Fast Indexing Strategies for Robust Image Hashes

By

Christian Winter, Martin Steinebach and York Yannikos

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2014 EU

Amsterdam, NL (May 7th - 9th)

Digital Investigation 11 (2014) S27–S35

Contents lists available at ScienceDirect

Digital Investigation

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

Fast indexing strategies for robust image hashes

Christian Winter*, Martin Steinebach, York Yannikos

Fraunhofer Institute for Secure Information Technology SIT, Rheinstr. 75, 64295 Darmstadt, Germany[1]

Keywords:
Robust hashing
ForBild
Indexing
Vantage point tree
Locality-sensitive hashing

A B S T R A C T

Similarity preserving hashing can aid forensic investigations by providing means to recognize known content and modified versions of known content. However, this raises the need for efficient indexing strategies which support the similarity search. We present and evaluate two indexing strategies for robust image hashes created by the ForBild tool. These strategies are based on generic indexing approaches for Hamming spaces, i.e. spaces of bit vectors equipped with the Hamming distance. Our first strategy uses a vantage point tree, and the second strategy uses locality-sensitive hashing (LSH). Although the calculation of Hamming distances is inexpensive and hence challenging for indexing strategies, we improve the speed for identifying similar items by a factor of about 30 with the tree-based index, and a factor of more than 100 with the LSH index. While the tree-based index retrieves all approximate matches, the speed of LSH is paid with a small rate of false negatives.

Introduction

The forensic research community has created various tools for similarity search over the past decade. All these tools follow a two-step approach for identifying pairs of similar files: First they calculate short digests ("hashes") of the files, and then they compare the digests for similarity. Hence the hash function must map similar files to similar digests, and there must be a similarity function for the digests.

In the domain of multimedia data, the forensic research has adapted methodologies developed for multimedia retrieval and other multimedia applications. In particular, our *ForBild* tool for robust image hashing (Steinebach, 2012; Steinebach et al., 2012) has been developed based on the evaluation of different perceptual hashing methods (Zauner et al., 2011). The algorithm employed in ForBild is an improved version of *block mean value based hashing* (Yang et al., 2006). The choice of this algorithm is justified with its hash calculation speed and its low error rates. These properties are important requirements for forensic applications with huge amounts of data.

While the hash algorithm of the ForBild tool is based on the evaluation of various approaches, the search algorithm has not been considered by now. The ForBild tool searches for similar hashes in a naive way by comparing each query hash to each hash in the reference database. Although the hash comparison uses the Hamming distance, which can be calculated very efficiently, the naive brute force search requires a significant amount of time for databases with hundred thousands or even millions of images. Suitable indexing strategies should perform much better than brute force by restricting the search to a subset of the reference hashes for each query. Due to the computationally cheap Hamming distance only very effective (size of subset) and efficient (time needed for subset selection) indexes will be faster than a brute force search.

This paper presents two suitable indexing strategies we identified during the analysis of various approaches. These strategies can be applied to ForBild hashes as well as any

* Corresponding author. Tel.: +49 6151 869 259; fax: +49 6151 869 224.
E-mail addresses: christian.winter@sit.fraunhofer.de (C. Winter), martin.steinebach@sit.fraunhofer.de (M. Steinebach), york.yannikos@sit.fraunhofer.de (Y. Yannikos).
[1] http://www.sit.fraunhofer.de.

http://dx.doi.org/10.1016/j.diin.2014.03.004

other type of block mean value based hashes. Moreover, these strategies should also be suitable for other types of similarity hashes which are compared with the Hamming distance. Our first indexing strategy is based on a metric tree and presented in Sect. Tree-based index, and the second strategy is an LSH approach presented in Sect. LSH-based index. Evaluation results of these strategies are contained in Sect. Evaluation.

Block mean value based hashing

Block mean value based (BMB) hashing divides an image into a fixed number of blocks and calculates one hash bit for each block. ForBild uses $16 \times 16 = 256$ blocks; Yang et al. (2006) do not specify the number of blocks used in their work.

The hash bits are calculated according to the following procedure:

1. Convert the image to grayscale, i.e. remove the color information and retain the brightness information.[2]
2. Calculate the mean brightness of each block. This is an intuitive approach for scaling the image into the grid of blocks. The result is a tiny grayscale version of the image, which has one pixel per block. We call this result the intermediate hash of the image.
3. Determine the median value of the previously calculated mean values.
4. Set the final hash bit for each block according to whether its mean value is above the median or not. Hence the hash is a tiny bi-tonal version of the original image. For most images (very simple graphics are an exception) the hash has no visually recognizable content.

ForBild made two improvements for this approach: It calculates a separate median for each quadrant of the image to increase the hash collision resistance, and it has an automatic flipping mechanism to produce hashes robust against mirroring. Additionally, it inherits the robustness against image scaling (even non-proportional scaling), lossy compression, Gaussian filtering, noise adding, gamma correction, color adjustments, etc. from the original approach. These robustness properties and a low collision rate make it well-suited for identifying modified versions of known images. In the forensic domain blacklisting of child sexual abuse images is an obvious application for the ForBild tool.

Match decision

In order to check a given hash against a database of reference hashes (e.g. a blacklist), the Hamming distance is employed, which counts the number of non-matching bits. Hashes of versions of the same image have mostly identical bits while hashes of unrelated images should share on average half of the bits (128 in the case of ForBild) by chance. The procedure of selecting the closest hash from a reference list reduces the average Hamming distance to 62 for images unknown to the database (Steinebach, 2012). At a first glance, this is an unexpectedly low distance. A naive calculation under the assumption of independent and identically distributed (i. i. d.) bits implies that the distance of unrelated images should be above hundred with very high probability. However, the assumption of i.i.d. bits is not satisfied because neighboring bits of BMB hashes are strongly correlated. Consequently, the distribution of distance values is wider than expected, and hence the average of the best distance is lower than expected. This observation is important for our optimization in Sect. Choice of vantage points.

ForBild declares hashes with Hamming distance of at most 8 as good match. A distance above 8 and below 33 indicates a potential match. Such potential matches are reexamined by calculating a mismatch penalty, originally called “weighted distance” (Steinebach et al., 2012, Sect. 2.3) and credited to a “quantum hash” method developed by Jin and Yoo (2009). The calculation of the mismatch penalty requires as additional input the intermediate hash of one of the images.[3] Each non-matching bit of the two hashes is penalized based on the heuristic that a hash bit is less stable if the according intermediate value is closer to the median. The penalty for the mismatch of an unsteady bit (i.e. small difference between intermediate value and median) is small while the penalty for the mismatch of a reliable bit (i.e. large difference) is high. If the mismatch penalty falls below a threshold, the potential match is declared as match.

Query performance

The time needed for checking a query hash against a reference list obviously depends on the size of this list. As ForBild performs a naive linear search through the list, the required time is linear in the number of reference hashes. We evaluated the running time of the original ForBild tool using our workstation, and we measured an average time of about 5.0 ms for checking one pre-calculated hash against our reference list containing approximately 130,000 hashes (see Sect. Evaluation for details about the experimental environment). The hash calculation required on average 46 ms for an image from the reference image collection.[4] Thus the hash comparison needs about 10% of the total time in the present setting.

Advanced ForBild variants

While the ForBild hash is robust against many image operations, the underlying BMB approach does not

[2] None of the existing papers specifies a conversion method. Hence the retained “brightness” might be for example the luma, (gamma compressed) relative luminance, or perceptual lightness. ForBild actually uses luma.

[3] Hence the penalty function is asymmetric, which does not comply with the term “distance”.

[4] Our initial evaluation of the ForBild tool resulted in an average hashing time of 12.5 ms because a different image set with smaller average image size was used (Steinebach et al., 2012, Sect. 3.2). The figures presented by Breitinger et al. (2013, Table 2) conform to a linear dependency between image size and hashing time.

withstand image cropping. To counter this, we introduced the concept of hashing subsections of images (Steinebach et al., 2013). We use a face detection algorithm and further divide the faces through blob segmentation. Finally, the bounding boxes of these blobs are hashed. This approach is based on face detection because faces are a crucial element for the investigation of child pornographic images, which is the main goal of ForBild. The number of hashes to be processed per image increases significantly as multiple blob hashes are created for each face in an image. The matching algorithm searches for the image with the highest number of similar blob hashes.

A different advancement of ForBild is video hashing. Various approaches for robust video hashing (e.g. Oostveen et al., 2001) are available today, but most of them use inter-frame features like the change of brightness over time. Contrary to such techniques, frame-based video hashing facilitates the matching of images against videos. This requires that all video frames have a suitable reference in the hash database. As differences between adjacent frames are mostly small, only roughly 10% of all hashes need to be stored. A video with 30 frames per second and a duration of 60 min has 108,000 frames. Due to the similarity-based reduction, about 10,000 hashes must be stored.

The need for efficient matching

The last section shows that applications can produce huge numbers of hashes – especially when combining both ideas, so that cropped video frames can be identified. A 60 min video with an average number of one face on screen requires about 100,000 blob hashes to be stored in the hash database. For 1000 videos, we end up with roughly 100 M hashes. As each hash has a size of 256 bits = 32 bytes, the database needs about 3 GiB of memory.

While the memory requirement is not a challenge for common computers today, the database lookup time will become the central issue in robust hashing. Extrapolating the search time observed in Sect. Query performance to 100 M hashes yields approximately 3.8 s for checking a single hash. Therefore the following sections introduce more efficient approaches beyond the brute force search discussed above.

Tree-based index

Trees are widely used data structures which come in many different flavors for a large range of applications. Structures like AVL trees and B-trees are used to mange items which expose a total order. Spatial data can be handled for example with k-d trees and R-trees. But these trees are only suitable for low-dimensional data because their performance drops with increasing dimension. Hence these structures cannot be used for BMB hashes, which are high-dimensional binary vectors. Searching neighbors in high-dimensional spaces is commonly associated with the "curse of dimensionality" because it is challenging to perform this task efficiently.

A promising class of tree structures for high-dimensional data can be labeled as *vantage point trees* (vp-trees). Such trees just rely on the existence of a distance function for the data. The first variant has been proposed by Burkhard and Keller (1973, "File Structure 1") without giving a name to it. Uhlmann (1991) defined a type of metric tree he called "ball decomposition". The same structure has been introduced by Yianilos (1993) as vantage point tree. Baeza-Yates et al. (1994) defined another variant as *fixed-queries trees* (FQ-trees). The commonality of all variants is the usage of so-called vantage points for constructing and querying the tree. Hence we generalize the term "vantage point tree" to the name of the whole class of tree structures.

A vp-tree is constructed in a top-down manner. After choosing a vantage point for the root node, the distance between each data point and the vantage point is calculated. Each child of the root node receives a subset of the datapoints corresponding to a certain range of distance values. This procedure is applied recursively to the children until a cancel criterion is reached, e.g. a desired depth or at most one data point per node.

The processing of similarity queries against the tree exploits the triangle inequality: If the query point has distance d_1 from a vantage point and a data point distance d_2 from this vantage point, then the distance between query point and data point is at least $|d_1 - d_2|$.

Hence the query is processed in the following way: Initially, the distance between the query point and the vantage point of the root node is calculated. For each child of the root there is a lower bound for the distance between query point and data points due to the argument above and the range of distance values associated with the child node. This lower bound is 0 for the child under which the query point would be stored, and it grows when moving to more distant children. Consequently, the child nodes can be prioritized according to their individual lower bound, and some children may be dropped completely in case of a thresholded similarity search. The search strategy first follows the child with highest priority, and processes it the same way as the root node. This procedure finds those leaf nodes first which contain the most promising candidates among the data points. For each candidate the distance to the query point is calculated. The closest neighbor of the query point is identified as soon as the best candidate is closer than the remaining unprocessed nodes.

The various variants of vp-trees differ e.g. in the choice of vantage points and the fanout of the tree structure. The following sections provide variants found in the literature and describe our choices.

Choice of vantage points

The most basic strategy selects one of the data points under the current node at random. However, some vantage points may be more favorable than others. Good vantage points lead to a wider distribution of distance values, and thus to a more effective tree (Yianilos, 1993). Moreover, the vantage points are not restricted to the set of data points, and in fact, some variants of vp-trees admit any point from the underlying metric space as vantage point. An FQ-tree has the special property that it uses the same vantage point for all nodes on the same level. This reduces the number of distance calculations while traversing the tree.

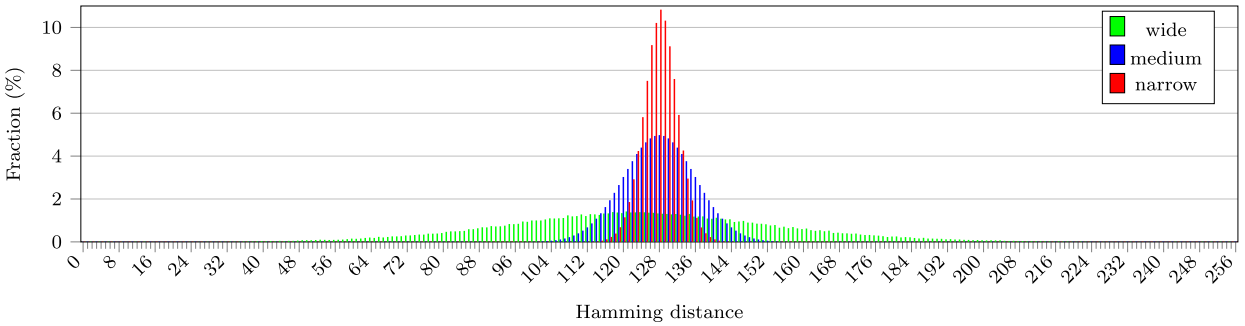

Fig. 1. Distance distribution for various vantage points. The widest distribution corresponds to our best vantage point, the medium wide distribution to a vantage point with randomly generated i.i.d. bits (this distribution would be expected for any vantage point if the hash bits were i.i.d.), and the narrow distribution to the worst possible vantage point, which has a checkered 16 × 16 bit pattern (strict anti-correlation between neighboring bits).

Our tree structure is indeed an FQ-tree, and it utilizes a collection of outstanding vantage points. Based on the observation that neighboring hash bits of BMB hashes are correlated (see Sect. Match decision) we identified vantage points which lead to very wide distance distributions. This is illustrated in Fig. 1. Good vantage points are characterized by smooth bit patterns, which have a strong correlation between neighboring bits and only few positions with non-equal values for neighboring bits.[5] Our first 6 vantage points are shown in Fig. 2. All of our vantage points have a balanced number of 0 s and 1 s in each quadrant because they are optimized for the quadrant-wise hashing approach of ForBild. Moreover, the pairwise distance between all of our vantage points is half way the maximal distance (i.e. 128) because we want to maximize the combined effectivity of the vantage points.

Fanout

Depending on the scenario, very different branching factors can be preferable. The original vp-tree is a binary tree, but it is has also been generalized to a multi-way tree (Bozkaya and Ozsoyoglu, 1999). In the context of discrete distance functions (like the Hamming distance) the usage of one child for each possible distance value maximizes the fanout and simplifies the tree structure. In fact, FQ-trees have initially been proposed in this flavor (Baeza-Yates et al., 1994).

After experimenting with various approaches, we decided against the maximal fanout and implemented a tree structure with adjustable fanout. In our setting, a fanout of 10 combined with a depth of 4 provides good results.[6]

Each node selects data dependent distance ranges for its children, such that each child receives approximately the same number of data points. Such a quantile-based partitioning scheme is a standard approach for vp-trees. Consequently, each child in the tree covers an individual range of distance values. Children in the dense center cover a small range of distances while exterior children cover a large range of distances.

LSH-based index

Locality-sensitive hashing (LSH) is a generic probabilistic framework. The basic constituent of an LSH scheme is a family of hash functions which are likely to produce hash collisions for similar items.[7] When hashing two items with a hash function selected uniformly at random from the LSH family, the probability that the hash function maps both items to the same hash value corresponds to the similarity of the items. LSH schemes for various similarity measures and diverse kinds of items are known. The first LSH scheme MinHash (Broder, 1997; Broder et al., 1998) uses “min-wise independent permutations”, and it resembles the Jaccard index, which measures the similarity of sets. Indyk et al. invented the term “locality-sensitive hashing”, and they introduced an LSH scheme based on bit sampling for Hamming spaces (Indyk and Motwani, 1998; Gionis et al., 1999). Moreover, they explain how to map other vector spaces into a Hamming space in order to use the bit sampling scheme more generally. Charikar (2002) defined an LSH scheme based on random projections for the cosine similarity and another scheme for the earth mover distance.

An LSH scheme can be used for estimating the similarity of two items by randomly picking a certain number of hash functions from the LSH family and evaluating these functions on the two items. The approximate similarity score is the relative frequency of hash matches. This is useful if the evaluation of the actual similarity function is significantly more expensive than the evaluation of several hash functions. Moreover, a similarity hash can be constructed by concatenating the hash values produced by the selected hash functions.

More applications of LSH arise when comparing items against a large reference set. LSH can be used for indexing this reference set. A simple variant of an LSH index tabulates the reference items according to their hash values

[5] In terms of physics and signal processing, such vantage points are low-frequency or low-energy bit patterns.

[6] While a fixed depth of 4 is suitable for our experiments, the implementation should actually adjust the depth automatically to the size of the reference set.

[7] This concept differs from other similarity preserving hash functions in the sense that there is no similarity relation between different hash values.

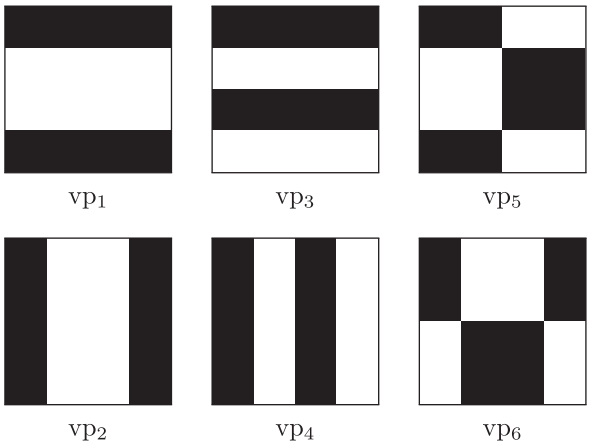

Fig. 2. The 6 best vantage points for ForBild depicted as bi-tonal images with 16 × 16 blocks.

under selected hash functions from the LSH family. Such an index can be employed for the batch calculation of approximate similarity scores. However, the typical query against a reference set does not demand a full list of similarity scores because only similar items are of interest. Suitable LSH indexes support this task by selecting candidates for close items. The similarity score is evaluated only for these candidates. In the following, we describe the structure of such an index (see also Gionis et al., 1999).

An LSH search index is a collection of L tables. Each table uses the combination of k randomly selected hash functions from the LSH family for hashing items into table buckets. In the case of bit sampling each table has 2^k buckets. LSH indexes may use $k = 1$ if the co-domain of the individual hash functions is sufficiently large like for typical applications of MinHash. The case $k = 1$ yields the simple index variant mentioned above. In general, k is used to configure the number of buckets in the index tables. For larger k there are more buckets. Thus less items go to the same bucket, and those items which go to the same bucket are closer related. On the downside, the probability increases that two similar items are hashed into different buckets. This problem is mitigated by using L different tables for increasing the chance that two similar items belong to the same bucket in at least one table. Hence the parameters k and L determine the trade-off between selectivity, sensitivity, and space/time requirement of the indexing approach.

Constructing an LSH index just means putting each reference item into the according bucket of each of the L tables. The bucket is determined by applying the k hash functions associated with the table. When answering a similarity query with the index, the query item is hashed as if it was inserted into the index. Each table provides a certain bucket, and the elements in these buckets are treated as potential neighbors of the query item. Finally, the similarity between the query item and these candidates can be calculated to find e.g. the closest candidate. With some probability, the closest candidate is the true closest neighbor.

Adaption for indexing BMB hashes

As BMB hashes are elements in a Hamming space, bit sampling is the proper LSH scheme for our task of indexing ForBild hashes. Each hash function of this scheme corresponds to one of the 256 bit positions of a ForBild hash. In contrast to the standard LSH indexing approach we do not select hash functions at random, but make choices which minimize the influence of the correlation between BMB hash bits.[8] Based on the geometry of ForBild hashes and the evaluation of some alternatives, we decided to use $L = 16$ tables and $k = 16$ bits per table. For each table the selected bits form a regular 4 × 4 grid on the BMB hashes, and the distance between neighboring bit positions is always 4. This strategy partitions the 256 bits in a structured way as shown in Fig. 3.

ForBild accepts a Hamming distance of up to 32 for declaring images as similar (see Sect. Match decision). Hence the LSH index should declare such hashes as candidates for neighbors, i.e. two similar hashes should be mapped into the same bucket by one of the index tables. In fact, if the Hamming distance of two hashes is smaller than 16, there is a guarantee that at least one of the $L = 16$ tables puts both hashes into the same bucket. But for larger distances there is the possibility, that two similar hashes are mapped into different buckets for all 16 LSH tables.

Fig. 4 shows the amount of missed hits as function of the Hamming distance. The values have been determined empirically on our test images (see Sect. Evaluation). Although the rate is close to 1/3 at the outer end, the total number of missed hits is low because such Hamming distances are rare even in a scenario with strong image modifications. In our experiment we had 150 missed hits out of 64,123 queries, which corresponds to an additional amount of 0.23% false negatives. As a small false negative rate is acceptable in the scenario of black- or whitelisting, the errors introduced by the LSH index are of minor relevance.

Evaluation

This section evaluates the acceleration gained with our indexing strategies. We implemented these strategies as well as a very efficient brute force search in C++. The brute force search serves as baseline of the evaluation.

All search strategies solve the query task of finding the closest neighbor within a reference set. If there are multiple items with the best distance, an arbitrary one of these is returned. Distances above the threshold of 32 are not considered when processing a query because ForBild defines larger distances as a mismatch (see Sect. Match decision).

All search strategies use the same efficient implementation of the Hamming distance in order to get meaningful time measurements. Note that creating fast indexing strategies is more challenging for inexpensive distance functions because the computational overhead of

[8] Gionis et al. (1999) already hypothesized that systematic choices should be able to take advantage of the structure of the data set.

0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
4	5	6	7	4	5	6	7	4	5	6	7	4	5	6	7
8	9	A	B	8	9	A	B	8	9	A	B	8	9	A	B
C	D	E	F	C	D	E	F	C	D	E	F	C	D	E	F
0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
4	5	6	7	4	5	6	7	4	5	6	7	4	5	6	7
8	9	A	B	8	9	A	B	8	9	A	B	8	9	A	B
C	D	E	F	C	D	E	F	C	D	E	F	C	D	E	F
0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
4	5	6	7	4	5	6	7	4	5	6	7	4	5	6	7
8	9	A	B	8	9	A	B	8	9	A	B	8	9	A	B
C	D	E	F	C	D	E	F	C	D	E	F	C	D	E	F
0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
4	5	6	7	4	5	6	7	4	5	6	7	4	5	6	7
8	9	A	B	8	9	A	B	8	9	A	B	8	9	A	B
C	D	E	F	C	D	E	F	C	D	E	F	C	D	E	F

Fig. 3. Assignment of the hash bits to table IDs 0…F.

the index has a larger influence in this case. The Hamming distance implementation uses a 16-bit lookup table (LUT) instead of the 8-bit LUT used by ForBild, and it has a cancel option for large distances. Moreover, modern CPUs provide a POPCNT ("population count") instruction, which implements the Hamming weight function in hardware. Hence we also created a Hamming distance implementation which uses the POPCNT instruction instead of a LUT.

Test setup

The experiments use 128,036 JPEG images with a total size of 40 GiB crawled from the Internet. We derived various image sets from this collection:

Complete. The complete collection.
Suspicious. A subset of 64,123 images, which was randomly selected by tossing a (virtual) coin for each image in the collection. This set is our artificial "blacklist".
Modified. Modified versions of the *suspicious* images, obtained by reducing the image resolution by 25% in both directions and by setting the JPEG quality to 20.
Unknown. The complement of the suspicious set. It contains 63,913 images.

The ForBild tool generated a hash list for each of the sets. The hash list for the *complete* set has a size of 3.9 MiB.

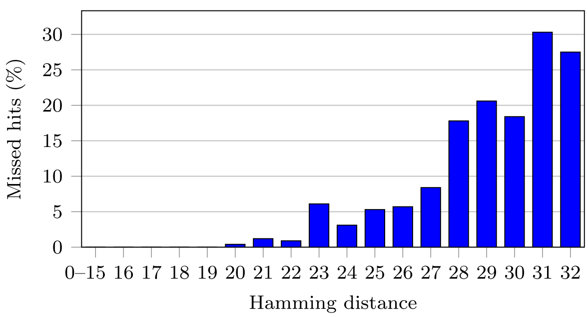

Fig. 4. Missed hits of our LSH strategy as function of the Hamming distance.

We run all test cases (see below) on two different computers, namely a laptop with a 5 year old Intel Core2 Duo P8700 processor and a workstation with a quite modern Intel Core i5-3570 processor. Both computers have Window 7 as operating system; our C++ implementations have been compiled with Visual Studio 2012. Main memory is not an issue for our tests because neither the hash sets nor the indexes consume a large amount of memory. Even a database with 100 M items as depicted in Sect. The need for efficient matching can be handled in RAM.

As the processor of the workstation supports the POPCNT instruction, the workstation runs both the LUT version and the POPCNT version while the laptop is restricted to the LUT version. The results use the following nomenclature for the test environments:

LAP LUT Laptop with LUT variant.
WS LUT Workstation with LUT variant.
WS POP Workstation with POPCNT variant.

Test cases and results

Each of our test cases uses one of the image sets from above as reference set, and another set as query list. The different tests analyze the performance of our indexing strategies in different situations. We run each test 10 times in each environment and – after removing a few outliers – calculate the average CPU time needed for performing the tests. Additionally, we determine the count of Hamming distance invocations for each search strategy (denoted as "count" in the result tables). This number is independent from the environment running the test.

As indexing strategies must construct their indexes before answering queries, we measure the time and the count of Hamming distance invocations needed for this preprocessing step, too. Table 1 provides these figures for the two reference sets used in the test cases, namely the *suspicious* and the *complete* set. Constructing a vp-tree index is less expensive than constructing an LSH index. Each hash inserted into the tree index requires 4 (depth of tree) distance comparisons to the vantage points while the LSH index does not use the distance function during construction. Instead it applies the $L = 16$ (number of tables) bit-sampling functions to each hash. All measurements for the test cases below include the construction process and show that the construction overhead is easily compensated by the performance of the search algorithm.

Case 1. This case corresponds to a classical scenario of similarity preserving hashing. Known suspicious files are

Table 1
Measurements for the index construction.

	suspicious set (Case 1, 2, and 3)		*complete* set (Case 4 and 5)	
	vp-tree	LSH	vp-tree	LSH
LAP LUT	14 ms	51 ms	37 ms	113 ms
WS LUT	8 ms	37 ms	18 ms	81 ms
WS POP	6 ms	37 ms	14 ms	81 ms
Count	256,492	0	512,144	0

Table 2
Measurements for Case 1: *modified* checked against *suspicious*.

	Baseline	vp-tree	LSH
LAP LUT	42.455 s	1.360 s	0.314 s
WS LUT	23.372 s	0.718 s	0.154 s
WS POP	15.070 s	0.530 s	0.137 s
Count	3,849,984,319	45,821,395	2,225,758

contained in a blacklist, and modified versions of these files reside in the investigation target. Hence we use the *suspicious* set as reference set, and the *modified* set as query set. Table 2 shows the measurements, and Fig. 5 illustrates the timing results.

Obviously, the laptop is the weakest environment, while the workstation with POPCNT instruction is the most powerful environment. However, the gain of better hardware is small compared to the gain of advanced algorithms. The search strategy based on the vp-tree runs about 30 times faster than the baseline in each environment. The LSH approach is even 135–150 (LUT variant) or 110 (POPCNT variant) times faster than the baseline. When considering the Hamming distance calculations, the vp-tree is 84 times more efficient than the baseline, and LSH even 1730 times. However, the computational overhead of the indexes reduces this advantage to the observed time benefits. In the case of LSH, the calculation of the $L = 16$ bit-sampling functions for each hash has a quite large impact.

Case 2. An investigation target usually contains many files which are not known by the blacklist. For simulating this, we match the *unknown* set against the *suspicious* set. Results are shown in Table 3.

The tree-based approach must process a relatively large number of nodes (and hence reference points) to decide that the reference list does not contain a matching item. Thus the speedup factor is only about 3, but this is still a valuable improvement for practice. Contrarily, the bucketing approach of the LSH index limits the number of items to be processed. Thus it does not degrade and gains a factor of 120 to 140. In terms of Hamming distance invocations, the vp-tree has an advantage of factor 5, and LSH has an advantage of factor 540.

Case 3. Despite the focus on similarity detection, the performance for the detection of exact matches is also relevant because an investigation target may contain unmodified versions of blacklisted files as well. Hence we compare the *suspicious* set against itself.

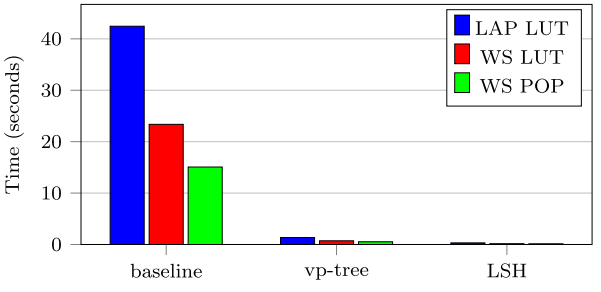

Fig. 5. Timing for Case 1.

Table 3
Measurements for Case 2: *unknown* checked against *suspicious*.

	Baseline	vp-tree	LSH
LAP LUT	60.526 s	20.520 s	0.498 s
WS LUT	34.974 s	11.543 s	0.245 s
WS POP	23.751 s	7.873 s	0.184 s
Count	4,097,939,910	798,022,874	7,552,142

Table 4
Measurements for Case 3: *suspicious* checked against *suspicious*.

	Baseline	vp-tree	LSH
LAP LUT	30.763 s	0.292 s	0.212 s
WS LUT	17.761 s	0.139 s	0.112 s
WS POP	12.140 s	0.125 s	0.109 s
Count	2,055,891,955	3,715,319	301,339

Now the vp-tree benefits strongly from its metric structure. It finds the matching item about 100 to 130 times faster than the baseline as we can learn from Table 4. The LSH index is 145–160 (LUT variant) or 110 (POPCNT variant) times faster than the baseline. The count of Hamming distance invocations is even 550 times better than the baseline when using the tree index and 6800 times better when using the LSH index.

Case 4. This case increases the reference set of Case 1 by using the *complete* set as reference. The *modified* set provides the query items.

The expected result is a doubling of runtime and Hamming distance invocations compared to Case 1. While most values in Table 5 satisfy this expectation roughly, the tree-based approach is 56% slower than expected on the laptop. Maybe this is caused by an overproportionally increased number of CPU cache misses due to the larger reference set. However, the vp-tree is still more than 20 times faster than the baseline. In contrast, the LSH approach is 20% faster than expected as the time for calculating the bit-sampling functions is independent from the reference set.

Another surprise is the speed of our baseline compared to ForBild. In Sect. Query performance we provided the estimation of 5.0 ms for answering a query against the *complete* set on the workstation. Contrarily, the baseline on the workstation with LUT needs just about 0.73 ms in this test case. Extrapolating the results from Case 2 to the complete reference set yields about 1.1 ms for processing an unknown query item with the baseline.

Case 5. Now we compare the *complete* set against itself in analogy to Case 3.

The expected result is four times the processing time of Case 3 because we double both the reference and query set.

Table 5
Measurements for Case 4: *modified* checked against *complete*.

	Baseline	vp-tree	LSH
LAP LUT	92.479 s	4.222 s	0.502 s
WS LUT	46.492 s	1.381 s	0.242 s
WS POP	30.161 s	0.964 s	0.214 s
Count	7,682,867,620	89,947,700	4,287,182

Table 6
Measurements for Case 5: *complete* checked against *complete*.

	Baseline	vp-tree	LSH
LAP LUT	124.437 s	0.991 s	0.461 s
WS LUT	70.400 s	0.370 s	0.244 s
WS POP	48.005 s	0.304 s	0.242 s
Count	8,196,503,659	13,754,314	1,078,818

The baseline satisfies this expectation quite well according to our timings shown in Table 6. The indexing strategies perform even better. Hence the speedup is even factor 125 to 190 for the tree-based search and factor 200 to 290 for the LSH-based search.

Summary

Our indexing strategies beat the baseline in all situations. The tree-based strategy degrades for non-matching items, but it surpasses the baseline still by factor 3 in the worst case. In the classical case of modified, but known, content it is 30 times faster than the baseline, and in the situation of unmodified content it exceeds even factor 100. The performance of the LSH approach is less sensitive to the type of query item, and the speedup factor ranges from 110 to more than 200. LSH achieves its stable speed by taking less care for cases close to the distance threshold, which is paid with a small rate of false negatives (see Sect. LSH-based index).

Related work

A plethora of indexing approaches has been developed in the last decades. In the following we cite some work on indexing of images and other types of media.

The perceptual image hash library pHash (Klinger and Starkweather, 2008–2010) is accompanied by an implementation of the MVP-tree approach (Bozkaya and Ozsoyoglu, 1999), which is a special class of vp-trees. We ran some tests with this code on our data and observed that it is inherently slower than our brute force baseline. Hence we did not include it in our evaluation.

Zhang et al. (2011) propose an indexing strategy derived from the LSH indexing approach. Their strategy replaces the LSH tables by a tree-like structure. The main advantage of this approach is efficient dynamic updates of the index. They combine it with the LSH scheme of random projections and apply their strategy to features extracted from images. Grauman and Darrell (2007) describe an indexing approach for approximate nearest neighbor search on feature vectors of images. The basic idea here is to utilize multi-resolution histograms and random projections. Muja and Lowe (2009, 2012) use variants of hierarchical k-means trees and randomized k-d trees for indexing image features.

Other multimedia data are considered in the literature as well. Haitsma and Kalker (2002) developed an indexing strategy for audio fingerprints by using a large index table which stores references to sub-fingerprints. Miller et al. (2002) created a tree-based index for audio data.

In the domain of hash functions for raw binary data, we introduced F2S2 (Winter et al., 2013) for indexing and searching piecewise hash signatures like ssdeep hashes (Kornblum, 2006a, 2006b). The method applied in F2S2 is based on the *n*-grams contained in piecewise hash signatures. The tool sdhash (Roussev, 2010a, 2010b), which implements a different hashing algorithm, supports indexing since version 3.0 (October 2012). It seems to use large Bloom filters for this purpose, but the method has not been described in a publication by now.

Conclusion

Indexing strategies are valuable tools to encounter the ever growing amounts of data in forensic investigations. We presented two powerful solutions for indexing robust image hashes and other bit-vector data. These solutions will be integrated into the ForBild tool, and they can also be adopted for similar hashing approaches. Hence such tools will be prepared for growing databases and applications with higher data throughput like robust video hashing.

Acknowledgment

This work was supported by CASED (www.cased.de).

References

Baeza-Yates R, Cunto W, Manber U, Wu S. Proximity matching using fixed-queries trees. In: Combinatorial pattern matching. LNCS, vol. 807. Springer; 1994. pp. 198–212.
Bozkaya T, Ozsoyoglu M. Indexing large metric spaces for similarity search queries. ACM Trans Database Syst 1999;24(3):361–404.
Breitinger F, Liu H, Winter C, Baier H, Rybalchenko A, Steinebach M. Towards a process model for hash functions in digital forensics. In: ICDF2C 2013. Institute for Computer Sciences, Social Informatics and Telecommunications Engineering; 2013.
Broder AZ. On the resemblance and containment of documents. In: Compression and complexity of sequences 1997. IEEE Comput Soc; 1997:21–9.
Broder AZ, Charikar M, Frieze AM, Mitzenmacher M. Min-wise independent permutations. In: 30th Annual ACM Symposium on Theory of Computing. ACM; 1998. pp. 327–36.
Burkhard WA, Keller RM. Some approaches to best-match file searching. Commun ACM 1973;16(4):230–6.
Charikar M. Similarity estimation techniques from rounding algorithms. In: 34th Annual ACM Symposium on Theory of Computing. ACM; 2002. pp. 380–8.
Gionis A, Indyk P, Motwani R. Similarity search in high dimensions via hashing. In: 25th International Conference on Very Large Data Bases. Morgan Kaufmann; 1999. pp. 518–29.
Grauman K, Darrell T. Pyramid match hashing: sub-linear time indexing over partial correspondences. In: Computer vision and pattern recognition 2007. IEEE Comput Soc; 2007.
Haitsma J, Kalker T. A highly robust audio fingerprinting system. In: ISMIR 2002. IRCAM – Centre Pompidou; 2002. pp. 107–15.
Indyk P, Motwani R. Approximate nearest neighbors: towards removing the curse of dimensionality. In: 30th Annual ACM Symposium on Theory of Computing. ACM; 1998. pp. 604–13.
Jin M, Yoo CD. Quantum hashing for multimedia. IEEE T Inf Foren Sec 2009;4(4):982–94.
Klinger E, Starkweather D. phash. http://phash.org; 2008–2010.
Kornblum J. Identifying almost identical files using context triggered piecewise hashing. Digit Investig 2006a;3(Suppl.):S91–7.
Kornblum J. ssdeep. http://ssdeep.sourceforge.net; 2006b.
Miller ML, Rodriguez MA, Cox IJ. Audio fingerprinting: nearest neighbor search in high dimensional binary spaces. In: IEEE Workshop on Multimedia Signal Processing. IEEE Sig Process Soc; 2002:182–5.
Muja M, Lowe DG. Fast approximate nearest neighbors with automatic algorithm configuration. In: Fourth International Conference on Computer Vision Theory and Applications, vol. 1. INSTICC Press; 2009. pp. 331–40.

Muja M, Lowe DG. Fast matching of binary features. In: Ninth Conference on Computer and Robot Vision. IEEE Comput Soc; 2012:404–10.
Oostveen JC, Kalker T, Haitsma J. Visual hashing of digital video: applications and techniques. In: Applications of digital image processing XXIV. Proc. SPIE, vol. 4472. SPIE Optics & Photonics; 2001. pp. 121–31.
Roussev V. Data fingerprinting with similarity digests. In: Advances in digital forensics VI. IFIP AICT, vol. 337. Springer; 2010a. pp. 207–26.
Roussev V. sdhash. http://roussev.net/sdhash; 2010b.
Steinebach M. Robust hashing for efficient forensic analysis of image sets. In: Digital forensics and cyber crime. LNICST, vol. 88. Springer; 2012. pp. 180–7.
Steinebach M, Liu H, Yannikos Y. ForBild: efficient robust image hashing. In: Media watermarking, security, and forensics 2012. IS&T/SPIE Electronic Imaging. Proc. SPIE, vol. 8303; 2012. p. 0O-1–8.
Steinebach M, Liu H, Yannikos Y. FaceHash: face detection and robust hashing. In: ICDF2C 2013. Institute for Computer Sciences, Social Informatics and Telecommunications Engineering; 2013.
Uhlmann JK. Satisfying general proximity/similarity queries with metric trees. Inf Process Lett 1991;40(4):175–9.
Winter C, Schneider M, Yannikos Y. F2S2: fast forensic similarity search through indexing piecewise hash signatures. Digit Investig 2013; 10(4):361–71.
Yang B, Gu F, Niu X. Block mean value based image perceptual hashing. In: Intelligent information hiding and multimedia signal processing. IEEE Comput Soc; 2006:167–72.
Yianilos PN. Data structures and algorithms for nearest neighbor search in general metric spaces. In: Fourth ACM-SIAM Symposium on Discrete Algorithms. ACM/SIAM; 1993. pp. 311–21.
Zauner C, Steinebach M, Hermann E. Rihamark: perceptual image hash benchmarking. In: Media watermarking, security, and forensics III. Proc. SPIE, vol. 7880. IS&T/SPIE Electronic Imaging; 2011. p. 0X-1–15.
Zhang D, Agrawal D, Chen G, Tung AKH. HashFile: an efficient index structure for multimedia data. In: ICDE Conference 2011. IEEE Comput Soc; 2011:1103–14.