Digital Investigation 11 (2014) S1–S9

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

On the database lookup problem of approximate matching

Frank Breitinger [a,b,*,1,2], Harald Baier [a,1], Douglas White [b,2]

[a] da/sec – Biometrics and Internet Security Research Group, Hochschule Darmstadt, Haardtring 100, 64295 Darmstadt, Germany
[b] National Institute for Standards and Technologies, 100 Bureau Dr, Gaithersburg, MD 20899, United States

Keywords:
Digital forensics
Approximate matching
Bloom filter
sdhash
mrsh-v2
Indexing
Blacklisting

A B S T R A C T

Investigating seized devices within digital forensics gets more and more difficult due to the increasing amount of data. Hence, a common procedure uses automated file identification which reduces the amount of data an investigator has to look at by hand. Besides identifying exact duplicates, which is mostly solved using cryptographic hash functions, it is also helpful to detect similar data by applying approximate matching.

Let x denote the number of digests in a database, then the lookup for a single similarity digest has the complexity of $O(x)$. In other words, the digest has to be compared against all digests in the database. In contrast, cryptographic hash values are stored within binary trees or hash tables and hence the lookup complexity of a single digest is $O(\log_2(x))$ or $O(1)$, respectively.

In this paper we present and evaluate a concept to extend existing approximate matching algorithms, which reduces the lookup complexity from $O(x)$ to $O(1)$. Therefore, instead of using multiple small Bloom filters (which is the common procedure), we demonstrate that a single, huge Bloom filter has a far better performance. Our evaluation demonstrates that current approximate matching algorithms are too slow (e.g., over 21 min to compare 4457 digests of a common file corpus against each other) while the improved version solves this challenge within seconds. Studying the precision and recall rates shows that our approach works as reliably as the original implementations. We obtain this benefit by accuracy–the comparison is now a file-against-set comparison and thus it is not possible to see which file in the database is matched.

Introduction

Handling hundreds of thousands of files is a major challenge in today's digital forensics. In order to cope with this information overload, investigators often apply hash functions for automated input identification. A common processing is *known file filtering* which is quite simple: compute the hashes for all files on a target device and compare them to a reference database. Depending on the underlying database, files are either filtered out (e.g., files of the operating system) or filtered in (e.g., known offensive content). A very common database for 'filter out' data is the National Software Reference Library (NSRL) (NIST Information Technology Laboratory, 2013) maintained by National Institute for Standards and Technologies (NIST).

Besides identifying exact duplicates, which is mostly solved running cryptographic hash functions, it is also necessary to cope with similar inputs (e.g., different versions of files), embedded objects (e.g., a JPG within a Word document), and fragments (e.g., network packets) which is commonly solved by approximate matching. The essential idea is to complement the use of cryptographic hash

* Corresponding author. da/sec – Biometrics and Internet Security Research Group, Hochschule Darmstadt, Haardtring 100, 64295 Darmstadt, Germany.
E-mail addresses: frank.breitinger@cased.de (F. Breitinger), harald.baier@h-da.de (H. Baier), douglas.white@nist.gov (D. White).

[1] https://www.dasec.h-da.de.
[2] http://nsrl.nist.gov.

http://dx.doi.org/10.1016/j.diin.2014.03.001

functions to detect data objects with bytewise identical representation with the capability to find objects with bytewise similar representations.

However, the lookup complexity of similarity digests hamper the usage in the field. Let x denote the amount of digests in a database, then the naive lookup for a single similarity digest has the complexity of $O(x)$. In contrast, cryptographic hash values can utilize binary trees or hash tables and hence the lookup complexity is $O(\log_2(x))$ or $O(1)$, respectively. Assuming a set instead of a single digest, the lookup complexity of similarity digests has a quadratic runtime as it is solved by an all-against-all comparison (brute-force).

Recently, at least one approximate matching algorithm (`ssdeep`) was extended and now has a possibility of indexing (Winter et al., in press). The authors showed an improvement of a factor of almost 2000 which is 'practical speed'. However, analysis showed there are more powerful algorithms like `sdhash` (Roussev, 2011) and `mrsh-v2` (Breitinger et al., 2013b) which output a different type of similarity digest. While `ssdeep` produces a Base64 encoded fingerprint, both other algorithms output Bloom filter based hashes. In spite of all of them, the problem remains.

In this paper we present and evaluate a concept for approximate matching that allows a *file-against-set* comparison with a lookup complexity of $O(1)$ for a single digest. In contrast to general approximate matching, our approach can only answer the question "does this set contain a similar file to *file A*?" by

- **yes**, there is a similar file (but it cannot say which one), or
- **no**, there is no similar file,

which is sufficient in case of blacklisting. We obtain this benefit either at the cost of more hashing operations or requiring a lot of main memory. Our evaluation demonstrates that the current procedures are too slow (e.g., over 21 min to compare 4457 digests of the *t*5-corpus[3] against each other) while our improved version solves this challenge within seconds. Analyzing the precision and recall rates shows that our approach works as reliably as the original implementations.

The rest of the paper is organized as follows. Sec. 2 introduces the necessary background and related work. The problem description and solution overview is explained in Sec. 3. All details about our concept are presented in Sec. 4. The experimental results are given in Sec. 5. Sec. 6 concludes the paper.

Background & related work

This section explains the foundations and presents related literature. First, we briefly present the usage of hash functions and approximate matching in digital forensics which is followed by an introduction of Bloom filters. Sec. 2.3 starts with an overview of approximate matching and then introduces three concepts in more detail.

Hash functions and approximate matching in digital forensics

Currently a popular use case is to employ hashing methods for *known file filtering* of files which is quite simple: an investigator computes the hashes for all files on a target device and compares them to a reference database. Depending on the underlying database, files are either filtered out (e.g., files of the operating system) or filtered in (e.g., known offensive content). Files not found in the database remain unclassified.

In case of filter out (a.k.a. whitelisting) the database contains benign files, e.g., operating system files. We claim that here an investigator is only interested in exact matches and thus crypto hashes are the only choice. However, in case of filter in (a.k.a. blacklisting) the database contains illegal or suspicious inputs, e.g., child abuse or leaked company secrets, and an investigator is also interested in similar files. Note, approximate matching may operate on the byte level or the semantic level (Breitinger et al., 2014).

Bloom filter

Bloom filters (Bloom, 1970) have a wide field of applications, e.g., database applications (Mullin, 1990) or network applications (Broder and Mitzenmacher, 2005) and commonly used to represent elements of a finite set S. A Bloom filter is an array of m bits initially all set to zero. In order to 'insert' an element $s \in S$ into the filter, k independent hash functions are needed where each hash function h outputs a value between 0 and $m-1$. Next, s is hashed by all hash functions h. To insert, the bits at the positions $h_0(s)$, $h_1(s)$,…$h_{k-1}(s)$ of the Bloom filter are set to one.

To answer the question if s' is in S, we compute $h_0(s')$, $h_1(s')$, …$h_{k-1}(s')$ and analyze if the bits at the corresponding positions in the Bloom filter are set to one. If this holds, s' is assumed to be in S, however, we may be wrong as the bits may be set to one by different elements from S. Hence, Bloom filters suffer from a non-trivial false positive rate. Otherwise, if at least one bit is set to zero, we know with certainty that $s' \notin S$. It is obvious that the false negative rate is equal to zero.

In case of uniformly distributed data the probability that a certain bit is set to one during the insertion of an element is $1/m$, i.e., the probability that a bit is still zero is $1-1/m$. After inserting n elements into the Bloom filter, the probability of a given bit position to be one is $1-(1-1/m)^{k\cdot n}$. In order to have a false positive, all k array positions need to be set to one. Hence, the probability p for a false positive is

$$p = \left[1-(1-1/m)^{k\cdot n}\right]^k \approx \left(1-e^{-kn/m}\right)^k. \quad (1)$$

Bytewise approximate matching

Approximate matching is a rather new area and probably had its breakthrough in digital forensics in 2006 with an algorithm called context triggered piecewise hashing

[3] http://roussev.net/t5/t5.html (last accessed Nov. 29th, 2013).

(CTPH) (see Sec. 2.3.1). Since then, a couple of algorithms were presented. As this work focuses on Bloom filter based approaches, we discuss those in Sec. 2.3.2. A complete overview of different algorithms is given by Breitinger et al. (2013a).

Basically approximate matching consists of two separate functions. First, tools run a *feature extraction function* that extracts features or attributes from the input that allow a compressed representation of the original object (the exact proceeding depends on the implementation itself). Second, to compare two similarity digests, a *similarity function* is used that normally outputs a score s which is scaled to $0 \leq s \leq 100$. Despite its range, this value is not necessarily an estimate of percentage commonality between the compared objects but a level of confidence. It is meant to serve as a means to sort and filter the results.

ssdeep and the F2S2 software

The program `ssdeep`, also known as *context triggered piecewise hashing* (CTPH, Kornblum (2006)) may be the origin of approximate matching and is based on the spam detection algorithm from Tridgell (2002–2009). The implementation is open source and available online.[4]

The basic idea is very simple: split an input into chunks, hash each chunk independently and concatenate the chunk hashes to a final similarity digest. In order to split an input into chunks, the algorithm identifies trigger points using a rolling hash (a variation of the Adler-32 function[5]) which considers the current context of seven bytes. Each chunk is then given to the non-cryptographic hash function FNV (Noll, 1994–2012). Instead of using the complete FNV hash, CTPH only takes the least significant 6 bits which is equal to one Base64 character. Thus, two files are similar if they have common chunks.

F2S2 was presented by Winter et al. (in press) and is an extension for `ssdeep` that allows a faster similarity digest comparison. F2S2 initializes a hash table that allows to insert n-grams[6] of the Base64 similarity digest. Each similarity digest is split into its n-grams and the ID to the corresponding file is put into its corresponding hash table bucket. In order to lookup a similarity digest, the queried digest is split into its n-grams. Next, the content of all buckets are correlated in order to receive a set of possible similar files. The final decision is then made by using the `ssdeep` comparison function.

The authors showed an improvement of a factor of almost 2000 which is 'practical speed'. For instance, they decrease the time for verifying 195,186 files against a database with 8,334,077 entries from 364 h to 13 min.

Bloom filter based approaches

This section presents two further prominent approaches that outperform `ssdeep` with respect to precision & recall (Roussev, 2011; Breitinger et al., 2013b, 2013c). In the following, we provide a brief sketch of the feature extraction functions of `sdhash` and `mrsh-v2`, respectively; a detailed description is beyond the scope of this paper. Details about the similarity function and the similarity digests are given at the end of this section.

sdhash. This algorithm was proposed by Roussev (2010) and attempts to pick characteristic features for each object that are unlikely to appear by chance in other objects, which is the result from an empirical study. In the baseline implementation, each feature is hashed with SHA-1 (Gallagher and Director, 1995) and inserted into a Bloom filter (Bloom, 1970) where a feature is a sequence of 64 bytes. The similarity digest of the data object is a sequence of 256-byte Bloom filters each representing approximately 10 KiB of the original data, on average.

Subsequently, a block-aligned version was developed (Roussev, 2012), in which fixed-size blocks (16 KiB by default) are mapped to each 256-byte filter. Although the two versions are compatible (the two versions of the digests can be meaningfully compared) we do not consider the block-aligned version in our study as it requires additional parameters.

mrsh-v2. Breitinger and Baier (2013) propose a new algorithm that is based on `ssdeep` and multi-resolution similarity hashing (Roussev et al., 2007). Equal to `ssdeep`, the algorithm divides an input into chunks using a rolling hash where the estimated blocksize is 160 bytes. Each chunk is then hashed by the 64-bit non-cryptographic hash function FNV-1a (Noll, 1994–2012) and inserted into a Bloom filter where a filter can store up to 160 chunks. Once a Bloom filter reaches its capacity, a new one is created.

Note, in the following we are using the term *feature* as a synonym for chunk.

Similarity digest. The similarity digest is very similar in both cases. To insert a feature-hash into a $m = 2048$ bit Bloom filter (default size for both algorithms), the algorithms take 55 bits of the digest, split them into $k = 5$ sub-hashes of 11 bits and set the corresponding bit. For instance, the sub-hash $00010001100_2 = 8C_{16} = 140_{10}$ sets bit 140 in the Bloom filter. Both implementations have a maximum of features per Bloom filter. If this limit is reached, a new Bloom filter is created. Hence, the final similarity digest is a sequence of Bloom filters which is supposed to be approximately 1.0% (`mrsh-v2`) or 1.6 %–2.6% (`sdhash`) of the input length (compression ratio). To identify the similarity between two digests, all Bloom filters of fingerprint A are compared against all Bloom filters of similarity digest B with respect to the Hamming distance as metric.[7]

Problem & solution

Currently, the problem is that it is not possible to order/index Bloom filter digests. Thus, if a database contains x digests a comparison of a given similarity digest against the database requires an 'against-all' comparison. Extending

[4] http://ssdeep.sourceforge.net (last accessed Nov. 29th, 2013).

[5] http://en.wikipedia.org/wiki/Adler-32 (last accessed Nov. 29th, 2013).

[6] n-grams a fragments of a longer sequence, e.g., 2-g of ABCD are AB, BC and CD.

[7] The original comparison is only sketched in this paper, as we replace it in our new concept.

this scenario means that comparing y similarity digests against the same database corresponds to an all-against-all comparison (bruteforce) which equals a quadratic runtime complexity: $O(xy)$.

Sec. 3.1 gives an overview of the overall idea. After that, we briefly repeat the terminology and definition, as it is really important to have the abbreviations in mind.

Proceeding overview

The basic idea is to insert all features into a single Bloom filter instead of having multiple filters as the lookup complexity per filter is $O(1)$. Thus, we overcome the drawback of existing approaches and avoid the all-against-all comparison. To speed up the comparison decision we additionally replace the classical comparison function by a decision based on a sufficiently large number of common substrings (later called longest run) as explained in Sec. 4.

More precisely, let S_B and S_D be two sets of digests. Traditionally (using cryptographic hash functions) an investigator possesses a database containing the elements of S_B (e.g., the blacklist). When he receives D (e.g., a seized device), he hashes all files to S_D and compares them against S_B. Note, the database S_B can be precomputed and hence its generation time is irrelevant.

Regarding our concept, there are two alternatives depending on the underlying hardware:

1. Alternative one is identical to the traditional procedure. That is, the Bloom filter is filled with the features of S_B in advance, that is we can neglect its generation time. Note, using more than one Bloom filter will slow down the process as they always have to be loaded into memory.
2. The second possibility assumes that S_B does not fit into the Bloom filter, but S_D does. In that case we turn the work flow upside down by filling the Bloom filter with S_D and compare S_B against it.

The difference between these two procedures is the overall time. While in traditional procedure (alternative (1)) only S_D needs to be processed, the second possibility also has to hash S_B as a precomputation step. In the following (1) is denoted by *best-case* and (2) by *worst-case*.

The reason why alternative (2) might be necessary is that it is not possible to load the Bloom filter for S_B into main memory. Hashing all files of a set into a single Bloom filter requires a large Bloom filter which has to fit into main memory due to efficiency reasons. Thus, the limiting source is the physically available RAM.

For instance, let us assume that $|S_B| = 1500$ GiB and $|S_D| = 200$ GiB. As shown later, an everyday working station with 8 GiB RAM cannot handle a Bloom filter of S_B but of S_D. Therefore, we suggest creating a Bloom filter out of S_D and comparing all files of S_B in a second step. It is obvious that both sets have to be hashed – it is not possible to create the database in advance.

To optimize (2), one may store a list of hash values of S_B instead of the files. Thus, the files are already hashed and the overall proceeding is almost as fast as (1). In addition, the compression is better as we only store a 256-bit (32 byte) hash for each 64-byte chunk.

Another downside of this approach is that we can only say: *yes, there is a similar file* but not which file is matched. In contrast, the traditional procedure allows a statement: file A of the seized device matches file B in the database. However, we argue that with respect to filter in (blacklisting) this is sufficient as an investigator has to analyze matches anyway. In short, we only want to know if the investigated file is similar to any file on our set which is perfectly suited for blacklisting. In the case a yes or no decision is insufficient, this procedure can be used as pre-proceeding–if a file is not found in the Bloom filter it is definitely not a black listed file and can be ignored.

Terminology & definition

This section repeats the notations from the previous section which are necessary to understand all improvements and design decisions. Let $m, k, n \in \mathbf{K}$.

feature describes a byte sequence which is hashed and inserted into the Bloom filter. In case of `mrsh-v2` this equals a chunk of approximately 160 bytes and regarding sdhash this is a sequence of exactly 64 bytes.

m denotes the Bloom filter size in bits.

k number of sub-hashes where each one sets a bit in the Bloom filter.

n is the number of features inserted into a Bloom filter.

s denotes the file set size in MiB.

Design decisions and implementation

This section describes the design and code changes of our approach. Sec. 4.1 shows the correlation between the input file size and the number of features which are inserted into the Bloom filter. The relevance of the feature hash function is discussed in Sec. 4.2. Based on all these findings, Sec. 4.3 explains the procedure to calculate the best Bloom filter size. Sec. 4.4 introduces our match decision approach and the false positive rate which is the final parameter of our configuration. After that, we describe the result presentation in Sec. 4.5 and motivate the default configuration in Sec. 4.6. The final part shows some details about our reference implementation.

Correlation between elements (n) and file set size

In Eq. (1), n denotes the number of elements that are inserted into a Bloom filter. Note, the number of elements is different to the amount of files in the set but equal to the number of features. Hence, this section analyzes the relation between n and file set size. Let s denote the file set size in MiB.

sdhash inserts 192 features into a Bloom filter for every approximately 10 KiB of the input file. Thus, n is calculated by $n = s \cdot 2^{20} \cdot 192/(10 \cdot 2^{10}) \approx s \cdot 2^{14}$ where 2^{20} is needed to change from MiB to bytes.

In case of `mrsh-v2`, the implementation splits the input in 160-byte features. Thus, n is calculated by $n = s \cdot 2^{20}/160 \approx s \cdot 2^{13}$ where 2^{20} converts MiB into bytes.

Since we adapted the feature size of `mrsh-v2` in our prototype to 64 bytes, we set $n = s \cdot 2^{14}$ in the following.

Feature hash function

To insert a feature into a Bloom filter of m bits length, k bits are set which requires a hash value of the feature hash function of at least $k \cdot \log_2(m)$ bits. More formally, having a feature hash function of b bits, $b \geq k \cdot \log_2(m)$ which is equivalent to $m \leq 2^{b/k}$.

The default implementations of `sdhash` and `mrsh-v2` run 160-bit SHA-1 and the 64-bit FNV hash, respectively, and set $k = 5$. This comes at a maximum Bloom filter size of $2^{160/5} = 2^{32}$ bits $= 2^{29}$ bytes is 512 MiB (`sdhash`) and $2^{64/5} = 2^{12.8}$ bits $= 891$ bytes (mrsh-v2).

Depending on the file set, this is insufficient as we might have a Bloom filter of several gigabytes (for instance when mapping several terabytes into a Bloom filter). As a consequence, both tools should implement 256-bit versions of the hashing algorithms. For instance, keeping $k = 5$ and using a 256-bit hash function allows us to handle Bloom filter sizes of $2^{256/5} = 2^{51.2}$ bits $\approx 2^{18}$ GiB. Alternatively, if we have an upper limit of the Bloom filter size, we can increase k which reduces the false positive rate (see Sec. 2.2). For instance, assuming a Bloom filter of $m = 8$ GiB $= 2^{36}$ bits, k can be at most $256/36 = 7.11$. For the remainder of this paper, we limit k to $5 \leq k \leq 7$ where 5 is the lower limit to minimize the chance for a false positive. The upper limit is necessary to handle huge amounts data, e.g., 1 TiB. If 7 is too small, one might change the hash function to 512-bit or more.

Defining the Bloom filter size

Traditionally when dealing with Bloom filters, one tries to optimize k for a given n, m, p setting. However, we have limited $5 \leq k \leq 7$ and discussed n. Thus, this section identifies a reasonable Bloom filter size m by transposing Eq. (1) and substituting n in the last step by $s \cdot 2^{14}$ (see Sec. 4.1):

$$\begin{aligned} p &= \left(1 - e^{-kn/m}\right)^k \\ m &= -\frac{k \cdot n}{\ln\left(1 - \sqrt[k]{p}\right)} \\ &= -\frac{k \cdot s \cdot 2^{14}}{\ln\left(1 - \sqrt[k]{p}\right)}. \end{aligned} \tag{2}$$

Note, in our reference implementation, the Bloom filter size has to be a power of two, i.e., $m = 2^c$, $c \in \mathbb{N}$.

Match decision and false positives

In contrast to the classical approximate matching comparison, we are only interested in a binary decision: the currently processed file or a fragment of it is on the blacklist or not. Therefore we adapt the classical comparison function of Bloom filters as follows: a fragment of a given file is assumed to be part of the Bloom filter, if a *sufficiently* large number of subsequent features is found in the filter. We discuss in the following our approach to identify a reasonable interpretation of what 'sufficiently' means.

Eq. (1) and Eq. (2) relate the false positive probability p for a single feature to the different parameters k, n, s, m. In fact, we are less interested in the false positive rate for a single feature but more for a fragment of a whole file (which may consist of hundreds of thousands of features). Thus, we have to extend Eq. (2) to a false positive probability of a fragment.

Let p_f denote the false positive probability for a fragment. If we require $r_{\min} \in \mathbb{N}$ consecutive false positive features to be a false positive fragment, the false positive probability for a fragment is $p_f = p^{r_{\min}}$.

Regarding Eq. (2), we can substitute p by $\sqrt[r_{\min}]{p_f}$ and obtain

$$m = -\frac{k \cdot s \cdot 2^{14}}{\ln\left(1 - \sqrt[k \cdot r_{\min}]{p_f}\right)}. \tag{3}$$

Result presentation

Instead of printing a similarity score between two files, our algorithm outputs.

```
file1.ppt: 163 of 2518 (longest run: 111)
```

which means that `file1.ppt` consists of 2518 features in total where 163 match the underlying Bloom filter. The longest run are 111 features which means that the algorithm identified 111 features in a row (which is larger then $r_{\min}$ and therefore a match).

Sample setting

In the following we briefly discuss the default values. n cannot be influenced as it is defined by the set size. In case of the false positive rate it is obvious that the smaller the better. Since we do not expect more than 1 million files on a device, we set $p_f = 10^{-6}$ k is fixed between 5 and 7 and we decided for 5 as the smaller k the better the runtime (feature hash length can be reduced; see Sec. 4.7.1). $r_{\min}$ is by default 6.

For instance, let us assume $s = 200$ GiB $= 25 \cdot 2^{13}$ MiB of data, a false positive rate per file of $p_f = 10^{-6}$, $k = 5$ and $r = 6$:

$$\begin{aligned} m &= -\frac{k \cdot s \cdot 2^{14}}{\ln\left(1 - \sqrt[k \cdot r]{p_f}\right)} = -\frac{5 \cdot (25 \cdot 2^{13}) \cdot 2^{14}}{\ln\left(1 - 10^{-\frac{6}{30}}\right)} = -\frac{125 \cdot 2^{27}}{-0.9968} \\ &= 125.40 \cdot 2^{27} \approx 2^{34} \text{ bits} \end{aligned}$$

Thus, our procedure requires a Bloom filter size of 2 GiB. Note, using the default setting, the Bloom filter size in megabytes m_{mb} can be estimated by $m_{mb} \approx s/100$.

Implementation details

To verify our findings, we released a tool called `mrsh-net` which is basically a modification of the latest `mrsh-v2` version. It can be downloaded from our website for tests.[8]

[8] http://wp1187348.server-he.de/z_downloads/tool.zip; anonymous for review.

The implementation is very simple and only has two options:

- **g** generates the database and prints it to stdout. Usage: `./mrsh-net -g t5-corpus > dbFile`
- **i** reads DB-FILE dbFile and compares DIR/FILE against it. Usage: `./mrsh-net -i dbFile t5-corpus`

The final step is to compile it by running `make mrsh-net`.

Feature hash function

The main change was the implementation of the FNV-1a 256 bit function which only consists of an XOR and the multiplication with the prime $2^{168} + 2^8 + 0 \times 63$. As the runtime efficiency is very important, the implementation of the multiplication is 'hardcoded', i.e., it is not trivial to change the prime number or extend it to 512 bit.

In order to speed up the implementation, one may manipulate the FNV implementation in `src/fnv.c`. The function `mulWithPrime2` is responsible for the multiplication with the 256-bit prime. However, in case of a small Bloom filter, we do not need the most significant bits and can remove them. For instance, setting the Bloom filter to 32 MiB and $k = 5$, we only need $\log_2(32 \cdot 2^{20} \cdot 8) \cdot 5 = 140$ bits. Thus, we can comment out lines 108–112, which then ignores the bits 160–255.

Settings

To adapt our prototype for a specific use case, the user can change the following configuration in header/`config.h`:

`SUBHASHES` – amount of sub-hashes, parameter k (default: 5).

`MIN_RUN` – minimal longest run, parameter r (default: 6).

`BF_SIZE_IN_BYTES` – Bloom filter size in bytes (default: $33 \cdot 554 \cdot 432 = 2^{25} = 32$ MiB). It has to be a power of 2.

There are more settings available. However, this is ongoing research and we therefore do not recommend changing them at this time.

Experimental results & assessment

This chapter mainly consists of three parts. First, we analyze the general efficiency of different approaches. The second part relates `mrsh-net` with the longest common substring. The final part compares `mrsh-net` and `mrsh-v2` with each other.

All the presented results are based on the *t*5-corpus[9] (Roussev, 2011), which contains 4457 files with a total size of 1.78 GiB. The average file is ≈400 KiB and the file type distribution is given in Table 2.

For our testing, we used the default configuration of `mrsh-net`, with $k = 5$,$r_{\min} = 6$ and a Bloom filter size of 32 MiB. The blocksize, i.e., the approximate length of a feature, is set to 64 bytes.

Efficiency in general

Let S_D denote the hashes of files from a device and let S_B denote database set (i.e., the blacklist). Traditionally the proceeding requires to hash all files in S_D and to compare the hashes against an 'existing database' of S_B files. Thus, this section focuses the general properties of the different approaches with respect to runtime efficiency and database size (compression).

The results are given in Table 1 whereby the details are discussed in the upcoming subsections. First, the compression (row 1) is analyzed in Sec. 5.1.1. Next, Sec. 5.1.2 explains the runtime of the algorithms (rows 2–4). The last section is an estimation for a large scale scenario to clarify the impact of non-indexing.

Columns 1 and 2 present the results for the original implementations of `sdhash` and `mrsh-v2`, respectively. In column 3 we show the results for the worst case which means that we do not have an underlying database (see Sec. 3.1). The following column also presents the worst case but we modified `mrsh-net` based on the defaults in Sec. 4.7.2 (less bits of the FNV hash are considered). For completeness we included the results for F2S2 and SHA-1 in the last two columns.

Database size

Let S_B be the *t*5-corpus. Then, this section shows the size of the corresponding database. In case `sdhash`, `mrsh-v2`, F2S2 and SHA-1 the database is trivial as the database is equal to the hashes.

Regarding `mrsh-net` there are two possibilities: worst vs. best case. The worst case describes the scenario where the database does not fit in RAM and hence a hash-database as such does not exist. The investigator needs to have the whole dataset available. In contrast, for the best case where sufficient RAM is available, the database is simply the Bloom filter.

To conclude, the size of the databases of `sdhash`, `mrsh-v2` and `mrsh-net` (best) are in the same order of magnitude and therefore only a weak assessment criterion.

Experimental runtime efficiency

This section focuses on the time of hashing S_D and comparing it against the database of S_B (generating the database is neglected as it can be done in advance). As we are interested in the runtime only, we use *t*5-corpus as both S_B and S_D. Note, this results in 4457 × 4457 comparisons.[10]

The times are given in Table 1. Row 2 states that all algorithms perform well in hashing but are still slow compared to SHA-1. The problem is shown in row 3 where both Bloom filter approaches need an extremely long time for the all-against-all comparison. The last row only sums rows 2 and 3. Note, the 'worst-columns' constitute an exception. Admittedly the comparison takes less then a second, however there is no underlying database and thus

[9] http://roussev.net/t5/.

[10] We run the tools by `./tool -c D B` which compares both lists also there are duplicate comparisons, i.e., A against B and B against A.

Table 1
Database size and runtime efficiency of different algorithms.

	sdhash	mrsh-v2	mrsh-net worst	mrsh-net worst	mrsh-net best	F2S2	SHA-1
Database size	61.18 MiB	27.33 MiB	1.78 GiB	1.78 GiB	32 MiB	3.69 MiB	0.24 MiB
Hashing	178 s	53 s	123 s	77 s	123 s	221 s	24 s
Comparing	1281 s	1259 s	<1 s[a]	<1 s[a]	<1 s	<1 s	<1 s
Total	1459 s	1312 s	246 s	154 s	123 s	221 s	24 s

[a] The comparison itself need less than a second, however, in worst case B needs to be hashed.

Table 2
Number of files per file type: *t*5-corpus.

jpg	gif	doc	xls	ppt	html	pdf	txt
362	67	533	250	368	1093	1073	711

S_B has to be processed. Therefore, row 4 contains two times the hashing time (as $D = B$).

One should keep in mind that the comparison has quadratic complexity and thus increases enormously when the number of files increases. In contrast, our new concept has a linear runtime as it only needs to hash the files.

Impact on runtime in large scale forensics

Based on the findings from before, this section estimates the efficiency for a real life scenario. More precisely, we used the numbers from Table 1 and calculated the upcoming ones for a larger use case where we pick up the example from Sec. 4.6 assuming 200 GiB of seized data and a database of 1500 GiB[11].

The results are given in Table 3. The estimated database size is given in row 1 where in the best case mrsh-net needs the less space, however, it should be kept in RAM. Row 2 calculates the approximate hashing time by multiplying 200/1.78 (we have to process 200 GiB instead of 1.78 GiB). The last row assesses the comparison time. For instance, sdhash needed 1281 s for comparing $1.78 \times 1.78 = 3.17$ GiB of data. As this sample requires to compare $200 \times 1500 = 300{,}000$ GiB of data, we estimate the overall time by $300{,}000/3.17 \cdot 1281$ s.

Again, there two possible scenarios with mrsh-net. In the worst case, we hash the 200 GiB to the Bloom filter and then process the 1500 GiB. As the comparison 'costs nothing', mrsh-net has to hash 1700 GiB which is $(1700/1.78 \cdot 123)$ s = 117471 s (approx 32 h). In the best case we have a powerful station that can hold the Bloom filter for 1500 GiB data in RAM (approximately 16 GiB of RAM are needed). Thus, we only need to process the 200 GiB which takes 227 min.

Precision & recall on base of the longest common substring

The current version of mrsh-net decides between match and non-match based on the longest run. Hence, this section focuses on the relation between mrsh-net and the longest common substring. Due to the complexity, we build our ground truth on the approximate longest common substring which is briefly described in the upcoming subsection. Based on this assumed ground truth, we evaluate mrsh-net in Sec. 5.2.2.

Table 3
Estimated runtime for a sample use case.

	sdhash	mrsh-v2	mrsh-net worst	mrsh-net best
Database size	49.79 GiB	22.22 GiB	1500 GiB	16 GiB
Hashing	329 min	98 min	227 min	227 min
Comparing	3.84 years	3.77 years	32.63 h	<1 min

Approximate longest common substring

The basic idea of the approximate longest common substring metric (aLCS) is not to compare files byte by byte but rather block by block. To identify the blocks, we utilize the rolling hash from ssdeep and aim at having a block size $bs \approx 80$ byte. If we set the blocks size smaller than 80, the runtime efficiency decrease enormously (Note, a block size of 1 equals the tradition longest common substring). Instead of comparing blocks bytewise, each one is hashed and compared using the 64-bit FNV-1a hash Noll (1994–2012). Besides the hash value, the entropy and length for each block is stored in the final linear list called *aLCS-digest*.[12]

The output of the aLCS tool is a list which we denote as ground truth. It contains both file names, the longest common substring and the entropy for this sequence, e.g.,

```
file1 | file2 | 993 | 5.56
file2 | file3 | 11945 | 0.5
```

For instance, the first line says that file1 and file2 have an aLCS score of 993 bytes with an entropy of 5.56. The second line shows a special case with a very low entropy which could be an indicator that both files share mostly zeros.

Precision & recall rates

To calculate the rates, we perform an all-against-all comparison but neglect self-comparisons.[13] Thus, we use the following simplified notation:

mrsn(f,*BF*) compares file f against the Bloom filter *BF* and returns the longest run.

aLCS(f,*GT*) returns the longest aLCS score for f in the ground truth *GT*.

[11] The current NSRL of NIST contains about 2 TiB of unique data, hence this a realistic size.

[12] Note, this is ongoing research and currently in review. In the future we will publish some more details about the evaluation and show that this is a valid approximation.

[13] Compare a file against itself will result in a perfect match.

According to this, we define the true positive (TP), false positive (FP), true negative (TN) and false negative (FN) as follows:

TP: $mrsn(f, BF) \geq r_{\min}$ and $aLCS(f, GT) \geq r_{\min} \cdot bs$.
FP: $mrsn(f, BF) \geq r_{\min}$ and $aLCS(f, GT) < r_{\min} \cdot bs$.
TN: $mrsn(f, BF) < r_{\min}$ and $aLCS(f, GT) < r_{\min} \cdot bs$.
FN: $mrsn(f, BF) < r_{\min}$ and $aLCS(f, GT) \geq r_{\min} \cdot bs$.
where $r_{\min} \cdot bs = 6 \cdot 64 = 384$ bytes.

Positives. Our comparison returned 2555 positive matches with a true positive rate of 99.3% and a false positive of 0.7%. Reviewing the false positives, all but one of the longest run *lr* do not exceed 9 which means that they are very close to our threshold.

In addition, we studied the distribution of the aLCS scores d_{aLCS} relative to $r_{\min} \cdot bs$ for the false positives where

$$d_{aLCS} = \left[100 \times \left(1 - \frac{aLCS(f, GT)}{384}\right)\right], d_{aLCS} \in \mathbb{N}.$$

This shows how close the false positives are to the threshold of 384. The results are given in Table 4. For instance, over 60% have an aLCS score above 30% (=269 bytes). To sum it up, although these are false positive, they are close to the thresholds.

Next, we consider the relation between the longest run and the aLCS score. In other words, we expect that the longest run *lr* multiplied by the blocksize *bs* is greater or equal the aLCS score, i.e., $lr \cdot bs \geq aLCS$. According to that, we adapt the configuration from the beginning of this section and changed $r_{\min} \cdot bs$ to $lr \cdot bs$. Thus, the new true positive setting is:

TP: $mrsn(f, BF) \geq r_{\min}$ and $aLCS(f, GT) \geq lr \cdot bs$.
…

In this case, the detection rates worsen and fall down to a true positive rate of 92.3% and a false positive rate of 7.7%. Again, we consider the distribution for the aLCS scores in Table 5. As we can see, over 75% vary by less or equal then 30% and we rate these results as still acceptable.

Negatives. Obviously the negatives are $4457 - 2555 = 1902$ which can be broken down into 77.1% true negatives and 22.9% false negatives. Having a closer look at this very high false negatives, we observe that most aLCS matches are based on low entropy sequences. In other words, the high aLCS scores between some files are based on long runs of zeros only, i.e., the entropy of the substring is $e = 0$, or runs of with a lot of zeros, e.g., the entropy of the substring is $0 < e < 3$. Thus, Table 6 shows the impact of considering aLCS sequences with a higher entropy.

Nevertheless, false negatives are not so much relevant. For instance, with respect to blacklisting, these files remain unclassified and an investigator has to analyze them manually. Hence, false negatives are considered during a further investigation.

Table 4
Empirical probability distribution function (*pdf*) and cumulative distribution function (*cdf*) for d_{aLCS}.

X	10	20	30	50	70
$P\{d_{aLCS} = X\}$	0.1111	0.2778	0.2222	0.1111	0.0556
$P\{d_{aLCS} \leq X\}$	0.1111	0.3889	0.6111	0.8889	0.9444

Table 5
Empirical *pdf* & *cdf* for d_{aLCS} for the relation between longest run and aLCS score.

X	10	30	50	70	100
$P\{d_{aLCS} = X\}$	0.3214	0.2296	0.0714	0.0051	0.0051
$P\{d_{aLCS} \leq X\}$	0.3214	0.7551	0.9235	0.9796	1.0000

Precision & recall rates compared to mrsh-v2

This section compares the relation between `mrsh-v2` and `mrsh-net`. As both are based on the same procedure, we expect that both implementations yield similar results. In other words, comparing a file *f* against database *BF*, both algorithms should either output a match or a non-match. Thus, we define the following rates:

TP: $mrsn(f, BF) \geq r_{\min}$ and $mrsh(f, BF) \geq 1$.
FP: $mrsn(f, BF) \geq r_{\min}$ and $mrsh(f, BF) = 0$.
TN: $mrsn(f, BF) < r_{\min}$ and $mrsh(f, BF) = 0$.
FN: $mrsn(f, BF) < r_{\min}$ and $mrsh(f, BF) \geq 1$.

Positives

Regarding the 2555 positive matches from `mrsh-net`, 92.1% are true positives and also identified by `mrsh-v2`. The false positive rate is therefore at 7.9%. Comparing the false positives against the aLCS showed that in fact only 3.6% (out of the 7.9%) are really false positive. On the other side, Table 7 shows the distribution of the longest run for the false positives. Most of them are close to the longest run threshold 8.

To conclude, the results are slightly different, however, the `mrsh-net` shows a finer granularity as in fact these are not false positives but true positives.

Negatives

The negatives yield a 61.8% true negative rate and a 38.2% false negative rate. Recall, false negative means that `mrsh-net` does not identify a match while `mrsh-v2` outputs a score greater 0. In other words, `mrsh-v2` identifies a positive.

Thus, we first compared the `mrsh-v2` results against aLCS. In fact, almost 7 0% percent of these matches are based on files that share less than 384 bytes which have no

Table 6
Distribution of false negative with respect to entropy.

Entropy	>0	>1	>2	>3
TN	78.5%	82.3%	86.4%	91.2%
FN	21.5%	17.7%	13.6%	8.8%

Table 7
Empirical *pdf* & *cdf* for longest run *lr*.

X	10	15	20	30	50
$P\{lr = X\}$	0.1095	0.0200	0.0200	0.0200	0.0050
$P\{lr \leq X\}$	0.4925	0.7363	0.7960	0.9665	0.9950

false negatives for `mrsh-net` (our setting aims at having more than 384 bytes). Regarding the remaining 30%, most of the matches are again based on a low entropy, e.g., over 75% have $e < 3$.

To conclude, the algorithms do not coincide very much with respect to negatives.

Conclusion

We have presented and evaluated a new approach to efficiently decide about the similar membership of a file to a given dataset and hence solve an important issue in the context of approximate matching. Our approach decreases the lookup complexity from $O(x)$ to $O(1)$, where x is the number of files in the reference dataset. We released a sample implementation for a practical evaluation. For the well-known $t5$-corpus as evaluation file data set we were able to solve the similar membership problem for all files in the order of seconds (rather than minutes). The drawback is that we are only able to decide about membership, but not about the similarity to a certain file, which is sufficient for the important use case of blacklisting.

References

Bloom BH. Space/time trade-offs in hash coding with allowable errors. Commun ACM 1970;13:422–6.

Breitinger F, Baier H. Similarity preserving hashing: eligible properties and a new algorithm mrsh-v2. In: Rogers M, Seigfried-Spellar K, editors. Digital forensics and cyber crime. Lecture notes of the institute for computer sciences, social informatics and telecommunications engineering, vol. 114. Berlin Heidelberg: Springer; 2013. pp. 167–82.

Breitinger F, Guttman B, McCarrin M, Roussev V. Approximate matching: definition and terminology. NIST Publication; 2014.

Breitinger F, Liu H, Winter C, Baier H, Rybalchenko A, Steinebach M. Towards a process model for hash functions in digital forensics. In: 5th International Conference on Digital Forensics & Cyber Crime; 2013a.

Breitinger F, Stivaktakis G, Baier H. FRASH: a framework to test algorithms of similarity hashing. In: 13th Digital Forensics Research Conference (DFRWS'13); 2013b. Monterey.

Breitinger F, Stivaktakis G, Roussev V. Evaluating detection error trade-offs for bytewise approximate matching algorithms. In: 5th ICST Conference on Digital Forensics & Cyber Crime (ICDF2C); 2013c.

Broder A, Mitzenmacher M. Network applications of bloom filters: a survey. Internet Math 2005;1:485–509.

Gallagher P, Director A. Secure Hash Standard (SHS). Technical Report National Institute of Standards and Technologies, Federal Information Processing Standards Publication; 1995. pp. 180–1.

Kornblum J. Identifying almost identical files using context triggered piecewise hashing. Digit Investig 2006;3:91–7.

Mullin J. Optimal semijoins for distributed database systems. IEEE Trans Softw Eng 1990;16:558–60.

NIST Information Technology Laboratory. National software reference library http://www.nsrl.nist.gov; 2013.

Noll LC. Fnv hash. http://www.isthe.com/chongo/tech/comp/fnv/index.html; 1994–2012.

Roussev V. Data fingerprinting with similarity digests. In: Chow K-P, Shenoi S, editors. Advances in digital forensics VI. IFIP advances in information and communication technology, vol. 337. Berlin Heidelberg: Springer; 2010. pp. 207–26.

Roussev V. An evaluation of forensic similarity hashes. Digit Investig 2011;8:34–41.

Roussev V. Managing terabyte-scale investigations with similarity digests. In: Peterson G, Shenoi S, editors. Advances in digital forensics VIII. IFIP advances in information and communication technology, vol. 383. Berlin Heidelberg: Springer; 2012. pp. 19–34.

Roussev V, Richard III GG, Marziale L. Multi-resolution similarity hashing. Digit Investig 2007;4:105–13.

Tridgell A. spamsum. http://www.samba.org/ftp/unpacked/junkcode/spamsum/; 2002–2009. Accessed 29.11.13.

Winter C, Schneider M, Yannikos Y. F2s2: fast forensic similarity search through indexing piecewise hash signatures. Digit Investig 2013;10(4):361–71.