality-Sensitive Hashing (LSH):** This approach from the data mining field is technically not an approximate matching approach. It is considered because it is often employed in a similar context as approximate matching.

  Locality-sensitive hashing is a general mechanism for nearest neighbor search and data clustering whose performance strongly relies on the hashing method. An example is the `TLSH` algorithm [26], which processes an input byte sequence using a sliding window to populate an array of bucket counts and determines the quartile points of the bucket counts. A fixed-length digest is constructed, which comprises a header with the quartile points, input length and checksum, and a body comprising a sequence of bit pairs that depend on each bucket's value in relation to the quartile points. The distance between two digest headers is determined by the differences in file lengths and quartile ratios. The bodies are compared using their approximate Hamming distance. The similarity score is computed based on the distances between the two headers and the two bodies.

2.1 Current State of Approximate Matching

In 2014, Breitinger and Baggili [4] proposed an approximate matching algorithm for filtering relevant files in network packets. The algorithm divides input files into 1,460 byte sequences, which simulates packets with maximum transmission units of 1,500 bytes less 20 bytes each for the IP and TCP headers. It achieved false positive rates between 10^{-4} and 10^{-5} for throughputs exceeding 650 Mbps.

The algorithm of Breitinger and Baggili was evaluated using simulated network traffic [4]. In contrast, this research has tested algorithms using real network traffic to understand their real-world applicability; additionally, this work has conducted experiments with variable packet sizes. Another key issue is that the throughput of 650 Mbps reported by Breitinger and Baggili was achieved without parallelization. In fact, they noted that parallelization could increase the speed significantly because hashing packets and performing comparisons with a Bloom filter can be done in an unsynchronized manner [10].

Since 2014, there has been little progress in advancing approximate matching in the network context. This is largely due to the encryption of network traffic payloads, which prevents file detection.

However, significant improvements have been made to approximate matching algorithms. Cuckoo filters, which are more practical than Bloom filters [17], have been adapted to approximate matching in digital forensic applications [21]. Researchers have also proposed improvements to Bloom and cuckoo filters such as XOR filters that are faster and smaller than Bloom filters [20], Morton filters that are faster, compressed cuckoo filters [11] and hierarchical Bloom filter trees that are improved Bloom filter trees [28]. However, it remains to be seen if the application of these improved filters to approximate matching will provide better file recognition performance.

2.2 Approximate Matching Algorithms

Filters are constantly being improved by the research community. However, approximate matching algorithms that employ filters and underlying matching methods advance more slowly. The following approximate matching algorithms are used frequently:

- **MRSH:** Algorithms in the multi-resolution similarity hash (`MRSH`) family are based on `ssdeep` and, thus, employ context-triggered piecewise hashing. Roussev et al. [33] created the original `mrshash` algorithm, which was subsequently improved to `MRSH-v2`, a faster version [6]. The algorithm also provides a fragment detection mode and the ability to compute file similarity.

Most relevant to this research are two variants of `MRSH-v2`. The `mrsh-net` algorithm [4] is a special version of `MRSH-v2` created for network packet filtering; it is also the first approximate matching algorithm to be evaluated in a network context. The `mrsh-cf` algorithm [21] is a special version of `mrsh-net` that replaces the Bloom filter with a cuckoo filter, providing runtime improvements and much better false positive rates.

The latest variant of the `MRSH` family, `mrsh-hbft`, uses hierarchical Bloom filter trees instead of conventional Bloom filters. Lillis et al. [28] have demonstrated that the hierarchical Bloom filter trees used in `mrsh-hbft` improve on the original `MRSH-v2` algorithm, which uses a standard Bloom filter. However, as discussed later, the `mrsh-cf` algorithm is still unmatched in terms of speed and accuracy, which is especially important in network applications.

- **mvHash-B:** The `mvHash-B` algorithm [3] employs block-based and similarity-preserving hashing, but also relies on majority voting and Bloom filters. It is highly efficient with very low runtime complexity and small digest size. The algorithm exhibits weaknesses to active adversaries [13], but these issues are now addressed [13]. A caveat is that the algorithm must be adjusted for every filetype. The original algorithm only applied to JPG and DOC files.

- **FbHash:** The `FbHash` algorithm [12] builds on the `mvHash-B` algorithm. However, its source code is not published, so the algorithm cannot be adapted to network traffic analysis. Moreover, the algorithm is limited to a small set of filetypes.

- **sdhash:** Roussev [31] developed the `sdhash` algorithm four years after the release of `ssdeep`. The `sdhash` algorithm uses hashed similarity digests as a means of comparison. A similarity digest contains statistically-improbable features in a file. The `sdhash` algorithm has been thoroughly evaluated against its predecessor `ssdeep` [7]. It detects correlations with finer granularity than `ssdeep`, which makes it a viable candidate [27]. However, a runtime comparison by Breitinger et al. [9] has demonstrated that `MRSH-v2` outperforms `sdhash` by a factor of eight. `MRSH-v2` is chosen over `sdhash` in this research because of its superior runtime performance and similar ability to correlate small fragments.

- **TLSH:** `TLSH` developed by Oliver et al. [26] is an adaptation of the Nilsimsa hash [15] used for spam detection. As mentioned above, `TLSH` is not an approximate matching algorithm because it is based on locality-sensitive hashing that does not conform to the approximate matching definition [8]. `TLSH` uses the Hamming distance between Bloom filters to compare hashes and provides similarity scores ranging from 0 to 1,000. `TLSH` has worse performance than `ssdeep` and `sdhash`, with `sdhash` consistently recognizing files at the smallest granularity. `TLSH` is extremely robust to random manipulations such as insertions and deletions in a file. However, it produces high false positive rates and is slightly slower than `mrsh-net`.

- **ssdeep:** The `ssdeep` algorithm is commonly used in digital forensics. It is implemented on several platforms [35] and is the *de facto* standard in some cyber security areas [29]. Applications such as VirusTotal are based on `ssdeep`. The algorithm generates a signature file that depends on the actual file content. It is used to compare two signature files or a signature file against a data file; the results are the same in both cases and the use of one or other method depends on the available data. Although `ssdeep` is a key achievement in similarity detection and is still relatively up-to-date, some limitations have been identified recently, for which certain enhancements and alternative theoretical approaches have been suggested [2].

Table 1 presents a detailed evaluation of approximate matching algorithms for network traffic analysis. Note that the `mrsh-cf` algorithm is the 2020 version with a newer cuckoo filter.

3. Controlled Study

As discussed in Section 2.2, based on their reliability and performance, three algorithms are best suited to network traffic analysis: (i) `ssdeep`, (ii) `TLSH` and (iii) `MRSH` (`mrsh-net` and `mrsh-cf` variants). The performance, speed and applicability of the algorithms were first evaluated with respect to data at rest. Next, the three algorithms were evaluated on their ability to deal with live network packets. In the case of the `MRSH` family, the `mrsh-net` variant was evaluated as the first algorithm capable of filtering network traffic whereas the `mrsh-cf` was evaluated as a faster and more reliable version of `mrsh-net`.

The algorithms were evaluated using the well-known *t5-corpus*, which contains 4,457 files with a total size of 1.8 GB [32]. The average file size is almost 420 KiB. Table 2 shows the composition of the *t5-corpus*. Note

Table 1. Evaluations of approximate matching algorithms.

	mrsh-cf	**mrsh-net**	**mvHash-B**	**sdhash**	**TLSH**	**ssdeep**	**FbHash**	**mrsh-hbft**
Speed	37% runtime improvement over `mrsh-net`	Faster than `mrsh-v2`	Lowest runtime complexity	No current benchmarks	No current benchmarks	Twice as fast as `TLSH` (2017 version)	No current benchmarks	No current benchmarks
Fragment Detection	Small	Small	NA	5%	Small	20 to 50% of original	1%	NA
Remarks	Fastest `MRSH` algorithm	Special version of `mrsh-v2` for network traffic	Limited to a few file types	`MRSH` algorithms are more effective, but `sdhash` is slightly more accurate	Used for file similarity instead of file identification	Cannot hash files over 2 GB	Unpublished code and limited to DOC file types	Preferable in cases with memory limitations
Basis	Context-triggered piecewise hashing, Cuckoo filter	Context-triggered piecewise hashing, Bloom filter	Block-based hashing	Statistically-improbable features	Locality-sensitive hashing	Context-triggered piecewise hashing	Term frequency inverse document frequency, similar to `mvHash-B`	Context-triggered piecewise hashing, Hierarchical Bloom filter
Latest Version	2020	2015	2012	2013	2020	2017	2018	2018

Table 2. *t5-corpus* composition.

JPG	GIF	DOC	XLS	PPT	PDF	TXT	HTML
362	67	533	250	368	1,073	711	1,093

that the *t5-corpus* is a subset of the *GovDocs* corpus, which was created by crawling U.S. Government websites [18]. Due to the data collection process, it is assumed that the corpus has multiple related files.

Table 3. Time requirements for filter generation and application.

	mrsh-cf	mrsh-net	ssdeep	TLSH	mrsh-hbft
Filter Generation	12.51 s	32.90 s	14.90 s	17.18 s	274 s
All vs. All	12.94 s	67.84 s	27.37 s	78.29 s	300 s

3.1 All vs. All Evaluation

The All vs. All evaluation sets algorithm performance baselines using data at rest. Each algorithm first generated a filter of the *t5-corpus*. Next, each algorithm with its filter was given the entire *t5-corpus* (1.8 GB) to process. Table 3 shows the time requirements for the algorithms to generate the filter and apply it to every file in the corpus.

Speed is a key indicator of the suitability of an algorithm for detecting files in network traffic. This is why the performance of each algorithm was evaluated when matching the *t5-corpus* with itself. Three of the algorithms are capable of performing small fragment detection, which is important for file matching in network packets. `ssdeep` is limited in this regard, but it has a well-maintained code base [35] and is claimed to be twice as fast as `TLSH`, which is why it is considered in the evaluation. Variants of `ssdeep` [25] have addressed the problem related to small fragment detection, so the `ssdeep` the algorithm can no longer be excluded on this basis. For reference, the `mrsh-hbft` algorithm is very slow compared with the other algorithms; it would be much too slow for network traffic analysis.

3.2 Evaluation Methodology

Since it is difficult to evaluate algorithm performance in a real network with high precision, a controlled evaluation environment was created. The environment incorporated a network interface that listened to traffic coming in from a virtual server and destined to a virtual client. Only unencrypted TCP packets were transmitted because they led to the HTTP and FTP network packets that were dominant in file transmission. As mentioned earlier, dealing with encrypted traffic is outside the scope of this research.

In the experiments, each packet was stripped off its header and its payload was compared against the target hash. Since none of the selected algorithms had a built-in live filtering function for network traffic, this feature was added to each algorithm. The loopback interface was used to pass unencrypted TCP packets to each algorithm, which reported whether or not the payloads were recognized.

Most approximate matching algorithms only confirm how many chunks of a target hash are found in input data. Therefore, if the hashes of multiple files are compared against a single file, an algorithm can only tell if a file is present, not which file from among all the others. When filtering packet payloads, this means that an algorithm can tell if a packet contains a hash, but not which file is contained when multiple files are identified. This means that packet payloads matched by an algorithm have to be verified later.

When a packet payload is matched as belonging to a file, the payload is concatenated with all the other matched payloads and fed back to the algorithm after passing through the network filtering component. Since the order in which the randomly-chosen files were sent and their sizes were recorded, a second comparison of the packet payloads against the corpus of all possible files was performed to reveal whether each algorithm correctly identified the files transferred over the network interface. The only alternative would be to modify each algorithm to concretely identify one file out of many, but this could degrade the performance of the algorithm in a manner that was not intended by its developers.

Each algorithm essentially has a filter of the entire *t5-corpus* that it compares against each packet and, thus, identifies the files that were transmitted. A `curl` job on the client side pulled random files with adjustable throughputs over the loopback interface. The files were recorded along with whether or not they were recognized by the algorithm. The bandwidth of the algorithm was limited by its ability to recognize a payload before the next payload arrived. If the algorithm was still handling a payload when a new payload arrived on the network interface, then the new payload was lost. Thus, the algorithm could not recognize all the files that passed through. As the throughput increased, the file detection rate decreased.

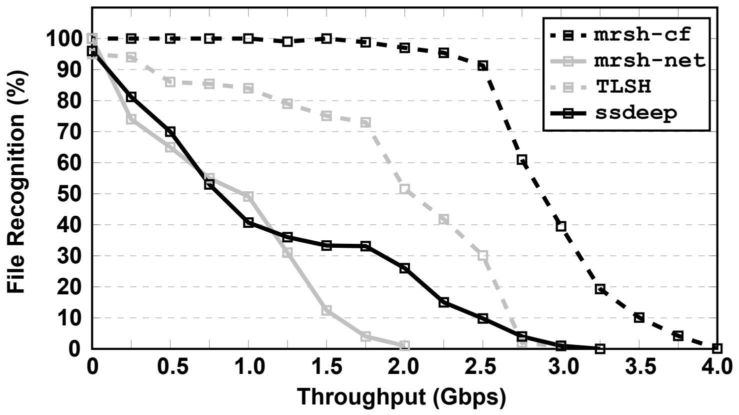

Figure 1. Initial throughput evaluations using the *t5-corpus*.

Figure 1 presents the results of initial evaluations of all the algorithms. The evaluations were performed on a workstation with a 3.5 GHz Intel Core i7-7500U mobile CPU (single-threaded). The throughput was incremented in steps of 250 Mbps and runs were performed ten times for each algorithm. The results show the efficacy of the algorithms at recognizing the randomized *t5-corpus* transferred over the loopback interface at various throughputs. Note that the non-optimized evaluations compared packet payloads against the algorithm hashes. The deviations over several runs were negligible. The evaluations demonstrate the ability of the algorithms to filter files correctly. Only true positives were considered, no false positives. Note that the filter performance of most of the algorithms decreased rapidly as the throughput increased.

The following statements can be made about the algorithms:

- The TLSH algorithm can detect finer-grained similarities compared with ssdeep, but it is more focused on differentiation than identification. TLSH returns a score for each comparison, where zero indicates a perfect match and higher scores indicate that the files are less similar. Only scores below 300 were accepted because they were either true or false positives. If the scoring threshold is set to a number below 300, there is a risk of not recognizing all the files. The 300 score appears to be the right threshold; if distances beyond this score are accepted, then the false positive rate becomes too high and would include true positives by accident. TLSH, which is best suited to evaluate the degree of similarity between two files, is used by VirusTotal to determine the similarities between malware variants. When TLSH was used to identify a single file in the *t5-corpus* (with no time constraints and counting all files below 300 as positives), up to 100 false positives were recorded depending on the filetype. Not visible in this evaluation, but nevertheless noteworthy, is that `TLSH` had a high false positive rate.

- The `ssdeep` algorithm was only able to detect file fragments containing 25-50% of the original files [27]. This issue may have been addressed in later versions of the algorithm. The evaluation used the standard version of `ssdeep` [35], which struggled to identify small fragments. With regard to the visibility of files on a per-payload basis, this means that smaller payloads cannot be matched correctly. The results show that `ssdeep` can filter some files at high throughputs that the other algorithms cannot handle. This is seen in Figure 1 as the long flattening curve that reaches 3.25 Gbps on the x-axis. The inability of `ssdeep` to match small fragments contained in payloads leads to poor detection rates.

- The evaluations reveal that `mrsh-cf` is by far the most powerful algorithm. Its superior cuckoo filter yields much better results than its predecessor `mrsh-net`. Further improvements were obtained because the `mrsh-cf` version used in this research incorporated an improved cuckoo filter compared with the original version. In the original `mrsh-net` evaluation [4], input packets had a uniform size of 18 chunks, and if 12 out of 18 chunks were found by the filter, then the packet was considered a match. As explained in Section 2.1, the evaluations conducted in this research employed variable-size payloads to better simulate real-world network scenarios.

- The `mrsh-net` algorithm was the only one originally intended for a network traffic matching scenario. The high discrepancy between the maximum throughput of 650 Mbps with a 99.6% true positive rate reported by the algorithm developers [4] and the results obtained in this research is probably due to the network scenarios being very different. Specifically, the developers of `mrsh-net` evaluated it in a simulated network scenario in which the *t5-corpus* was divided into equal-sized chunks (average TCP payload size of 1,460 bytes) that were input to the algorithm. The algorithm performed rather poorly in this evaluation due to the reduced runtime efficiency in a real network environment.

A closer investigation was conducted on the two best performing algorithms, `TLSH` and `mrsh-cf`. The interesting result is that the certainty with which the algorithms identified positives differed considerably. The

Table 4. mrsh-cf error rates for various throughputs.

	Throughput (Mbps)				
Measure	**500**	**1,000**	**1,500**	**2,000**	**2,500**
True Positives	2,000	2,000	2,000	1,997	1,891
False Positives	1,904	1,764	1,703	1,205	1,002
Average False Positive Size (%)	2.41	3.77	6.03	8.25	9.00
Average True Positive Size (%)	83.0	79.5	64.9	59.5	49.5

`mrsh-cf` algorithm found most chunks to be true positives and the false positives were mostly due to less than 10% matches of a file. Further filtering of false positives could be accomplished by applying a threshold. For example, if a file is identified with less then 10% of its chunks, then it is most likely a false positive and may be ignored. However, this applies only when a file is transferred in its entirety. When portions of a file are transferred, this heuristic might lead to false negatives.

In contrast, the `TLSH` algorithm identified true and false positives with the same certainty when the threshold for a positive was set below 300 (of 1,000). Positives were all identified as being 80% identical to the input file. Unlike, the `mrsh-cf` algorithm, a true positive is indistinguishable from among all the positives found by `TLSH`. This makes `mrsh-cf` the preferred algorithm for detecting unmanipulated files.

4. Experimental Results and Optimizations

In the evaluations, the `mrsh-cf` algorithm consistently achieved the highest detection rates of all algorithms. Using `mrsh-cf` as an exemplar, experiments were conducted to demonstrate the significance of the false positive rate and how an algorithm can be optimized using a heuristic. In the case of an approximate matching algorithm, a false positive corresponds to a file that was falsely matched. If an approximate matching algorithm is used to blacklist files in network traffic, then a high false positive rate can significantly reduce its utility.

Table 4 shows the number of false positives obtained for various throughputs. The main observation is that the number of false positives decreases with higher throughputs. This is because there are too many packets at higher throughputs and the `mrsh-cf` algorithm is unable to keep up and match all the data against the filter. Nevertheless, the ratio of true positives to false positives remains around 1/3.

The `mrsh-cf` algorithm is impractical as a packet filter because it drops too much traffic. In fact, it drops a packet as soon as it recognizes a file in its filter; this is the most aggressive form of payload filtering. However, the experiments show that this can be adjusted to increase accuracy while decreasing speed. The average false positive size shows how much of a false positive file was recognized on average. The algorithm can be adjusted based on this value to reduce its false positive rate, for example, by requiring at least $x\%$ of a file to be found before it is considered a match. With this threshold, the algorithm can be adjusted to mark files as positive only if a certain percentage of the target hash has been recognized. A suitable threshold enabled the false positive rate of the algorithm to be reduced by approximately 90%.

As mentioned above, in its fastest "mode," `mrsh-cf` can only determine the presence of a filter file in a payload. It can reveal how many chunks in the filter match the input payload, but not the exact file that matched because the filter does not hold this information. Only through exclusion – comparing file after file with the payload — can the algorithm narrow down the exact matching file.

The algorithm detection rate could be improved by chaining multiple instances of the algorithm that run in different modes. Harichandran et al. [23] have hinted at this possibility. First, a rapid (rough) preselection is performed using an instance of the algorithm that compares payloads against the filter. Next, all the positives are compared by individual instances of the algorithm that compare them against every file in the filter. As Breitinger and Baggili [4] have remarked, hashing algorithms can greatly benefit from parallelization by using multiple threads to compare hashes. As discussed below, the behavior of each of the selected algorithms was evaluated under multi-threading. The results for `mrsh-cf`, the fastest adapted algorithm, are presented.

Figure 2 shows that delegating the comparison task to two threads that do not require synchronization increased the throughput at which all the files were detected correctly by 100%. Specifically, the single threaded version attributed files 100% correctly at a maximum throughput of 1.5 Gbps whereas adding a second thread increased the throughput at which all files were detected correctly to 3 Gbps. Triple threading increased the throughput even further to 4 Gbps with a recognition rate of 97%. It can be assumed that adding more threads, which is not a problem with modern hardware, would increase the throughput even more while keeping the recognition rate constant.

However, the robustness of the algorithm may become an issue. At this time, noise and entangled files encountered in live traffic scenarios negatively impact robustness. Breitinger and Baggili [4] have demonstrated that pre-sorting the files using common substring filtering and

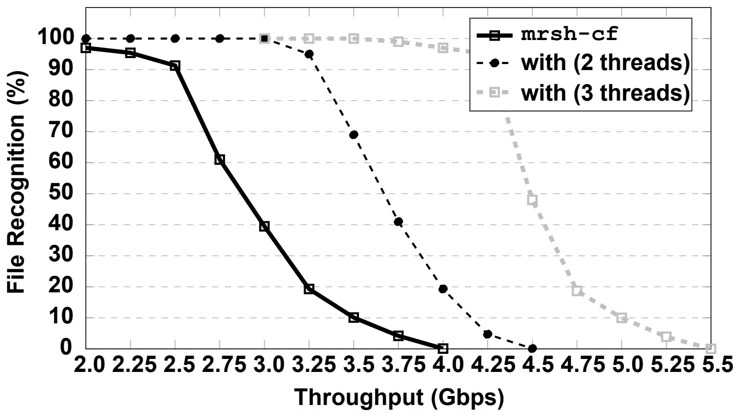

Figure 2. Throughput evaluations using the *t5-corpus* (threaded and normal).

measures of anti-randomness greatly benefit the application of the algorithm in live traffic scenarios.

5. Conclusions

This research has investigated the ability of prominent approximate matching algorithms to detect files in network traffic for purposes of data loss protection. In modern gigabit environments, network-focused data loss prevention solutions must be fast and reliable. For the first time, an effort has been made to adjust approximate matching algorithms to reliably recognize files at throughputs in the gigabit range. Optimizations have been introduced to render the algorithms more viable for live traffic detection. Indeed, the best algorithm, `mrsh-cf`, detects 97% of all files at a throughput of 4 Gbps in an idealized scenario. Open-source technology has been employed throughout this research and the algorithms are available to the digital forensics community at `github.com/dasec/approx-network-traffic`.

Future research will investigate techniques to reduce the false positive rates encountered when applying approximate matching algorithms in real-world network environments. The potential of matching several packets simultaneously to increase precision and incorporating multithreading to enhance file detection and throughput will be examined and the results integrated in the algorithms. Another problem involves reducing the bottleneck during payload inspection. Future work will also explore other promising approximate matching algorithms for file matching in network traffic; for example, Charyyev and Gunes [14] have shown that Nilsimsa hashes of Internet of Things traffic can be used as means for identification. A compelling problem is to perform holistic filtering of network traffic that would enable entire sets of network communications to be filtered by approximate matching techniques and matched by content and/or origin. This would address the challenges that encryption imposes on visibility in modern networks while refraining from TLS inspection.

Acknowledgement

This research was supported by the German Federal Ministry of Education and Research under Forschung an Fachhochschulen (Contract no. 13FH019IB6) and the Hessian Ministry of Higher Education, Research, Science and the Arts via their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

[1] S. Alneyadi, E. Sithirasenan and V. Muthukkumarasamy, A survey of data leakage prevention systems, *Journal of Network and Computer Applications*, vol. 62, pp. 137–152, 2016.

[2] H. Baier and F. Breitinger, Security aspects of piecewise hashing in computer forensics, *Proceedings of the Sixth International Conference on IT Security Incident Management and IT Forensics*, pp. 21–36, 2011.

[3] F. Breitinger, K. Astebol, H. Baier and C. Busch, `mvHash-B` – A new approach for similarity-preserving hashing, *Proceedings of the Seventh International Conference on IT Security Incident Management and IT Forensics*, pp. 33–44, 2013.

[4] F. Breitinger and I. Baggili, File detection in network traffic using approximate matching, *Journal of Digital Forensics, Security and Law*, vol. 9(2), pp. 23–36, 2014.

[5] F. Breitinger and H. Baier, A fuzzy hashing approach based on random sequences and Hamming distance, *Proceedings of the Annual ADFSL Conference on Digital Forensics, Security and Law*, pp. 89–100, 2012.

[6] F. Breitinger and H. Baier, Similarity-preserving hashing: Eligible properties and a new algorithm `MRSH-v2`, in *Digital Forensics and Cyber Crime*, M. Rogers and K. Seigfried-Spellar (Eds.), Springer, Berlin Heidelberg, Germany, pp. 167–182, 2013.

[7] F. Breitinger, H. Baier and J. Beckingham, Security and implementation analysis of the similarity digest `sdhash`, *Proceedings of the First International Baltic Conference on Network Security and Forensics*, 2012.

[8] F. Breitinger, B. Guttman, M. McCarrin, V. Roussev and D. White, Approximate Matching: Definition and Terminology, NIST Special Publication 800-168, National Institute of Standards and Technologies, Gaithersburg, Maryland, 2014.

[9] F. Breitinger, H. Liu, C. Winter, H. Baier, A. Rybalchenko and M. Steinebach, Towards a process model for hash functions in digital forensics, in *Digital Forensics and Cyber Crime*, P. Gladyshev, A. Marrington and I. Baggili (Eds.), Springer, Cham, Switzerland, pp. 170–186, 2014.

[10] F. Breitinger and K. Petrov, Reducing the time required for hashing operations, in *Advances in Digital Forensics IX*, G. Peterson and S. Shenoi (Eds.), Springer, Heidelberg, Germany, pp. 101–117, 2013.

[11] A. Breslow and N. Jayasena, Morton filters: Fast, compressed sparse cuckoo filters, *The VLDB Journal*, vol. 29(2-3), pp. 731–754, 2020.

[12] D. Chang, M. Ghosh, S. Sanadhya, M. Singh and D. White, `FbHash`: A new similarity hashing scheme for digital forensics, *Digital Investigation*, vol. 29(S), pp. S113–S123, 2019.

[13] D. Chang, S. Sanadhya and M. Singh, Security analysis of `MVhash-B` similarity hashing, *Journal of Digital Forensics, Security and Law*, vol. 11(2), pp. 22–34, 2016.

[14] B. Charyyev and M. Gunes, IoT traffic flow identification using locality-sensitive hashes, *Proceedings of the IEEE International Conference on Communications*, 2020.

[15] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi and P. Samarati, An open digest-based technique for spam detection, *Proceedings of the ICSA Seventeenth International Conference on Parallel and Distributed Computing Systems*, pp. 559–564, 2004.

[16] Editorial Team, Our work with the DNC: Setting the record straight, *CrowdStrike Blog*, June 5, 2020.

[17] B. Fan, D. Andersen, M. Kaminsky and M. Mitzenmacher, Cuckoo filter: Practically better than Bloom, *Proceedings of the Tenth ACM International Conference on Emerging Networking Experiments and Technologies*, pp. 75–88, 2014.

[18] S. Garfinkel, P. Farrell, V. Roussev and G. Dinolt, Bringing science to digital forensics with standardized forensic corpora, *Digital Investigation*, vol. 6(S), pp. S2–S11, 2009.

[19] S. Gatlan, Software AG, IT giant, hit with $23 million ransom by Clop ransomware, *BleepingComputer*, October 9, 2020.

[20] T. Graf and D. Lemire, XOR filters: Faster and smaller than Bloom and cuckoo filters, *ACM Journal of Experimental Algorithmics*, vol. 25(1), article no. 5, 2020.

[21] V. Gupta and F. Breitinger, How cuckoo filters can improve existing approximate matching techniques, in *Digital Forensics and Cyber Crime*, J. James and F. Breitinger (Eds.), Springer, Cham, Switzerland, pp. 39–52, 2015.

[22] N. Harbour, `dcfldd` version 1.3.4-1 (`dcfldd.sourceforge.net`), 2006.

[23] 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. 59–78, 2016.

[24] J. Kornblum, Identifying almost identical files using context-triggered piecewise hashing, *Digital Investigation*, vol. 3(S), pp. 91–97, 2006.

[25] V. Martinez, F. Hernandez-Alvarez and L. Encinas, An improved bytewise approximate matching algorithm suitable for files of dissimilar sizes, *Mathematics*, vol. 8(4), article no. 503, 2020.

[26] 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.

[27] A. Lee and T. Atkison, A comparison of fuzzy hashes: Evaluation, guidelines and future suggestions, *Proceedings of the ACM SouthEast Conference*, pp. 18–25, 2017.

[28] D. Lillis, F. Breitinger and M. Scanlon, Expediting `MRSH-v2` approximate matching with hierarchical Bloom filter trees, in *Digital Forensics and Cyber Crime*, P. Matousek and M. Schmiedecker (Eds.), Springer, Cham, Switzerland, pp. 144–157, 2018.

[29] F. Pagani, M. Dell'Amico and D. Balzarotti, Beyond precision and recall: Understanding uses (and misuses) of similarity hashes in binary analysis, *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, pp. 354–365, 2018.

[30] V. Roussev, Building a better similarity trap with statistically-improbable features, *Proceedings of the Forty-Second Hawaii International Conference on System Sciences*, 2009.

[31] 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.

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

[33] V. Roussev, G. Richard and L. Marziale, Multi-resolution similarity hashing, *Digital Investigation*, vol. 4(S), pp. S105–S113, 2007.

[34] C. Sadowski and G. Levin, SimHash: Hash-Based Similarity Detection, Technical Report, Department of Computer Science, University of California Santa Cruz, Santa Cruz, California, 2007.

[35] `ssdeep` Project, `ssdeep` – Fuzzy Hashing Program, GitHub (`ssdeep-project.github.io/ssdeep`), April 11, 2018.