How Cuckoo Filter Can Improve Existing Approximate Matching Techniques

Vikas Gupta[1] and Frank Breitinger[2]

[1] Netskope, Inc.
vikasgupta.nit@gmail.com
[2] Cyber Forensics Research and Education Group (UNHcFREG)
Tagliatela College of Engineering University of New Haven, West Haven CT, 06516, United States
FBreitinger@newhaven.edu

Abstract. In recent years, approximate matching algorithms have become an important component in digital forensic research and have been adopted in some other working areas as well. Currently there are several approaches, but `sdhash` and `mrsh-v2` especially attract the attention of the community because of their good overall performance (runtime, compression and detection rates). Although both approaches have quite a different proceeding, their final output (the similarity digest) is very similar as both utilize Bloom filters. This data structure was presented in 1970 and thus has been used for a while. Recently, a new data structure was proposed which claimed to be faster and have a smaller memory footprint than Bloom filter – *Cuckoo filter*.
In this paper we analyze the feasibility of Cuckoo filter for approximate matching algorithms and present a prototype implementation called `mrsh-cf` which is based on a special version of `mrsh-v2` called `mrsh-net`. We demonstrate that by using Cuckoo filter there is a runtime improvement of approximately 37% and also a significantly better false positive rate. The memory footprint of `mrsh-cf` is 8 times smaller than `mrsh-net`, while the compression rate is twice than Bloom filter based fingerprint.

Key words: Approximate matching, similarity hashing, Bloom filter, Cuckoo filter.

1 Introduction

Approximate matching (a.k.a. similarity hashing or fuzzy hashing) is a technology to identify similarities among digital artifacts and can be seen as a counterpart to traditional (cryptographic) hash functions. According to the definition of Breitinger et al. [7], approximate matching algorithms can not only be used to detect similarities among objects, but also to detect embedded objects or fragments of objects.

Within the last decade, the community has proposed several approximate matching algorithms starting with `ssdeep` by Kornblum [14]. Four years later, Roussev presented a new algorithm called `sdhash` [21] which outperforms `ssdeep`

with respect to precision and recall but was slightly slower. Therefore, Breitinger et. al. combined both implementations and published `mrsh-v2` [5]. Compared to `sdhash`, this new algorithm had less complexity and thus significant advantage with respect to runtime efficiency, however `sdhash` remained slightly more accurate [8].

One of the main working fields of approximate matching is digital forensics where investigators use it to reduce the amount of data automatically, e.g., filter out non-relevant data like OS-files and highlight suspect files. Since a single case can consist of several hundreds of gigabytes, it is important to have fast and reliable algorithms. Due to the continuous improvements over recent years, these algorithms have gotten very powerful and been applied in further areas, e.g., deep network packet analysis and detecting known file fragments in packets [13, 4] or in biometrics and iris recognition [20].

From a high level perspective `ssdeep`, `sdhash` and `mrsh-v2` work similarly. They all use some intermediary logic to extract features from the given input, compress these features and finally print out a similarity digest a.k.a. fingerprint. Accordingly, in order to find similar objects, we compare fingerprints instead of comparing complete objects. Depending on the overlap of the fingerprints, a similarity score is returned. While `ssdeep` uses a Base64 encoded sequence to represent the fingerprint, both of the other algorithms utilize Bloom filters [2]. A Bloom filter is an excellent data structure for set-membership queries, offers great memory efficiency (compression) and has been studied extensively (more details are given in Sec. 2.1).

Recently, Fan et. al. [10] have proposed a new data structure for set-membership queries, called *Cuckoo filter*. A "Cuckoo filter is a compact variant of cuckoo hash table [18] that stores only *fingerprints* – a bit string derived from the item using a hash function – for each item inserted, instead of key-value pairs". The authors demonstrate that a Cuckoo filter is practically better than Bloom filters in terms of performance and space overhead.

In this paper we explore the possibility of using Cuckoo filter for approximate matching. We build a new version of `mrsh-net` (which is a fork of `mrsh-v2`) that uses Cuckoo filter and demonstrate in our evaluation section that there is a practical performance improvement compared to the Bloom filter version. Note that we only manipulate and evaluate the fingerprint representation of the approximate matching algorithm which means other existing algorithms (e.g, `sdhash`) could profit from our findings, too. The technical contribution of this paper include:

- Exploring the feasibility of Cuckoo filter as a more efficient alternative to Bloom filter.
- Developing a prototype by implementing `mrsh-net` with Cuckoo filter, called `mrsh-cf`.
- Analyzing some key performance parameters like runtime efficiency, memory usage and compression.

We want to point out that this is the first paper trying to use a new fingerprint representation. Prior to this work, no one has attempted to enhance approximate

matching algorithms by using alternative data structures to the widely used Bloom filter.

The rest of the paper is organized as follows: In Sec. 2, we discuss the related work which includes Bloom filter, `mrsh-v2` and its branch `mrsh-net`. Next, in Sec. 3, we briefly mention cuckoo hashing which is the basis of Cuckoo filter followed by a comprehensive description of Cuckoo filter, e.g., workflow of insertions and lookups. In Sec. 4 we present our assessment and experimental results. Sec. 5 concludes this paper.

2 Background and related work

The introduction briefly mentioned the three most popular approximate matching algorithms. However, for the remainder of this paper we focus on `mrsh-v2`, since `ssdeep` can be overcome by an active adversary [1] and `sdhash` is rather complex which makes it harder to integrate our changes.

In the following subsections, we first explain Bloom filter in Sec. 2.1 followed by an overview of `mrsh-v2` including its branch `mrsh-net`.

2.1 Bloom filter

A Bloom filter is a probabilistic data structure used to test the membership of an element against a given set and was introduced by Burton H. Bloom [2]. Since then, it has found use in different applications like networks [9] and approximate matching (`mrsh-v2` and `sdhash`) algorithms.

From a programming perspective, a Bloom filter is a bit array of length m (all set to zero), that provides a compact representation of a set S, containing $|S|$ elements. Normally it supports two operations: *insert* and *lookup*.

In order to perform an operation, we need k independent hash functions that output values in the range of $0 \leq h(s) \leq m-1$ for all $s \in S$. To *insert* s into a Bloom filter, the item is hashed using k different hash functions and the corresponding bits in the Bloom filter are set to one. The *lookup* is performed in a similar manner, the element is hashed using the k hash functions and the corresponding bits are checked, if all bits are set to one, the query returns true; otherwise returns a false.

Thus, a query to a Bloom filter returns either 'definitely no' or 'probably yes' (bits might be set to one but by different elements). The probability of being wrong ϵ (i.e. the false positive rate) is tunable and given by Eq. 1

$$\epsilon \approx (1 - e^{-kn/m})^k \tag{1}$$

where, k is the number of hash functions and n is the number of elements added to the Bloom filter. The lower the value of ϵ, the larger the bit-size m of the Bloom filter.

Bloom filter is a very space-efficient approach to represent a set of digital objects. The space used by Bloom filter is much less than compared to the

space occupied by the original set. Over the years, many extensions/mutations of Bloom filter were presented like the counting Bloom filter [11], blocked Bloom filter [19] or d-left counting Bloom filter [3]. However, for the remainder of the paper any further reference to Bloom filter is with respect to standard Bloom filter.

As argued by Pagh et. al. [17], there are more efficient ways to represent $|S|$ than Bloom filters. For a false positive rate ϵ, a space-optimized Bloom filter uses $k = \log_2(1/\epsilon)$ hash functions. As per information-theory, minimum $\log_2(1/\epsilon)$ bits are required per item, while Bloom filter uses $1.44 \cdot \log_2(1/\epsilon)$ bits. The number of bits per item is dependent on ϵ, rather than item size or total number of items.

2.2 mrsh-v2 and mrsh-net

As aforementioned, `mrsh-v2` is a very straightforward algorithm that was proposed by Breitinger et. al. [5]. Its proceeding is quite simple. `mrsh-v2` identifies trigger points in any given byte sequence (e.g., a file or a device), that are used to divide it into chunks of approximately *bs* bytes. Next, each chunk is hashed using FNV [16]. In order to insert a 64 bit FNV chunk hash into a $m = 2048 = 2^{11}$ bits Bloom filter, `mrsh-v2` builds five 11-bit sub-hashes based on the least significant 55 bits of the FNV hash. Finally, each sub-hash sets one bit within the Bloom filter. Note, instead of inserting complete files into the Bloom filter (as explained in Sec. 2.1), we insert chunks which then allow the similarity identification.

`mrsh-v2` allows a maximum of 160 chunks per Bloom filter. If this limit is reached a new Bloom filter is created. Hence, the final fingerprint for an input can be a sequence of multiple Bloom filters. As a consequence, we have variable fingerprint lengths in contrast to the traditional definition of hash functions where we have fixed output lengths [15]. Comparing two fingerprints is a comparison of all Bloom filters of fingerprint A against all Bloom filters of fingerprint B with respect to the Hamming distance as metric.

Despite all the benefits offered by Bloom filters, there is one major issue – the database lookup complexity. Currently there is no technique to sort/order Bloom filter based similarity digests and thus comparing a single digest against a database containing x entries requires an 'against-all' comparison – a complexity of $O(x)$. In contrast, cryptographic digests can be stored in binary trees or organized in hash tables having a lookup complexity of $O(\log_2(x))$ or $O(1)$, respectively. The impact is best demonstrated on an example: comparing 1.8 GB of data with itself (the t5-corpus[1] containing 4457 files), `mrsh-v2` takes over 21 minutes, which is too slow for practical usage.

mrsh-net. To overcome this drawback, Breitinger et. al. [6] suggested a different strategy resulting in a complexity of $O(1)$. The idea is simple: instead of having multiple small Bloom filters, they recommended having a single large Bloom filter that contains all files (actually all chunks of all files). While this reduces the lookup times significantly, it loses information about successful matches.

[1] `http://roussev.net/t5/t5.html` (last accessed 2015-04-10).

While the original implementation was a fingerprint vs. fingerprint comparison, `mrsh-net` is a fingerprint vs. set comparison. In other words, a query only returns *true* or *false*, but does not return the file that it is matched to.

3 Cuckoo Filter

The overall idea of this paper is to demonstrate the feasibility and improvements of using *Cuckoo filter* instead of Bloom filters for approximate matching. This section explains the details about the concept. If you are already familiar with Cuckoo filter, you may skip this section.

The idea of Cuckoo filter originated from Cuckoo hashing, which was proposed by Pagh et. al. [18] and is comparable to a *dictionary* data structure, i.e., there are keys and values. Usually the key is generated out of the value and has extremely fast access time. However, unlike a traditional dictionary structure, Cuckoo hashing utilizes *two* hash functions and therefore utilizes two keys and two tables. Accordingly, each lookup has a constant lookup time of 2 queries and expected constant amortized[2] time for updates[3]. Explaining Cuckoo hashing in detail is beyond the scope of this paper, however. We will focus on Cuckoo filter in the subsequent paper.

Fan et. al. in [10] modified Cuckoo hashing so that it could handle set-membership queries and called it Cuckoo filter. In their paper they demonstrated that when compared to Bloom filter, Cuckoo filter

- has a better lookup performance (with respect to runtime),
- has a better space efficiency for applications requiring low false positive rates ($\epsilon < 3\%$) and
- supports deleting items dynamically (this property is irrelevant for us).

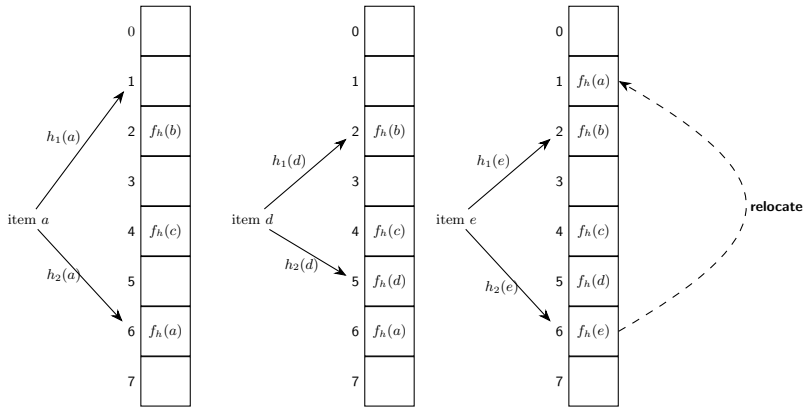
Generally speaking, a Cuckoo filter consists of a Cuckoo hash table and three hash functions named h_1, h_2 and f_h, where each position in the hash table is called a *bucket* and can store multiple *entries*. The three hash functions are used as follows: h_1 and h_2 identify the correct buckets for insertion or lookup (identify the position), and f_h is used to compress the item (to save memory space, a Cuckoo filter does not store the items themselves but a constant-sized hash of the items). For the remainder of this paper, the following terminology is used:

m - the size of a Cuckoo filter, i.e., number of buckets.
b - the bucket size, i.e the number of *entries* each bucket can have.
h_1, h_2 - the hash functions to obtain the positions in the Cuckoo filter for a given item.
f_h - the hash function used to obtain a tag for an item (compresses the item), where $|f_h|$ is the bit length of the hash value.

[2] `https://en.wikipedia.org/wiki/Amortized_analysis` (last accessed 2015-04-10).
[3] I.e, this overcomes chained hash table where worst case time for lookup will be linear $O(n)$.

3.1 Insert

The insertion process is best described by the example given in Fig. 1, where we have a Cuckoo filter with $m = 8$ and $b = 1$. For completeness, we printed the pseudo code in the Appendix under Algorithm 1. As shown in the figure, there are three possibilities:



(a) Inserting a at index 6. (b) Inserting d at index 5. (c) Inserting item e at index 6.

Fig. 1. Illustration of Cuckoo Filter where hash functions h_1 and h_2 are used for determining buckets for an item's insertion and f_h to get a constant-sized hash of the item to be inserted. Initial setup (a): the items b and c are already in the Cuckoo filter while item a is processed.

- **Both buckets are empty**: In Fig. 1 (a), item a can be inserted in bucket 1 or 6. Since both the buckets are empty, the final bucket for insertion is determined randomly (ensures buckets fill up equally). In the example, a is inserted into bucket 6.
- **One bucket is full**: In Fig. 1 (b), item d can go in bucket 2 or 5. Since, bucket 2 is already occupied, d is inserted into bucket 5.
- **Both buckets are full**: In Fig. 1 (c), item e can go in bucket 2 or 6. However, both buckets are occupied, which requires relocation of entries.

Relocation. The idea of relocation is to move an existing entry to its alternate bucket. For instance, in Fig. 1 (a), we randomly inserted a into bucket 6, while bucket 1 remained empty. On relocation (Fig. 1 (c)), we now move a from 6 to 1, which allows us to insert e into 6.[4]

[4] Since Cuckoo filter only store the hash of an item (the entry) and not the item itself, it is not possible to rehash an item and identify the other bucket. Therefore, the authors implemented the location hash functions (h_1 and h_2) in a manner allowing them to be derived from the current location and the entry: $h_1(x) = hash(x)$ and $h_2(x) = h_1(x) \oplus hash(f_h(x))$ where $hash$ is any hash function.

The relocation-process can carry on until a vacant bucket is found, or the maximum number of relocations is reached (500 in present implementation). In the latter case, the table is considered to be full.

Identical tags. By design Cuckoo filter allows identical tags in buckets (e.g., the same item is inserted multiple times), which allows that those items can be deleted. In theory, the same tag cannot be entered more than $2b$ times, as then both buckets are full and cannot be relocated (= full Cuckoo filter). Since, our approach does not require deleting entries, we only insert unique tags and ignore duplicates.

3.2 Lookup

The lookup process of a Cuckoo filter is fairly straightforward. For querying an item x, firstly $h_1(x)$ and $h_2(x)$ are calculated and then both buckets are checked for the presence of the $f_h(x)$. If the tag is present in either of the buckets, Cuckoo filter returns *true* or *false*.

3.3 False positive rate of a Cuckoo filter

As discussed in [10], the false positive rate ϵ for a Cuckoo filter depends on the bucket size b and on the tag size $|f_h|$. Since ϵ is usually a system requirement, we can estimate $|f_h|$ by:

$$|f_h| \geq \lceil log_2(2b/\epsilon) \rceil = \lceil log_2(1/\epsilon) + log_2(2b) \rceil \text{ bits.} \quad (2)$$

If the total size of the Cuckoo filter is kept constant, b influences the length as follows:

– **A larger b improves table occupancy:** A larger b reduces the chance to get a full Cuckoo filter and hence a higher load factor α can be achieved, where α is the ratio of number of entries made (total insertions made) divided by total number of entries possible. An overview is given in Table 1.

Table 1. Correlation between bucket size and load factor in a Cuckoo filter [10].

Bucket size b	**Load factor α**
1	50%
2	84%
4	95%
8	98%

– **A larger b requires a larger tag size to retain the same false positive rate:** A larger b increases the chance of collisions, as for each lookup more entries are checked. By increasing the tag size, the chances of having similar entries is reduced. Fig. 2 shows the relation between bits/item used and false positive rate achieved for various bucket sizes.

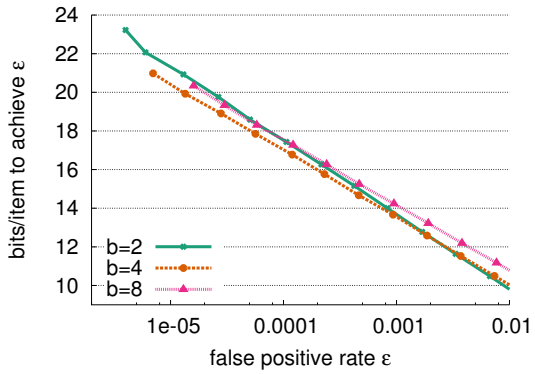


Fig. 2. Amortized space cost per item vs. measured false positive rate, with different bucket size b = 2,4,8 [10].

To conclude, a Bloom filter requires approximately 10 bits per item to achieve $\epsilon = 1\%$, regardless of whether one thousand, one million, or billion items are stored. In contrast Cuckoo filters require bigger tag size to retain the same high space efficiency of their hash tables, but lower false positive rates are achieved accordingly [10].

By considering Table 1 and Fig. 2, a high space efficiency and low false positive rate can be achieved by using $b = 4$. Hence, we will use (2,4)-Cuckoo filter, i.e, each item has two candidate buckets and each bucket has up to four entries.

3.4 Number of memory access

For a Bloom filter with k hash functions, a positive query must read k bits from the bit array where ideally $k = log_2(1/\epsilon)$. As ϵ gets smaller, positive queries require to probe more bits and are likely to incur more cache line misses when reading each bit. In the case of a negative query, half of k bits are read before a false is returned. Contrastingly, for a Cuckoo filter, for both positive or negative queries, it requires a fixed number of reads, resulting in at most two cache line misses. Note, the authors assume that memory access is expensive while comparing is cheap [10].

3.5 Construction rate

As discussed in [10], given the same false positive rate for both Bloom filter and Cuckoo filter and both filters configured to use same size of 192 MB, the construction rate of Cuckoo filter is higher than Bloom filter. 5.00 million keys/sec can be added to Cuckoo filter, as opposed to only 3.13 million keys/sec. This gives a clear indication of performance gains one can achieve by using Cuckoo filter over Bloom filter.

It is important to note that Cuckoo filter is a fairly new data structure and many of its characteristics have not been studied in detail. Several theoretical questions, however, remain open for future study - a couple being providing bounds on the cost of inserting a new item and studying how much independence is required of the hash functions. But as we will see in Sec. 4, Cuckoo filters do increase the performance of `mrsh-net` and any further improvement in Cuckoo filter will further enhance the performance of `mrsh-net`.

4 Assessment and experimental results

This section firstly gives some implementation details (Sec. 4.1) and describes the comparison criteria (Sec. 4.2). Next, we explain our experimental setup in Sec. 4.3 which is followed by a discussion about the false positives and further results in Sec. 4.4 and in Sec. 4.5, respectively.

4.1 Implementation details

While `mrsh-net` is implemented in C, we programed our reference implementation in C++. In order to integrate Cuckoo filter, we included the library[5] provided by Fan et. al. [10]. The chunk size of `mrsh-net` is set to $bs = 320$.

It is important to note that we are no experts in writing fully optimized C/C++ programs but we tried our best to make both implementations as optimized as possible. Our prototype of `mrsh-cf` is released and can be downloaded from our website[6].

The default implementation of Cuckoo filter uses SHA-1 [12] and a variant of Austin Appleby's MurmurHash2[7] for hashing tasks. When an item x is inserted in a Cuckoo filter, it is first hashed using SHA-1. The first part of the hash obtained acts as $h_1(x)$, while the second part represents $f_h(x)$ (the tag of x). To calculate $h_2(x)$, a variant of MurmurHash2 is used.

4.2 Comparison parameters

In [8] several criteria were proposed to compare approximate matching algorithms, which can be categorized into:

- Efficiency: includes runtime efficiency and compression rate.
- Sensitivity & robustness: includes random-noise-resistance, alignment robustness, fragment detection, and file correlation.

For this article we focus on the first category while "sensitivity & robustness" remains for future work. Apart from the above discussed parameters, the two versions are also compared by memory usage.

[5] `https://github.com/efficient/cuckoofilter` (last accessed 2015-04-10).
[6] `http://www.fbreitinger.de/?page_id=218` (last accessed 2015-04-10).
[7] `https://code.google.com/p/smhasher/wiki/MurmurHash2` (last accessed 2015-04-10).

4.3 Experimental setup

The development and testing was done on a machine with an Intel Core i5 1.80 GHz quad-core processor with 8 GB RAM and 3 MB L3 cache (Linux kernel 3.16). Both, `mrsh-net` and `mrsh-cf` were compiled using GCC 4.9 with highest possible compiler optimization `-O3`. The program's execution time and memory usage were recorded using the GNU `time`-command[8]. For certain cases, C/C++ header *time.h* was also used to determine execution time.

We decided to run our tests on the *t*5-corpus which consisted of 4457 files [22]. This set is widely used within the field of digital forensics and includes many common file types like html, jpg or doc. The main facts are given in Table 2.

Table 2. Statistics of t5-corpus.

Number of files	4457
Total size	1.8 GB
Minimum file size	4.0 KB
Maximum file size	17.4 MB
Average file size	428.72 KB
Number of file types	8

Also, each test ran 10 times for computing various performance parameters for both the versions.

4.4 False positive rate

`mrsh-net` aims for a false positive rate of $\epsilon = 6.33 \cdot 10^{-5}$. According to Eq. 2 and setting the bucket size to $b = 4$, this requires a tag size of $|f_h| \geq 17$ bits.

However, as outlined by Table 3, $|f_h|$ does not affect the actual runtime efficiency by much (the max RSS column is discussed in Sec. 4.5). The table shows the runtime for $2 \leq |f_h| \leq 32$.

Using $|f_h| = 32$ improves the false positive rate to approximately $\epsilon \approx 10^{-9}$, but also increases the total memory footprint. We accept this disadvantage to attain a better false positive rate. As shown later, `mrsh-cf` is still more space efficient than `mrsh-net`.

4.5 Testing results

As concluded in the last section, all the results were computed with Cuckoo filter configuration of tag size $|f_h| = 32$ bits and bucket size $b = 4$. Number of buckets m for the Cuckoo filter was obtained by dividing input object size by chunk size bs in bytes. The comparison of `mrsh-cf` against `mrsh-net` with respect to various parameters is presented below:

[8] `http://man7.org/linux/man-pages/man1/time.1.html` (last accessed 2015-04-10).

Table 3. Execution time (in seconds) and memory usage (in KB) for `mrsh-net` and various configurations of `mrsh-cf`. The version corresponds to 'mrsh-cf-b-$|f_h|$', e.g., `mrsh-cf`-4-2 means `mrsh-cf` having $b = 4$ and $|f_h| = 2$.

Version	**Exec. time** (sec)	**Maximum RSS** (KB)
mrsh-net	65.84	401396
mrsh-cf-4-2	40.03	35912
mrsh-cf-4-4	40.35	36844
mrsh-cf-4-8	40.28	38692
mrsh-cf-4-12	40.13	40944
mrsh-cf-4-16	40.06	42824
mrsh-cf-4-32	41.47	51165
mrsh-cf-8-16	39.69	51024
mrsh-cf-8-32	40.01	67452

Time efficiency. To determine the runtime efficiency, we measured the execution time for three scenarios. First, the time to generate the Cuckoo filter for t5-corpus. Next, to compare t5-corpus against a pre-generated Cuckoo filter. Finally, the time taken to perform an all-against-all comparison. This last scenario should be approximately the same as the sum of the two previous tests. Results are presented in Table 4.

Table 4. Comparison between `mrsh-cf` and `mrsh-net` on various performance parameters.

Parameters	`mrsh-cf`	`mrsh-net`
Cuckoo filter generation time (sec)	19.74	32.90
Comparison time (sec)	21.17	33.09
Against-all comparison time (sec)	41.47	65.84
Maximum RSS (KB)	51165	401396
Fingerprint size (MB)	16	32

As can be seen, `mrsh-cf` outperforms the original implementation in all categories. For instance, generating the filter takes 19.74s and 32.90s, respectively, which is a time difference of about 40%.

With respect to the all-against-all comparison, the Cuckoo filter for the complete t5-corpus was generated and then each file in the corpus was compared against this filter. The test showed that `mrsh-cf` requires 41.47s while `mrsh-net` needed 65.84s. To conclude, `mrsh-cf` has a significant performance gain of approximately 37%.

Compression. Traditional cryptographic hash functions return a hash value of constant size. On the other hand, approximate matching algorithms return digests of variable length. As digests are typically stored within a database, preferably short fingerprints are desirable. Therefore, compression measures the ratio between input and output size of an algorithm [8].

$$compression = \frac{output\ length}{input\ length} \cdot 100\ . \tag{3}$$

With `mrsh-cf`, the filter for t5-corpus has a size 16MB, while `mrsh-net` results in a filter size of 32MB. Accordingly, the compression for `mrsh-cf` is superior since it only needs half the size. Considering both kinds of filters, it can be safely concluded that the loading time for `mrsh-cf` will be faster than `mrsh-net`.

Memory Usage. Memory usage is calculated by determining *Maximum Resident Set Size* of a process using GNU `time`-command. 'Resident set size' (RSS) of a process represents the amount of non-swapped memory the kernel has already allocated to the process. This does not include the swapped out portion of the memory. As indicated by the name, maximum RSS is the maximum memory assigned to a process during its lifetime.

The results are shown in Table 3. The maximum RSS for `mrsh-cf` relies on the configuration. However, in all used configurations it was much smaller than `mrsh-net`. Note, this can provide a significant performance advantage when handling huge data sets as the majority of the filters will be present in memory.

5 Conclusion

In this paper, we demonstrated that Cuckoo filters have significant benefits over Bloom filters within the area of approximate matching, which is an important technology for digital forensics. While all previous work tried to improve the algorithms themselves, this is the first identification of an alternative fingerprint / similarity digest representation scheme.

Our results show that Cuckoo filters provide two major improvements over Bloom filters: (1) better lookup performance; and (2) better space efficiency for applications requiring a low false positive rate ($\epsilon \geq 3\%$). We concluded that Cuckoo filters are a superior compared to Bloom filters in approximate matching algorithms.

Furthermore, we released a prototype named `mrsh-cf` that is based on `mrsh-net` and Cuckoo filter library. This prototype outperformed its predecessor in all considered parameters. For example, the runtime efficiency was about 37% faster while using 8 times less memory. The size of the filter generated using `mrsh-cf` was only half the size of the `mrsh-net` filter.

Although there are several benefits of using Cuckoo filter, we also face new challenges. Comparing two Cuckoo filter-based fingerprints is not as straightforward as using traditional Bloom filter fingerprints. However, the initial tests from this current work show promising (but not perfect) results.

References

1. Harald Baier and Frank Breitinger. Security Aspects of Piecewise Hashing in Computer Forensics. *IT Security Incident Management & IT Forensics (IMF)*, pages 21–36, May 2011.
2. Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
3. Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An improved construction for counting bloom filters. In *Algorithms–ESA 2006*, pages 684–695. Springer, 2006.
4. Frank Breitinger and Ibrahim Baggili. File detection on network traffic using approximate matching. *Journal of Digital Forensics, Security and Law (JDFSL)*, 9(2):23–36, September 2014.
5. Frank Breitinger and Harald Baier. Similarity preserving hashing: Eligible properties and a new algorithm mrsh-v2. In Marcus Rogers and KathrynC. Seigfried-Spellar, editors, *Digital Forensics and Cyber Crime*, volume 114 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 167–182. Springer Berlin Heidelberg, October 2013.
6. Frank Breitinger, Harald Baier, and Douglas White. On the database lookup problem of approximate matching. *Digital Investigation*, 11:S1–S9, 2014.
7. Frank Breitinger, Barbara Guttman, Michael McCarrin, Vassil Roussev, and Douglas White. Approximate matching: Definition and terminology. Special publication 800-168, National Institute of Standards and Technologies, May 2014.
8. Frank Breitinger, Georgios Stivaktakis, and Harald Baier. Frash: A framework to test algorithms of similarity hashing. *Digit. Investig.*, 10:S50–S58, August 2013.
9. Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet mathematics*, 1(4):485–509, 2004.
10. Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88. ACM, 2014.
11. Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):281–293, 2000.
12. Patrick Gallagher and Acting Director. Secure Hash Standard (SHS). Technical report, National Institute of Standards and Technologies, Federal Information Processing Standards Publication 180-1, 1995.
13. Vikas Gupta. File detection in network traffic using approximate matching. Master's thesis, Technical University of Denmark, Copenhagen, Denmark, 2013.
14. Jesse Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3:91–97, September 2006.
15. Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*, volume 5. CRC Press, August 2001.
16. Landon Curt Noll. Fnv hash. `http://www.isthe.com/chongo/tech/comp/fnv/index.html`, 1994–2012.
17. Anna Pagh, Rasmus Pagh, and S Srinivasa Rao. An optimal bloom filter replacement. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 823–829. Society for Industrial and Applied Mathematics, 2005.
18. Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

19. Felix Putze, Peter Sanders, and Johannes Singler. Cache-, hash-and space-efficient bloom filters. In *Experimental Algorithms*, pages 108–121. Springer, 2007.
20. Christian Rathgeb, Frank Breitinger, Christoph Busch, and Harald Baier. On application of bloom filters to iris biometrics. *Biometrics, IET*, 3(4):207–218, August 2014.
21. Vassil Roussev. Data fingerprinting with similarity digests. In Kam-Pui Chow and Sujeet Shenoi, editors, *Advances in Digital Forensics VI*, volume 337 of *IFIP Advances in Information and Communication Technology*, pages 207–226. Springer Berlin Heidelberg, 2010.
22. Vassil Roussev. An evaluation of forensic similarity hashes. *Digital Investigation*, 8:34–41, August 2011.

Appendix

Algorithm 1 Pseudo code for insertion copied from [10] and slightly modified x.

```
1: f = fingerprint(x);
2: /* this is h_1 */
3: i_1 = hash(x);
4: /* this is h_2 */
5: i_2 = i_1 ⊕ hash(f);
6: if bucket[i_1] or bucket[i_2] has an empty entry then
7:     add f to that bucket;
8:     return Done;
9:
10: /* must relocate existing items */
11: i = randomly pick i_1 or i_2;
12: for n = 0; n < MaxNumKicks; n++ do
13:     randomly select an entry e from bucket[i];
14:     swap f and the fingerprint stored in entry e;
15:     i = i ⊕ hash(f);
16:     if bucket[i] has an empty entry then
17:         add f to bucket[i];
18:         return Done;
19:
20: /* Hashtable is considered full */
21: return Failure;
```