2014

An Efficient Similarity Digests Database Lookup -- a Logarithmic Divide and Conquer Approach

Frank Breitinger
University of New Haven, fbreitinger@newhaven.edu

Christian Rathgeb
Hochschule Darmstadt, Germany

Harald Baier
Hochschule Darmstadt, Germany

Publisher Citation

Breitinger, F. , Rathgeb, C., and Baier, H. (2014) An efficient similarity digests database lookup -- a logarithmic divide and conquer approach. Journal of Digital Forensics, Security and Law 9(2): 152-166.

AN EFFICIENT SIMILARITY DIGESTS DATABASE LOOKUP – A LOGARITHMIC DIVIDE & CONQUER APPROACH

Frank Breitinger, Christian Rathgeb and Harald Baier
da/sec - Biometrics and Internet Security Research Group
Hochschule Darmstadt, Darmstadt, Germany
{Frank.Breitinger, Chrisitian.Rathgeb, Harald.Baier}@cased.de

ABSTRACT

Investigating seized devices within digital forensics represents a challenging task due to the increasing amount of data. Common procedures utilize automated file identification, which reduces the amount of data an investigator has to examine manually. In the past years the research field of approximate matching arises to detect similar data. However, if n denotes the number of similarity digests in a database, then the lookup for a single similarity digest is of complexity of $O(n)$.
This paper presents a concept to extend existing approximate matching algorithms, which reduces the lookup complexity from $O(n)$ to $O(\log(n))$. Our proposed approach is based on the well-known divide and conquer paradigm and builds a Bloom filter-based tree data structure in order to enable an efficient lookup of similarity digests. Further, it is demonstrated that the presented technique is highly scalable operating a trade-off between storage requirements and computational efficiency. We perform a theoretical assessment based on recently published results and reasonable magnitudes of input data, and show that the complexity reduction achieved by the proposed technique yields a 2^{20}-fold acceleration of look-up costs.

Keywords: digital forensics, hashing, approximate matching, Bloom filter, mrsh-v2, sdhash, indexing

1. 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 forensic task is *known file filtering* which is rather trivial: (1) compute the hashes for all files on a target device and (2) compare them to a reference database. Depending on the underlying database, files are either filtered out (i.e., whitelisting, e.g., files of the operating system) or filtered in (i.e., blacklisting, e.g., known offensive content). In case of whitelisting we claim that an investigator is only interested in exact matches and thus cryptographic hashes are sufficient. A very common database for 'filter out' data is the National Software Reference Library (NSRL, 2013) maintained by the National Institute for Standards and Technologies (NIST). However, in case of 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.

However, the lookup complexity of similarity digests hamper their usage in the field. Let n denote the number of digests in a database, then the naive lookup for a single similarity digest is of complexity of $O(n)$. In contrast, cryptographic hash values can utilize tree data structures or hash tables which reduce the lookup complexity to $O(\log(n))$ or $O(1)$, respectively.

Recently, the community started to work on that challenge and presented some first ideas. For instance, `ssdeep` (Kornblum, 2006) was extended and now has a possibility of indexing (Winter, Schneider, & Yannikos, 2013). The authors showed an improvement of a factor of almost 2000 which is 'practical speed'. However, there are further approaches like `sdhash` or `mrsh-v2` that outperform `ssdeep` with respect to precision & recall (Roussev, 2011; Breitinger, Stivaktakis, & Baier, 2013; Breitinger, Stivaktakis, & Roussev, 2013).

As the indexing procedure of `ssdeep` is not applicable for the latter approaches[1], Breitinger et. al presented and evaluated a lookup improvement that allows a *file-against-set* comparison in case of a Bloom filter similarity digest. The lookup complexity of decreases to $O(1)$, however, their work only provides a binary decision (if or if not a file is in the blacklist). Breitinger et. al published a prototype and demonstrated that their concept works (Breitinger, Baier, & White, 2014).

In this paper we focus on an improvement of the proposed concept in (Breitinger et al., 2014). The main idea is to apply the well-known divide & conquer paradigm to their concept to find the actual similar file in the blacklist. Hence instead of providing a binary decision, our approach returns the actual matching file(s). Compared to an all-against-all comparison, we only have a lookup complexity of $O(\log(n))$ for a single digest. Our theoretical assessment shows that for a reasonable size of blacklist and seized device the comparison step of our approach is 2^{20} times faster than in the classical Bloom filter setting.

The remainder of the paper is organized as follows. Sec. 2 introduces the necessary background and related work. Our efficient digest comparison concept based on a Bloom filter hierarchy is explained in Sec. 3. All details about our concept are presented in Sec. 4 and Sec. 5. An assessment of the proposed concept is given in Sec. 6. The last section, Sec. 7, concludes the paper.

[1]Note, different approximate matching algorithms work on different similarity digest representations. While `ssdeep` outputs a Base64 encoded fingerprint, `sdhash` and `mrsh-v2` make use of Bloom filter based hashes.

2. BACKGROUND

This section explains the foundations and summarizes related literature. First we define the usage of hierarchical tree data structures in Sec. 2.1. Subsequently, Sec. 2.2 introduces the concept of Bloom filters. Finally, an overview of approximate matching is given in Sec. 2.3, introducing two concepts in more detail.

2.1 Hierarchical Tree Data Structure

A tree data structure $\mathbf{\Psi}$ of degree x is a tree where each node either has x references to nodes (its 'children') or does not have children (leaves). Let $|\mathbf{\Psi}|$ define the total number of leaves in $\mathbf{\Psi}$, then the height, i.e., the number of nodes on the longest path from the root to a leaf is

$$h(\mathbf{\Psi}) \quad = \quad \lceil \log_x(|\mathbf{\Psi}|) \rceil, \qquad (1)$$

where $\mathbf{\Psi}$ has exactly $h(\mathbf{\Psi})$ levels. According to this, the height of a non-empty tree $\mathbf{\Psi}$ is defined as $h(\mathbf{\Psi}) + 1$.

2.2 Bloom Filter

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

To answer the question if s' is in $\mathbf{S}$, we compute $h_0(s'), h_1(s'), \ldots, h_{k-1}(s')$ and analyze if the bits at corresponding positions in the Bloom filter are set to one. If this holds, s' is assumed to be in $\mathbf{S}$, however, these bits may be set to one by different elements previously inserted to $\mathbf{S}$. 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 $s' \notin \mathbf{S}$, i.e., 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 z elements into the Bloom filter, the probability of a given bit position to be one is $1 - (1 - 1/m)^{k \cdot z}$. To have a false positive, all k array positions need to be set to one. Hence, the probability p for a false positive is

$$\begin{aligned} p &= \left(1 - (1 - 1/m)^{kz}\right)^k \\ &\approx \left(1 - \mathrm{e}^{-kz/m}\right)^k \quad \text{for} \quad \frac{1}{m} \ll 1. \end{aligned} \tag{2}$$

If we fix m and z in Eq. (2), we can determine k such that the false positive probability is minimized. Rewriting Eq. (2) as $p(k) = e^{k \cdot \ln\left(1 - e^{-kz/m}\right)}$ yields one distinct candidate of a root of $\frac{\mathrm{d}p}{\mathrm{d}k}$ at

$$k = m/z \cdot \ln(2). \tag{3}$$

However, due to $p(0) = \lim_{k \to \infty} p(k) = 1$ and $0 < p(k) < 1$ for $k > 0$ shows that $p(k)$ actually is minimal at this value of k. In our case we have to choose an integer close to the value of Eq. (3).

Note, the lookup complexity for a feature is independent of the Bloom filter size if the filter is in the memory. We simply ask if the bit at a specific memory position is one.

2.3 Bytewise Approximate Matching

Approximate matching is a rather new area in digital forensics and probably had its breakthrough in 2006 with an algorithm called context triggered piecewise hashing (CTPH). Since then, a couple of algorithms were presented. As this work focuses on Bloom filter-based approaches, we briefly describe them in the following. A comprehensive overview of different algorithms is given by (Breitinger, Liu, et al., 2013).

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). This compressed representation is called a *similarity digest* or *similarity hash*. Second, to compare two similarity digests, a *comparison function* is used that normally outputs a score between 0 and 100.

2.3.1 Bloom Filter-Based Approaches

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 & Director, 1995) and mapped to a Bloom filter 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 & Baier, 2013) propose a new algorithm that is based on `ssdeep` and multi-resolution similarity hashing (Roussev, III, & Marziale, 2007). Similarly 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 mapped to 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 feature-hash, split them into $k = 5$ sub-hashes of 11 bits and set the corresponding bit.

Both implementations have a maximum amount of feature-hashes 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[2].

3. EFFICIENT COMPARISON USING A BLOOM FILTER HIERARCHY

The main drawback of Bloom filter-based similarity digests is that it is not possible to order / index them. More precisely, looking for a single digest in a database containing n digests, comes with an 'against-all' comparison (brute-force) and, thus, a complexity of $O(n)$.

For instance, we run `sdhash` on our private laptop and work on the t5-corpus[3] which comprises 4457 files. In a first step we precompute the similarity digests of all files and store them in a database of roughly 62 MiB. Then for each file we compare its similarity hash against the database. The overall run time for the 4457 comparisons is 1281 s. In other words, comparing $1.78 \cdot 2^{30} \times 1.78 \cdot 2^{30} = 3.17 \cdot 2^{60}$ of data (in units of bytes) takes about 20 minutes. According to this, we can estimate the runtime for a practical relevant scenario: we assume a database file set of 256 GiB and a seized hard drive with 200 GiB of data. Then we have to compare $200 \cdot 2^{30} \times 256 \cdot 2^{30} = 51,200 \cdot 2^{60}$ of data units, resulting in an unpractical response time of $^{51,200}/_{3.17} \cdot 1281\,\text{s} \approx 20,689,968\,\text{s} \approx 240\,\text{days}$. In contrast, for cryptographic hashes the complexity is $O(\log_2(n))$ or even $O(1)$ depending on the storage technique: binary tree data structures or hash tables, respectively.

In the following we present our approach to speed up the comparison step using a tree of Bloom filters. We start in Sec. 3.1 by introducing the terminology used in this paper. Our idea extends previous work of (Breitinger et al., 2014), which we describe in Sec. 3.2. Then Sec. 3.3 gives an overview of the basic operation mode of the proposed concept.

3.1 Terminology & Definitions

This section explains the notation and terminology used in this paper.

n... number of files which is equal to the number of similarity digests.

z... number of features inserted into a Bloom filter (The correlation between n and z is discussed in Sec. 4.1).

feature... byte sequence which is hashed and inserted into the Bloom filter.

m... Bloom filter size in bits.

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

p... false positive probability for an element / feature to be in the Bloom filter.

x... degree of the tree.

In case of `mrsh-v2` *feature* equals a chunk of approximately 160 bytes and regarding `sdhash` *feature* is a sequence of exactly 64 bytes.

3.2 Starting Point

In (Breitinger et al., 2014) the authors presented a first step towards a solution of the described problem. Instead of using multiple Bloom filters, the authors inserted all features of the database into a single Bloom filter. Thus, they overcome

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

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

the drawback of existing approaches and avoid the all-against-all comparison of similarity digests.

In other words, if **S** denotes a database of n digests, the paper shows that it is feasible to perform a membership query f in $O(1)$ yielding a binary answer to the question "does the set **S** contain a similar file to *file A*?", that is

$$f(A, \mathbf{S}) \mapsto \{0, 1\}, \tag{4}$$

where 0 means 'no' and 1 'yes'. In case of a positive result, **S** contains a file or fragment similar to A, but we do not know which one. In case of blacklisting this provides the hint that one will find evidence on the target device.

In their paper, the authors discussed different parameters and validated their results. Therefore, they published a tool called `mrsh-net` and showed a comprehensive evaluation that this approach works well. Since we change the parameter values and argumentation for different settings, we discuss our settings in Sec. 4.

3.3 Proceeding Overview

Our idea is based on the divide-and-conquer paradigm. The basic idea is, that if f is applicable to a set of n files it can be applied to any subset of **S** which contains $\ll n$ files, too. This implication enables the tracing of potential similar files, where we recursively divide a given set of similarity digests into x subsets, that is we build a hierarchical tree data structure yielding a logarithmic lookup complexity.

A tree data structure of Bloom filters is built over all similarity digests where each leaf is a 'file identifier' (*FI*). An *FI* is a link to a database which contains at least the similarity digest, but may contain additional information, too. Additionally, an *FI* points at an 'file identifier counter' (*FIC*). An *FIC* is initialized with zero and incremented if a tree traversal ends in the corresponding *FI*.

In the following we describe the tree generation process and the lookup strategy.

3.3.1 Tree Generation

For a given set **S** containing n elements (i.e., files), each element $s \in \mathbf{S}$ is mapped to the root node of the tree; a huge Bloom filter. Mapping means that the approximate matching algorithm selects the features, builds the feature hashes and sets the corresponding bits in the Bloom filter. In all, we map z features into the root Bloom filter.

Subsequently, depending on the degree x of the tree data structure, **S** is divided in x subsets each containing n/x consecutive elements of **S**. For each of the x subsets we generate a child node of the root Bloom filter by mapping the corresponding files of the subset to its Bloom filter. This procedure is applied recursively, i.e., in level L, n/x^{L-1} consecutive elements are mapped to x^{L-1} different Bloom filters, while $n/x^{L-1} > 1$. Finally, *FI*s and *FIC*s are stored at the corresponding leaves. The procedure is summarized in Fig. 1, an example for a binary tree ($x = 2$) is illustrated in Fig. 2.

Input:

- Set of elements, **S**

Output:

- Hierarchical tree data structure, Ψ

Procedure:

1. Insert each element $s \in \mathbf{S}$ into the root Bloom filter **b**
2. In level L insert x^{L-1} consecutive sequences of z/x^{L-1} elements into Bloom filters of according x^{L-1} nodes
3. Repeat 2. while $z/x^{L-1} > 1$
4. Insert *FI*s at leaves of Ψ

Figure 1 Generalized Construction of Bloom Ffilter-Based Tree Data Structure

3.3.2 Lookup Strategy

Once the Bloom filter tree is generated, it is not necessary to compare a digest of a seized device against all digests in the reference dataset; it only has to be compared against a subset of nodes. Moreover, we claim that most comparisons yield a non-match (i.e., $|good\ files| \gg |bad\ files|$ as we focus on blacklisting) and hence are dropped in the first step. Once a match is found in the root node, we trace a path down the tree data structure ending at a leaf and the according *FIC* is incremented. If an exact match score is

needed, the *FI* can be used to fetch the similarity digest from the database. We point to the fact that we expect to perform $x/2$ member queries at each level as we only process child nodes of the matching Bloom filter of the previous level. Each query is of lookup complexity $O(1)$. This improves the lookup complexity for one digest to $O(x \cdot \log_x(n))$, that is $O(\log(n))$ for a binary tree.

The final lookup procedure is different since we do not compare files to the nodes but multiple fragments of files. The impact and final workflow is discussed in Sec. 5.

4. DESIGN DECISIONS

This section explains our design decisions for the various parameters to implement our concept. Sec. 4.1 shows the correlation between the input file size and the number of features which are inserted into the Bloom filter. The size and height of the binary tree is discussed in Sec. 4.2. Sec. 4.3 introduces our match decision approach and the false positive rate. Based on all these findings, Sec. 4.4 explains the procedure of how to calculate the best Bloom filter size. The relevance of the feature hash function is discussed in Sec. 4.5.

4.1 Correlation Between Features (z) and Files (n)

In Eq. (2), z denotes the number of elements that are inserted into a Bloom filter which is equal to the amount of features. Thus, the number of elements differs to the amount of files n in the set. Hence, this section analyzes the relation between z and file set size. Let μ denote the file set size in MiB.

- `sdhash` maps 192 features into a Bloom filter for every approximately 10 KiB of the input file. Thus, z is calculated by $z = \mu \cdot 2^{20} \cdot 192/(10 \cdot 2^{10}) \approx \mu \cdot 2^{14}$, where 2^{20} and 2^{10} is needed to change from MiB and KiB to bytes, respectively.
- In case of `mrsh-v2`, the implementation splits the input on average in 160-byte features. Thus, z is calculated by $z = \mu \cdot 2^{20}/160 \approx \mu \cdot 2^{13}$ where 2^{20} converts MiB into bytes.

For the remainder of this paper we use $z = \mu \cdot 2^{14}$.

4.2 Memory Usage and Height of the Tree

Eq. (1) in Sec. 2.1 showed how to calculate the height of the tree if every leaf only contains one *FI* and one *FIC*. If we assume a dataset size of Ω GiB and an average file size of ω KiB, then our dataset consists of $|\mathbf{\Psi}| := {}^{\Omega \cdot 2^{30}}/_{(\omega \cdot 2^{10})} = {}^{\Omega}/_{\omega} \cdot 2^{20}$ files which results in a maximum height of $h(\mathbf{\Psi}) = \lceil \log_x({}^{\Omega}/_{\omega} \cdot 2^{20}) \rceil$ -- if each bucket contains exactly one entry.

As this is very memory consuming, there is also the possibility that a leaf is a bucket and contains multiple *FI*s and corresponding *FIC*s instead of a single one. In that case the queried similarity digest must be compared to all files in the bucket. However, by storing a total number of l *FI*s and l *FIC*s at each leaf, we reduce the height to,

$$h(\mathbf{\Psi}) = \lceil \log_x(|\mathbf{\Psi}|/l) \rceil,$$

The actual size of the each Bloom filter (each node) is variable. However, if we require that k, p, z are fix, then the relation of the sizes of Bloom filters between two consecutive levels is ${}^1/_x$. More precisely, let $L \in \mathbb{N}$ denote the level in the binary tree ($L = 1$ is the root level) and let m_L be the Bloom filter size at the corresponding level. Then, $m_{L+1} = {}^{m_L}/_x$. On a subsequent level we have an x-fold number of Bloom filters yielding a constant size of the sum of filters per level. Accordingly, the overall memory usage of the tree is $h(\mathbf{\Psi}) \cdot m_1$.

Remark: we reallocated the elements but the overall number is constant at each level. If the number of elements mapped to the tree reduces and k, p are fix, then the size of the filter reduces, too.

4.3 Match Decision and False Positives

Traditionally approximate matching compares two similarity digests against each other and returns the match score. Our procedure works

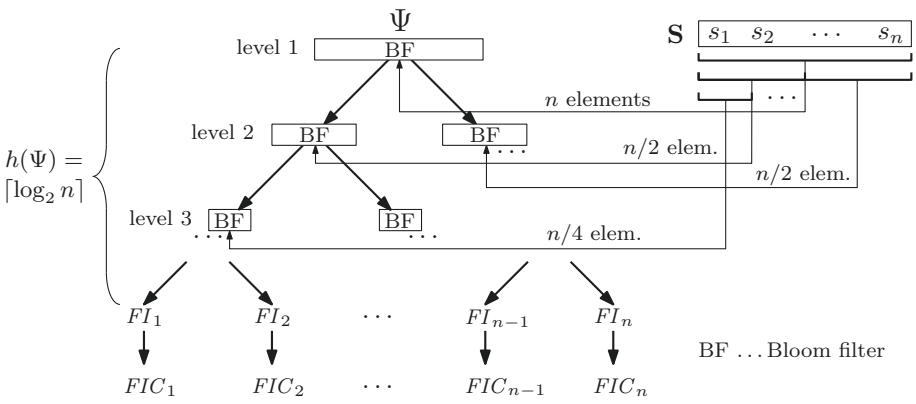

Figure 2 Construction of Bloom Filter-Based Tree Data Structure Using a Binary Tree

a bit different. Instead of searching for the complete files, we focus on fragments to identify potential buckets. A fragment of a file matches a Bloom filter, if r consecutive features are found in the node. Once a leaf is identified, we might perform the conventional comparison.

More precisely, Eq. (2) relates the false positive probability p for a single feature to the different parameters k, z, 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. Let p_f denote the desired false positive probability for a fragment. If we require $r \in \mathbb{N}$ consecutive false positive features to be a false positive fragment, the false positive probability for a fragment is $p_f = p^r$.

For instance, if we choose a targeted false positive probability of 10^{-8} for each fragment (which is equal to a file) and set $r = 8$, then $p = 10^{8/-8} = 0.1$ [4]. The predefined threshold r is scalable and defines the number of required tree traversals. Since r represents an absolute value, i.e., the lookup complexity remains $O(r \cdot \log(n)) = O(\log(n))$. The entire matching procedureis depicted in Fig. 3.

Note, we only focus on binary similarity. In other words, researchers have found that common artifacts such as color profiles are very common throughout multiple files. This might lead to a false positive from a semantic point of view, however, this might also be interesting for an investigator (e.g., these pictures where taken with the same camera). Depending on the requirements, it might be necessary to ignore (filter out) those artifacts which is ongoing research.

Input:

- Bloom filter **b**, file A, threshold r

Output:

- Binary match decision 1...match, 0...no match

Procedure:

1. Identify and hash feature of A
2. Add it to the similarity digest
3. Compare the feature to **b**
4. If it matches, then increase counter FIC and save feature hash, otherwise $FIC = 0$
5. If $FIC = r$, return '1', otherwise apply 1. to the next feature
6. Return '0'

Figure 3 Generalized Overview of the Proposed Matching Function

4.4 Root Bloom Filter Size

Let z and p be given and k be the optimal value from Eq. (3). Then Eq. (2) becomes

$$p = \left(1 - \mathrm{e}^{-\ln(2)}\right)^{m/z \cdot \ln(2)} = 2^{-m/z \cdot \ln(2)} = e^{-m/z \cdot (\ln(2))^2} .$$

Thus the root Bloom filter size is estimated as

$$m_1 = -z \cdot {}^{\ln p}/_{(\ln 2)^2} , \qquad (5)$$

[4] Note, $r = 8$ corresponds an approximate overlapping of $8 \cdot 160 = 1280$ bytes in case of `mrsh-v2`. We argue that this is a sufficient lower bound to talk about similarity. Of course, one may change these settings to the personal requirements.

where z denotes the amount of all features in a set. According to the findings from Sec. 4.3, $p = \sqrt[r]{p_f}$. If we use $r = 8$ and the false positive probability for a fragment $p_f = 10^{-8}$ [5]. Then, we have $p = 10^{8/-8} = 0.1$ and $m_1 = -z \cdot \ln 0.1/(\ln 2)^2 = -\mu \cdot 2^{14} \cdot (-4.7925) = \mu \cdot 2^{16.26}$ bits. Depending on μ, each level of the binary tree could easily have Bloom filters of 1-2 GiB (the size of the Bloom filter has to be of type 2^a where $a \in \mathbb{N}$).

4.5 Feature hash function

As known from Eq. (3) if z and m are given, the value k minimizing the false positive probability is

$$k = m/z \cdot \ln 2. \tag{6}$$

Note, k is independent from the considered level as m and z are both multiplied by $1/x$.

According to this, the best value is $k = \mu \cdot 2^{16.26}/\mu \cdot 2^{14} \cdot \ln 2 = 2^{1.73} = 3.34$. Since $k \in \mathbb{N}$ and it may vary depending on the further parameters, we suppose $3 \leq k \leq 7$.

Generally speaking, to set k bits in a Bloom filter of m bits length, it requires a feature hash function of at least $k \cdot \log_2(m)$ bits. More formally, having a feature hash function of b bits, $b \geq k \cdot \log_2(m)$. In order to allow also larger reference sets like 256 GiB or 512 GiB which needs Bloom filters of 4 or 8 GiB, we recommend to implement 256-bit versions of the feature hash functions (the default implementations of `sdhash` and `mrsh-v2` run 160-bit SHA-1 and the 64-bit FNV hash). For instance, set $k = 5$ 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.

4.6 Analysis of Subsequent Level (for Binary Trees)

If we consider a binary tree and a fragment of file A is found in the root node, we expect to have a match in one of the child nodes, too. We therefore jump to the Bloom filters on level 2 and start with the left node. If the fragment matches there, too, we continue on the next level. In case of a negative, there are two possibilities:

1. *Hard decision*: we trust that this fragment is not a false positive and jump to the next level of the right node.

2. *Soft decision*: we verify the result by comparing the fragment against the right node.

In the former case we obtain a logarithmic lookup efficiency where in the latter case we approximately have an additional 50% comparison overhead. However, we reduce the false positive rate. The entire procedure is depicted in Fig. 4.

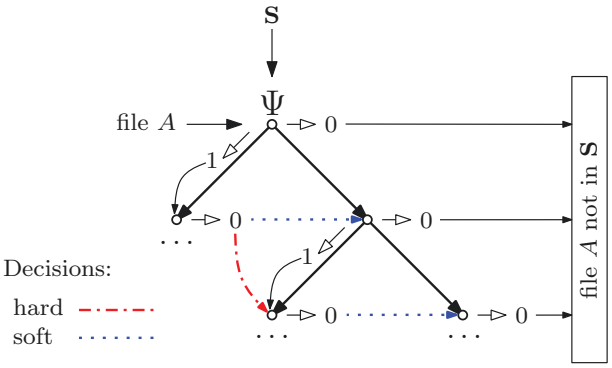

Figure 4 Generalized Overview of the Proposed Workflow

5. FINAL WORKFLOW

Traditionally the reference data is hashed and stored in a database. Next, for the comparison, all files on a seized device are processed and the similarity digests are compared against all entries in the database. As a result an investigator receives all files that are similar to the processed ones. File A, for example, could have several similar files in the database and the result for A is a list of similar files.

Due to the usage of a tree and fragments (i.e., r consecutive features) this overall procedure changes. We argue that the main scenario for approximate matching is blacklisting, i.e., filter in data. With respect to filter out data, we are more interested in exact duplicates and hence cryptographic hash functions should be used. Moreover, in case of filter in data, we claim that it is sufficient to identify one match -- an

[5] Note, the actual false positive rate will be higher due common artifacts.

investigator only wants to know if the file is blacklisted or not.

Thus, when comparing file A against our Bloom filter hierarchy, we process all fragments of a given file. If the fragment is not found in a Level, we neglect it. If we identify a leaf that contains this fragment, we increase FIC. At the end, we return the FI with the highest FIC. Depending on the personal preference, one may fetch the real similarity digest from the database based on the FI(s) and perform the conventional comparison.

Note, the lookup complexity for a fragment is $O(r \cdot k)$ and thus comes to $O(1)$.

In order to speed up the process, one may only compare some features and thus, one can stop when one fragment of A is found (that is r consecutive features). The difference is that this latter procedure the probability of a false positive is higher as we only use one instead of all fragments.

5.1 Insertion & Deletion of File Identifiers

Within any tree data structure the insertion of an FI can be handled in $O(\log(n))$ steps by inserting according features to $\log(n)$ Bloom filters and one FI at the resulting leaf. To construct $\mathbf{\Psi}$ we recommend to start from the top and go to the bottom as we can reuse the feature hashes from higher levels.

In contrast, the deletion of an FI can not be performed on an existing tree data structure. Since features are hashed into Bloom filters in an irreversible manner, i.e., we do not know which 1s result from which features and the number of times these occur, deletion has to be performed off-line. In order to delete a single FI from the proposed Bloom filter-based tree data structure $\mathbf{\Psi}$, the according FI has to be deleted from the original set $\mathbf{S}$ and $\mathbf{\Psi}$ has to be constructed from scratch. However, we claim that blacklisted files will always remain blacklisted and therefore deleting an FI never happens.

6. ASSESSMENT

As we are interested in a functional evaluation, the actual type of the data (blacklisted vs. whitelisted) does not matter. To get an overview of the average file size of large datasets, we analyzed the sizes of almost 1,000,000 files in the `govdoc`-corpus[6].

The average file size is 494 KiB with a distribution as shown in Table 1--nearly 91% of all files are smaller than 1 MiB. Additionally, we checked the average file size on our working stations which is 510 KiB and 611 KiB, respectively.

Table 1 File Sizes Distribution in the `Govdoc`-Corpus (Min Size is 1 KiB)

File size (KiB)	≤ 4	≤ 16	≤ 64	≤ 256	≤ 1024
Amount (%)	5.40	20.71	52.54	75.82	90.60

As shown later, this technique is very memory consuming and requires powerful hardware. However, we argue that nowadays hardware is not expensive any more and companies working in prosecution can afford servers with hundreds of gigabytes of RAM, e.g., according to amazon.com, 16 GiB RAM cost less than 150 dollar which results in 2400 dollar for 256 GiB of RAM.

6.1 Any Tree vs. Binary Tree

The size and the height of a tree depend on their degree x. For instance, if we assume a database $\mathbf{S}$ with 10,000 entries, then for choosing a binary tree, $x = 2$, we get a total number of $h(\mathbf{\Psi}) = \lceil \log_2(10,000) \rceil = 14$ levels (level 15 are the leaves). In the worst case, the complexity of looking up a single digest reduces from 10,000 comparisons to 15. On the other hand, if the digest is not found in the root node, it is dropped immediately.

In case there is not enough memory to handle a binary tree of 14 Bloom filter levels, one may set an upper limit. Thus, the leaves do not contain a single FI and FIC but a bucket

[6] "These documents were obtained by performing searches for words randomly chosen from the Unix dictionary, numbers randomly chosen between 1 and 1 million, and randomized combinations of the two, for documents of specified file types that resided on web servers in the .gov domain using the Yahoo and Google search engines"--`http://digitalcorpora.org/corpora/files` (last accessed May 2nd, 2014).

with multiple *FI*s and corresponding *FIC*s, e.g., setting an upper limit of $L = 10$ levels results in $\lceil 10,000/x^L \rceil = \lceil 10,000/2^{10} \rceil \approx 10$ *FI*s per bucket. In addition, it is possible to increase x, e.g., setting $x = 4$ reduces the number of levels to $\lceil \log_4(10,000) \rceil - 1 = 6$.

6.2 Sample Setup

We decided for underlying dataset of 256 GiB, an average file size of 512 KiB as a sample setup. Recall, we apply approximate matching in the context of blacklisting. According to law enforcement investigators[7] a typical size of a blacklist used by them is far smaller than 256 GiB. We restrict to focus our analysis on binary trees where leaves correspond to single *FI*s and *FIC*s.

According to our discussion in Sec. 4, the number of files is $^{256 \cdot 2^{30}}/_{(512 \cdot 2^{10})} = 2^{19} = 524,288$. Thus, the binary tree has $\log_2(524,288) = 19$ levels and the size of the Bloom filter at the root m_1 is 2 GiB. Overall, the tree requires $19 \cdot 2 = 38$ GiB of memory.

6.3 Memory Requirement

The required amount of memory depends on the amount of data to be mapped to the tree data structure $\mathbf{\Psi}$ and the number of files, n (see Sec. 3.3). Based on the overall size of the data and the configuration of $\mathbf{\Psi}$, we can define the size and number of required Bloom filters. The number of files defines the height of the tree, $h(\mathbf{\Psi})$. For the considered setting, i.e., the application of a binary tree data structure where each leaf points at a single *FI*, diverse sample configurations are summarized in Table 2.

As already mentioned, any restriction to memory can be handled by utilizing buckets of *FI*s and *FIC*s at leaves or extending the degree x of the tree data structure. Both of these actions will reduce the amount of required Bloom filters and, hence, the overall memory requirement.

6.4 Comparison Efficiency

This subsection compares the efficiency of the traditional approach and the new one. We decided to focus on the bit comparisons of each proceeding. We make use of the sample setup to simulate the (blacklist) database and assume a seized HDD with 200 GiB and an average file size is 512 KiB. In what follows, we show that our approach speeds up the comparison at least by a factor of 2^{20}.

6.4.1 Traditional Lookup

In Sec. 2.3.1 we briefly explained that a similarity digest may consist of several 'small' Bloom filters (each 256 bytes). In order to compare two digests, all filters of digest A have to be compared against all filters of digest B. Consequently, doing an all-against-all comparison of similarity digests equals an all-against-all comparison of all Bloom filters and thus of all bits. Hence, we need to know how many Bloom filters of 256 bytes exist.

`sdhash` roughly compresses 10 KiB of the input into a 256 bytes Bloom filter. Therefore, the 256 GiB reference dataset results in $256 \cdot 2^{20}/10 = 2^{24.68}$ filters and the analyzed data corresponds to $200 \cdot 2^{20}/10 = 2^{24.32}$ filters. To sum it up, we have to compare around $2^{49.00}$ Bloom filters each consisting of 2^{11} bits (256 bytes). Hence in all we have to compare 2^{60} bits in total.

`mrsh-v2` approximately compresses 25 KiB (blocksize of 160) of the input into a 256 bytes Bloom filter. Thus, we have $256 \cdot 2^{20}/25 = 2^{23.36}$ filters for the reference dataset and $200 \cdot 2^{20}/25 = 2^{23.00}$ for the HDD. Overall, it requires to compare $2^{46.36}$ Bloom filters or $2^{57.36}$ bits.

6.4.2 Proposed Approach

Sec. 4.1 correlated the input size and the amount of elements or features by $z = \mu \cdot 2^{14}$. This means that 200 GiB of data equal $z = 200 \cdot 2^{10} \cdot 2^{14} = 2^{31.64}$ features. If we use $k = 5$, then each feature requires to compare 5 bits: $2^{31.64} \cdot 5 = 2^{33.96}$ bits. If we perform an 'all fragments comparison' where all of these exist in the database (worst case), these features have to be compared to all $19 = 2^{4.25}$ levels which comes to a total of $2^{38.21}$ bits that have to be compared.

In the best case (no fragment is found), we only require $2^{(31.64+33.96)/2} = 2^{32.80}$ bit compar-

[7] Communicated unofficially via personal contact.

Table 2 Correlation Between Dataset Size and Average File Size. First value is $h(\mathbf{\Psi})$, Second Value is Total Size of the Binary Tree in GiB. m_1 Denotes the Size of the Root Node in GiB

Avg. file size	**Data size**				
	128 GiB $m_1 = 1$	256 GiB $m_1 = 2$	512 GiB $m_1 = 4$	1024 GiB $m_1 = 8$	2048 GiB $m_1 = 16$
32 KiB	22 / 22	23 / 46	24 / 96	25 / 200	26 / 416
64 KiB	21 / 21	22 / 44	23 / 92	24 / 192	25 / 400
128 KiB	20 / 20	21 / 42	22 / 88	23 / 184	24 / 384
256 KiB	19 / 19	20 / 40	21 / 84	22 / 176	23 / 368
512 KiB	18 / 18	19 / 38	20 / 80	21 / 168	22 / 352
1024 KiB	17 / 17	18 / 36	19 / 76	20 / 160	21 / 336

isons (i.e., in average we compare 2.5 bits out of the 5 to the Bloom filter). Additionally, the similarity digest comparison is more expensive as after comparing two similarity digests, the amount of 1s needs to be counted. If we assume a uniform distribution of matches (which is rather unlikely as blacklisted files will occur less frequent), we have to compare $^{2^{38.21}-2^{33.96}}/_{2} = 2^{37.13}$ in the average case.

7. CONCLUSION

In this paper we discussed the challenge of looking up Bloom filter-based similarity digests which is currently the main drawback of these approaches. Our concept decreases the lookup complexity from $O(n)$ to $O(\log_x(n))$ where n is the number of files in the reference database and x is the degree of the tree. This enhancement is obtained to the cost of hardware--the physical memory has to be large. For instance, a dataset having 256 GiB with an averaged file size of 512 KiB required 38 GiB of memory. In the design section we discussed multiple parameters which allow to adjust this concept to hardware requirements and to the cost of computation. Our theoretical assessment showed that the proposed approach is superior compared to the existing 'against-all' comparison.

In a next step we like to implement a prototype to validate our assumptions and to compare it against the straight forward approach of storing fragment hashes. That is, we store the cryptographic hash of the chunks in the database directly including an index of the original file.

ACKNOWLEDGEMENTS

This work was partly funded by the EU (integrated project FIDELITY, grant number 284862) and supported by CASED (Center for Advanced Security Research Darmstadt).

REFERENCES

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, *13*, 422--426.

Breitinger, F., & Baier, H. (2013). Similarity preserving hashing: Eligible properties and a new algorithm mrsh-v2. In M. Rogers & K. Seigfried-Spellar (Eds.), *Digital forensics and cyber crime* (Vol. 114, pp. 167--182). Springer Berlin Heidelberg. Retrieved from `http://dx.doi.org/10.1007/978-3-642-39891-9_11` doi: 10.1007/978-3-642-39891-9_11

Breitinger, F., Baier, H., & White, D. (2014). On the database lookup problem of approximate matching. *1st Digital Forensics Research Conference EU (DFRWS-EU'14)*.

Breitinger, F., Liu, H., Winter, C., Baier, H., Rybalchenko, A., & Steinebach, M. (2013, Sept). Towards a process model for hash

functions in digital forensics. *5th International Conference on Digital Forensics & Cyber Crime*.

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

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

Broder, A., & Mitzenmacher, M. (2005). Network Applications of Bloom Filters: A Survey. *Internet Mathematics*, *1*(4), 485--509.

Gallagher, P., & Director, A. (1995). *Secure Hash Standard (SHS)* (Tech. Rep.). National Institute of Standards and Technologies, Federal Information Processing Standards Publication 180-1.

Kornblum, J. (2006, September). Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, *3*, 91--97. Retrieved from `http://dx.doi.org/10.1016/j.diin.2006.06.015` doi: 10.1016/j.diin.2006.06.015

Mullin, J. (1990, may). Optimal semijoins for distributed database systems. *IEEE Transactions on Software Engineering*, *16*(5), 558 -560. doi: 10.1109/32.52778

NIST Information Technology Laboratory. (2013). *National Software Reference Library.* (`http://www.nsrl.nist.gov`)

Noll, L. C. (1994--2012). *Fnv hash.* `http://www.isthe.com/chongo/tech/comp/fnv/index.html`.

Roussev, V. (2010). Data fingerprinting with similarity digests. In K.-P. Chow & S. Shenoi (Eds.), *Advances in digital forensics vi* (Vol. 337, pp. 207--226). Springer Berlin Heidelberg. Retrieved from `http://dx.doi.org/10.1007/978-3-642-15506-2_15` doi: 10.1007/978-3-642-15506-2_15

Roussev, V. (2011, August). An evaluation of forensic similarity hashes. *Digital Investigation*, *8*, 34--41. Retrieved from `http://dx.doi.org/10.1016/j.diin.2011.05.005` doi: 10.1016/j.diin.2011.05.005

Roussev, V. (2012). Managing terabyte-scale investigations with similarity digests. In G. Peterson & S. Shenoi (Eds.), *Advances in digital forensics viii* (Vol. 383, pp. 19--34). Springer Berlin Heidelberg. Retrieved from `http://dx.doi.org/10.1007/978-3-642-33962-2_2` doi: 10.1007/978-3-642-33962-2_2

Roussev, V., III, G. G. R., & Marziale, L. (2007, September). Multi-resolution similarity hashing. *Digital Investigation*, *4*, 105--113. doi: 10.1016/j.diin.2007.06.011

Winter, C., Schneider, M., & Yannikos, Y. (2013). F2s2: Fast forensic similarity search through indexing piecewise hashsignatures. *Digital Investigation*, *0*, -. (Article in Press - no journal issue assigned by now) doi: http://dx.doi.org/10.1016/j.diin.2013.08.003

