A Fuzzy Hashing Approach based on Random Sequences and Hamming Distance

Frank Breitinger and Harald Baier

Center for Advanced Security Research Darmstadt (CASED)
and Department of Computer Science, Hochschule Darmstadt,
Mornewegstr. 32, D – 64293 Darmstadt, Germany
Mail: `{frank.breitinger,harald.baier}@cased.de`

Abstract. Hash functions are well-known methods in computer science to map arbitrary large input to bit strings of a fixed length that serve as unique input identifier/fingerprints. A key property of cryptographic hash functions is that even if only one bit of the input is changed the output behaves pseudo randomly and therefore similar files cannot be identified. However, in the area of computer forensics it is also necessary to find similar files (e.g. different versions of a file), wherefore we need a similarity preserving hash function also called *fuzzy hash function*.
In this paper we present a new approach for *fuzzy hashing* called `bbHash`. It is based on the idea to 'rebuild' an input as good as possible using a fixed set of randomly chosen byte sequences called *building blocks* of byte length l (e.g. $l = 128$). The proceeding is as follows: slide through the input byte-by-byte, read out the current input byte sequence of length l, and compute the Hamming distances of all building blocks against the current input byte sequence. Each building block with Hamming distance smaller than a certain threshold contributes the file's `bbHash`. We discuss (dis-)advantages of our `bbHash` to further fuzzy hash approaches. A key property of `bbHash` is that it is the first fuzzy hashing approach based on a comparison to external data structures.

Keywords: Fuzzy hashing, similarity preserving hash function, similarity digests, Hamming distance, computer forensics.

1 Introduction

The distribution and usage of electronic devices increased over the recent years. Traditional books, photos, letters and LPs became ebooks, digital photos, email and mp3. This transformation also influences the capacity of todays storage media ([1]) that changed from a few megabyte to terabytes. Users are able to archive all their information on one simple hard disk instead of several cardboard boxes on the garret. This convenience for consumers complicates computer forensic investigations (e.g. by the Federal Bureau of Investigation), because the investigator has to cope with an information overload: A search for relevant files resembles no longer to find a needle in a haystack, but more a needle in a hay-hill.

The crucial task to solve this data overcharge is to distinguish relevant from non-relevant information. In most of the cases an automated preprocessing is used, which tries to filter out some irrelevant data and reduces the amount of data an investigator has to look at by hand. As of today the best practice of this preprocessing is quite simple: Hash each file of the evidence storage medium, compare the resulting hash value (also called fingerprint or signature) against a given set of fingerprints, and put it in one of the three categories: known-to-be-good, known-to-be-bad, and unknown files. For instance, unmodified files of a common operating system (e.g. Windows, Linux) or binaries of a wide-spread application like a browser are said to be known-to-be-good and need not be inspected within an investigation. The most common set/database of such non-relevant files is the *Reference Data Set* (RDS) within the *National Software Reference Library* (NSRL, [2]) maintained by the US National Institute of Standards and Technology (NIST)[1].

Normally cryptographic hash functions are used for this purpose, which have one key property: Regardless how many bits changes between two inputs (e.g. 1 bit or 100 bits), the output behaves pseudo randomly. However, in the area of computer forensics it is also convenient to find similar files (e.g. different versions of a file), wherefore we need a similarity preserving hash function also called *fuzzy hash function*.

It is important to understand that in contrast to cryptographic hash functions, there is currently no sharp understanding of the defining properties of a fuzzy hash function. We will address this topic in Sec. 2, but we emphasize that the output of a fuzzy hash function need not be of fixed length.

In general we consider two different levels for generating similarity hashes. On the one hand there is the byte level which works independently of the file type and is very fast as we need not interpret the input. On the other hand there is the semantic level which tries to interpret a file and is mostly used for multimedia content, i.e. images, videos, audio. In this paper we only consider the first class. As explained in Sec. 2 all existing approaches from the first class come with drawbacks with respect to security and efficiency, respectively.

In this paper we present a new fuzzy hashing technique that is based on the idea of data deduplication (e.g. [3, Sec. II]) and eigenfaces (e.g. [4, Sec. 2]). Deduplication is a backup scheme for saving files efficiently. Instead of saving files as a whole, it makes use of small pieces. If two files share a common piece, it is only saved once, but referenced for both files. Eigenfaces are a similar approach. They are used in biometrics for face recognition. Roughly speaking, if we have a set of N eigenfaces, then any face can be represented by a combination of these standard faces. Eigenfaces resemble to the well-known method in linear algebra to represent each vector of a vector space by a linear combination of the basis vectors.

Our approach uses a fixed set of random byte sequences called *building blocks*. In this paper we denote the number of building blocks by N. The length in bytes

[1] NIST points out that the term known-to-be-good depends on national laws. Therefore, NIST calls files within the RDS non-relevant. However, the RDS does not contain any illicit data

of a building block is denoted by l. It shall be 'short' compared to the file size (e.g. $l = 128$). To find the optimal representation of a given file by the set of building blocks, we slide through the input file byte-by-byte, read out the current input byte sequence of length l, and compute the Hamming distances of all building blocks against the current input byte sequence. If the building block with the smallest Hamming distance is smaller than a certain threshold, too, its index contributes to the file's `bbHash`.

Besides similarity of files we are also able to match back parts of files to its origin, which could arise due to file fragmentation and deletion.

1.1 Contribution and Organization of this Paper

Similarity preserving hash functions on the byte level are a rather new area in computer forensics and get more and more important due to the increasing amount of data. Currently the most popular approach is implemented in the software package `ssdeep` ([5]). However, `ssdeep` can be exploited very easily ([6]). Our approach `bbHash` is more robust against an active adversary. With respect to the length of the hash value our approach is superior to `sdhash` ([7, 8]), which generates hash values of about 2.6% to 3.3% of the input, while our `bbHash` similarity digest only comprises 0.5% of the input size. Additionally, `sdhash` has a coverage of only about 80% of the input, while our approach is designed to cover the whole input.

Additionally, in contrast to `ssdeep` and `sdhash`, our similarity digest computation is based on a comparison to external data structures, the building blocks. The building blocks are randomly chosen static byte blocks being independent of the processed input. Although this implies a computational drawback compared to other fuzzy hashing approaches, we consider this as a security benefit as `bbHash` stronger withstands active anti-blacklisting.

The rest of the paper is organized as follows: In the subsequent Sec. 1.2 we introduce the notation used in this paper. Then in Sec. 2 we introduce the related work. The core of this paper is given in Sec. 3 where we present our algorithm `bbHash`. We give a first analysis of our implementation in Sec. 4. Sec. 5 concludes our paper and gives an overview of future work.

1.2 Notation and Terms used in this Paper

In this paper, we make use of the following notation and terms:

- A *building block* is a randomly chosen byte sequence to rebuild files.
- N denotes the number of different building blocks used in our approach. We make use of the default value $N = 16$.
- For $0 \leq k \leq N$ we refer to the k-th building block as bb_k.
- l denotes the length of a building block in *bytes.* Throughout this paper we assume a default value of $l = 128$.
- l_{bit} denotes the length of a building block in *bits.* In this paper we assume a default value of $l_{bit} = 8 \cdot 128 = 1024$.

- L_f denotes the length of an input (file) in bytes.
- `bbHash` denotes our new proposed fuzzy hash function on base of building blocks.
- BS denotes a byte string of length l: $BS = B_0 B_1 B_2 \cdots B_{l-1}$.
- BS_i denotes a byte string of length l starting at offset byte i within the input file: $BS_i = B_i B_{i+1} B_{i+2} \cdots B_{i+l-1}$.
- t denotes the threshold value. t is an integer with $0 \leq t \leq l_{bit}$.

2 Foundations and Related Work

According to [9] hash functions have two basic properties, *compression* and *ease of computation*. In this case compression means that regardless the length of the input, the output has a fixed length. This is why the term Fuzzy *Hashing* might be a little bit confusing and *similarity digest* is more appropriate – most of the similarity preserving algorithms do not output a fixed sized hash value. Despite this fact we call a variable-sized compression function h_f a fuzzy hash function if two similar inputs yield similar outputs.

The first fuzzy hashing approach on the byte level was proposed by Kornblum in 2006 [5], which is called context-triggered piecewise hashing, abbreviated as CTPH. Kornblum's CTPH is based on a spam detection algorithm of Andrew Tridgell [10]. The main idea is to compute cryptographic hashes not over the whole file, but over parts of the file, which are called *chunks*. The end of each chunk is determined by a pseudo-random function that is based on a current context of 7 bytes. In recent years, Kornblum's approach was examined carefully and several papers had been published.

[11, 12, 13] find ways to improve the existing implementation called `ssdeep` with respect to both efficiency and security. On the other side [6, 14] show attacks against CTPH with respect to blacklisting and whitelisting and also some improvements for the pseudo random function.

In [7, 8] Roussev presents a similarity digest hash function `sdhash`, where the idea is "to select multiple characteristic (invariant) features from the data object and compare them with features selected from other objects". He uses multiple unique 64-byte features selected on the basis of their entropy. In other words files are similar if the have the same features/byte-sequences. [15] explains a tool called `Simhash` which is another approach for fuzzy hashing "based on counting the occurrances of certain binary strings within a file". As we denote a similarity preserving hash function a *fuzzy hash function*, we rate `ssdeep`, `sdhash` and `Simhash` as possible implementations for fuzzy hashing.

[16] comes with a tool called `md5bloom` where "the goal is to pick from a set of forensic images the one(s) that are most like (or perhaps most unlike) a particular target". In addition, it can be used for object versioning detection. The idea is to hash each hard disk block and insert the fingerprints into a Bloom filter. Hard disks are similar if their Bloom filters are similar.

3 A New Fuzzy Hashing Scheme based on Building Blocks

In this section, we explain our fuzzy hashing approach `bbHash`. First, in Sec. 3.1 we describe the generation of the building blocks followed by the algorithm details in Sec. 3.2. Finally, Sec. 3.3 describes how to compare two `bbHash` similarity digests.

`bbHash` aims at the following paradigm:

1. *Full coverage*: Every byte of the input file is expected to be within at least one offset of the input file, for which a building block is triggered to contribute to the `bbHash`. This behaviour is very common for hash functions: each byte influences the hash value.
2. *Variable-sized length*: The length of a file's `bbHash` is proportional to the length of the original file. This is in contrast to classical hash functions. However, it ensures to be able to store sufficiently information about the input to have good similarity preserving properties of `bbHash`.

3.1 Building Blocks

We first turn to the set of building blocks. Their number is denoted by N. In our current implementation we decided to set the default value to $N = 16$. Then we can index each building block by a unique hex digit $0, 1, 2 \dots f$ (half a byte), therefore we have the building blocks $bb_0, bb_1, \cdots, bb_{16-1}$. This index is later used within the `bbHash` to uniquely reference a certain building block.

The length of a building block in bytes is referred to by l and influences the following two aspects:

1. A growing l decreases the speed performance as the Hamming distance is computed at each offset i for l bytes.
2. An increased l shortens the length of the hash value as there should be a trigger sequence every l bytes approximately (depending on the threshold t).

Due to run time efficiency reasons we decided to use a 'short' building block size compared to the file size. Currently we make use of $l = 128$.

The generation of the building blocks is given in Fig. 1. We use the `rand()` function to fill an array of unsigned integers. Hence, all building blocks are stored in one array whereby the boundaries can be determined by using their size. As `rand()` uses the same seed each time, it is a deterministic generation process. Using a fixed set of building blocks is comparable to the use of a fixed initialization vector (IV) of well-known hash functions like SHA-1. A sample building block is given in Fig. 2.

3.2 Our Algorithm `bbHash`

In this section we give a detailed description of our fuzzy hash function `bbHash`. It works as follows: To find the optimal representation of a given file by the set of building blocks, we slide through the input file byte-by-byte, read out the current input byte sequence of length l, and compute the Hamming distances of

```
/** amount: amount of the building blocks; default=16
 *  size: bitsize of each Building block; default=1024 */
uint32_t setBuildingBlocks(int amount, int size) {
    int i; uint32_t *tmp;

    tmp = (uint32_t *)malloc(sizeof(uint32_t) * amount * size/32);
    for (i = 0; i < amount * size / 32; i++) {
        tmp[i] = rand() << 10 ^ rand();
   }
    return *tmp;
}
```

Fig. 1. Generation of the building blocks

```
1f6ebfc6:9452ec73:da3818ff:c00470ec:7e94d8cd:3d1c3fab:84b0d6fb:ffe26d46:
27d192c2:cd9077f8:479a85e8:25d8df8d:0b6ffd5a:7e09ca63:68e71b9f:f47b239a:
8e09c532:2e5be3b7:ca6b2058:a679555a:d3081d5d:98d9b717:23bc3ae9:fbb3d0d4:
cf7520b2:4440d4c6:3d3ef2b4:e039d611:d9802582:5ff25641:83e9393d:82dd8087
```

Fig. 2. bb_0: building block with index 0

all building blocks against the current input byte sequence. If the building block with the smallest Hamming distance is smaller than a certain threshold, too, its index contributes to the file's `bbHash`.

We write L_f for the length of the input file in bytes. The pseudocode of our algorithm `bbHash` is given in Algorithm 1. It proceeds as follows for each offset i within the input file, $0 \leq i \leq L_f - 1 - l$: If BS_i denotes the byte sequence of length l starting at the i-th byte of the input, then the algorithm computes the N Hamming distances of BS_i to all N building blocks: $hd_{k,i} = HD(bb_k, BS_i)$ is the Hamming distance of the two parameters bb_k and BS_i, $0 \leq k < N$. As the Hamming distance is the number of different bits, we have $0 \leq hd_{k,i} \leq 8l$. In Sec. 3.1 we defined the default length of a building block in bytes to be 128, i.e. we assume $l = 128$. As an example $HD(bb_2, BS_{100})$ returns the Hamming distance of the building block bb_2 and the bytes B_{100} to B_{227} of the input. In other words the algorithm slides through the input, byte-by-byte, and computes the Hamming distance at each offset for all N building blocks like it is given in Fig. 3.

The `bbHash` value is formed by the ordered indicies of triggered building blocks. In order to trigger a building block to contribute to the `bbHash`, it has to fulfill two further conditions:

1. For a given i (fixed offset), we only make use of the closest building block, i.e. we are looking for the index k with the smallest Hamming distance $hd_{k,i}$.
2. This smallest $hd_{k,i}$ also needs to be smaller than a certain threshold t.

Each BS_i that fulfills both conditions will be called a trigger sequence. To create the final `bbHash` hash value, we concatenate all indicies k of all triggered building

Algorithm 1 Pseudocode of the `bbHash` Algorithm

```
$l$                      ▷ Processed chunk size; default length is 128 bytes
$N$                      ▷ Amount of building blocks; default is 16
$BS_i$                   ▷ A byte sequence of length $l$ starting at the $i$-th byte of the input
$t$                      ▷ Threshold; default is 461
$L_f$                    ▷ Length of the input file in bytes
$mHD, tHD, tk$           ▷ Unsigned int; temporary variables
$signature$              ▷ String variable for final hash value

$bb$ = set_building_blocks($N$, $l$)                 ▷ Processing is given in Fig. 1
for $i = 0 \to L_f - 1 - l$ do                      ▷ run through input, byte-by-byte
$tHD = mHD = 0xffffffff$                            ▷ reset
    for $k = 0 \to N - 1$ do                        ▷ run through all building blocks
        $tHD$ = getHammingDistance($bb_k$, $BS_i$)
        if $tHD < mHD$ then                         ▷ two same HDs will use the smaller $k$
            $mHD = tHD$
            $tk = k$                                ▷ $tk$ is a hex digit
        end if
    end for
    if $mHD < t$ then signature = "signature" + $tk$    ▷ Conversion to string
    end if
end for
```

blocks (in case we have two triggered building blocks for BS_i, only the smallest index k is chosen).

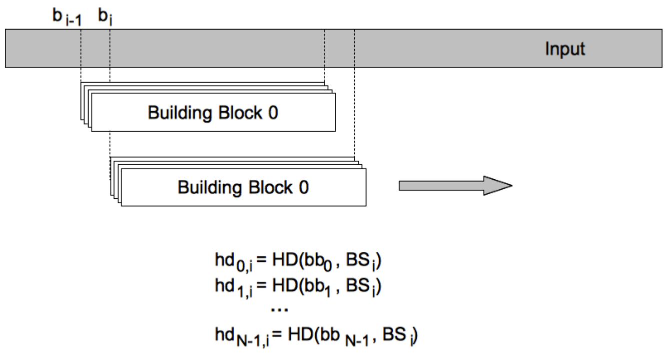

Fig. 3. Workflow of the Algorithm

In Sec. 3.1 we've already motivated why the choices $l = 128$ and $N = 16$ are appropriate for our approach. In what follows we explain how to choose a fitting threshold t. Our first paradigm in the introduction to Sec. 3 states *full coverage*, i.e. *every byte of the input file is expected to be within at least one offset of the* *input file, for which a building block is triggered to contribute to the* `bbHash`. Thus we expect to trigger every l-th byte, i.e. every 128-th byte. In order to have some overlap, we decrease the statistical expectation to trigger at every 100-th byte.

For our theoretic considerations we assume a uniform probability distribution on the input file blocks of byte length l. Let d be a non-negative integer (the distance) and $P(hd_{k,i} = d)$ denote the probability, that the Hamming distance of our building block k at offset i of the input file is equal to d.

1. We first consider the case $d = 0$, i.e. the building block and the input block coincide. Then we simply have $P(hd_{k,i} = 0) = 0.5^{l_{bit}} = 0.5^{1024}$.
2. For $d \geq 1$ we have $\binom{l_{bit}}{d}$ possibilities to find an input file block of Hamming distance d to bb_k. Thus $P(hd_{k,i} = d) = \binom{l_{bit}}{d} \cdot 0.5^{l_{bit}} = \binom{1024}{d} \cdot 0.5^{1024}$.
3. Finally, the probability to receive a Hamming distance smaller than t for bb_k is

$$p_1 := P(hd_{k,i} < t) = 0.5^{l_{bit}} \cdot \sum_{i=0}^{t-1} \binom{l_{bit}}{i} = 0.5^{1024} \cdot \sum_{i=0}^{t-1} \binom{1024}{i} . \qquad (1)$$

The binomial coefficients in Eq. (1) are large integers and we make use of the computer algebra system `LiDIA`[2] to evaluate Eq. (1) (`LiDIA` is a C++-library maintained by the Technical University of Darmstadt).

Let p_t denote the probability that at least one of the N buildings blocks satisfies Eq. (1), i.e. we trigger our input file and find a contribution to our `bbHash`. This may easily computed by the opposite probability that none of the building blocks triggers, that is $p_t = 1-(1-p_1)^N$. As explained above we aim at $p_t = 0.01$. Thus we have to find a threshold t with

$$0.01 = 1 - (1 - p_1)^N \iff p_1 = 1 - 0.99^{\frac{1}{N}} = 1 - \sqrt[16]{0.99} = 0.00062795 . \qquad (2)$$

Now we use our `LiDIA`-computations of Eq. (1) to identify a threshold of $t = 461$.

An example hash value of a 68,650 byte ($\approx$ 70 kbyte) JPG image is given in Fig. 4. Overall the hash value consists of 693 digits which is 346.5 bytes and therefore approximately 0.5% of the input.

3.3 Hash Value Comparison

This paper focuses on the hash value generation and not on its comparison. Currently there are two approaches from existing fuzzy hash functions which could be also usable for `bbHash`:

- Roussev uses Bloom filters where the similarity can be identified by generating the Hamming distance.
- Kornblum uses the weighted edit distance to compare the similarity of two hash values.

Both approaches may easily be adopted to be used for `bbHash`.

[2] `http://www.cdc.informatik.tu-darmstadt.de/TI/LiDIA/`; visited 05.09.2011

```
Reading ‘hacker_siedlung.jpg’...
4b:87:da:82:62:63:55:f6:07:a5:93:31:26:b6:75:35:50:b4:62:c7:a8:48:4a:
dd:7c:e1:07:e6:d9:6d:7e:d6:2d:c4:1b:62:94:9f:a0:7a:c0:ac:43:d3:a1:32:
3e:e8:a9:f4:93:9c:c4:b2:b4:a1:05:c0:e0:3e:f6:3c:ec:5d:b7:d3:d2:69:38:
bd:68:8a:12:28:e4:0d:fc:6c:d2:1b:3a:02:60:b7:39:20:81:50:8d:95:db:6f:
f1:77:11:5f:27:5b:f7:ce:88:b2:e6:19:7f:d0:e8:83:2e:f3:7a:05:f5:5c:52:
b1:0e:a6:44:4e:9b:a9:1d:9a:d0:84:f2:91:a8:db:72:b1:40:c7:f8:a6:9b:65:
73:6c:16:0b:b1:11:ac:3e:f9:3d:c3:6e:6a:9a:1b:2e:ef:17:52:2e:32:e8:71:
1f:29:41:be:80:72:bb:46:c8:7f:fa:d6:a2:04:3d:80:54:dc:a8:e7:9a:3b:a5:
36:79:c8:8b:41:dd:00:76:9a:59:cf:88:19:1f:26:f1:64:a8:89:b6:e9:5d:a5:
67:52:61:10:aa:e4:7a:af:80:7d:69:00:54:86:f5:3d:69:34:05:f0:ba:4f:e5:
27:65:15:cf:5f:a7:ca:b1:30:aa:85:98:d1:0e:79:8e:53:42:01:c0:71:82:a9:
7c:f1:78:7f:a6:8e:27:7b:c8:72:79:a9:3b:7c:bc:ff:0a:5a:b0:9e:8e:04:80:
c6:d6:a0:72:20:7b:c8:61:e0:a9:98:17:b1:67:b3:4d:41:6d:b9:a4:66:ea:74:
9f:97:88:63:06:46:87:67:14:10:fc:38:06:d3:ae:c6:fa:98:73:35:8e:55:44:
89:f7:33:23:39:16:85:62:2c:2e:57:7e:21:71:a9:59:49:df:cb:13:39:bf:00:
3b:2
total hash length: 693 digits
```

Fig. 4. `bbHash` of a 68,650 byte JPG-Image

4 Analysis of `bbHash`

In this section we analyze different aspects of our approach. The length of the hash values and how can it be influenced by different parameters is presented in Sec. 4.1. Our current choices yield a `bbHash` of length 0.5% of the input size. This is an improvement by a factor 6.5 compared to `sdhash`, where for practical data the similarity digest comprises 3.3% of the input size. Next, in Sec. 4.2 we discuss the applicability of `bbHash` depending on the file size of the input. An important topic in computer forensics is the identification of segments of files, which is addressed in Sec. 4.3. The run time performance is shown in Sec. 4.4. At least we have a look at attacks and compare our approach against existing ones. We show that `bbHash` is more robust against an active adversary compared to `ssdeep`.

4.1 Hash Value Length

The hash value length depends on three different properties: the file size L_f, the threshold t and the building block length l. If we expect that both other parameters are fixed, then

– a larger L_f will increase the hash value length as the input is supposed to have more trigger sequences.
– a higher t will increase the hash value length as more BS_i will have a Hamming distance lower than the threshold t.

– a large l will decrease the performance and the hash value length[3].

In order to have full coverage our aim is to have a final hash value where every input byte influences the final hash value. Due to performance reasons we've decided for a building block length of $l = 128$ byte. As we set the threshold to $t = 461$, the final hash value length nearly results in $\frac{L_f}{100}$ digits whereby every digit has half a byte length. Thus the final hash value is approximately 0.5% of the original file size.

Compared to the existing approach `sdhash` where "the space requirements for the generated similarity digests do not exceed 2.6% of the source data" [8], it is quite short. However, for real data `sdhash` generates similarity digests of 3.3% of the input size.

Besides these two main aspects the file type may also influence the hash value length. We expect that random byte strings will have more trigger sequences. Therefore we think that compressed file formats like ZIP, JPG or PDF have very random byte strings and thus yield a `bbHash` of 0.5% of the input size. This relation may differ significantly when processing TXT, BMP or DOC formats as they are less random.

4.2 Applicability of `bbHash` Depending on the File Size

Our first prototype is very restricted in terms of the input file size and will not work reliable for small files. Files smaller than the building blocks' length l cannot have a trigger sequence and cannot be hashed. To receive some trigger sequences the input file should be at least 5000 bytes, which results in $5000 - 1 - l$ possible trigger sequences. On the other side large files result in very long hashes wherefore we recommend to process files smaller than a couple of megabytes.

By changing the threshold t, it is possible to influence the hash value length. We envisage to customize this parameter depending on the input size which will be a future topic.

4.3 Detection of Segments of Files

A file segment is a part of a file, which could be the result of fragmentation and deletion. For instance, if a fragmented file is deleted but not overwritten, then we can find a byte sequence, but do not know anything about the original file. Our approach allows to compare hashes of those pieces against hash values of complete files. Depending on the fragment size, `ssdeep` is not able to detect fragments (roughly `ssdeep` may only fragments being at least about half the size of the original file size).

Fig. 5 simulates the aforementioned scenario. We first copy a middle segment of 20000 byte from the JPG image from Fig. 4 using `dd`. We then compute the hash value of this segment. If we compare this hash value against Fig. 4[4] (starting

[3] Remember, we would like to have a triggering in approximately every l-th byte and therefore we have to adjust t.

[4] We rearranged the output by hand to make an identification easier.

line 4), we recognize that they are completely identical. Thus we can identify short segments as a part of its origin.

```
$ dd if=hacker_siedlung.jpg of=hacker_siedlung.mid.jpg bs=1 count=20000
   skip=20175

$ ./bbHash
Reading ‘hacker_siedlung.mid.jpg’...
                                      7f:d0:e8:83:2e:f3:7a:05:f5:5c:52:
b1:0e:a6:44:4e:9b:a9:1d:9a:d0:84:f2:91:a8:db:72:b1:40:c7:f8:a6:9b:65:
73:6c:16:0b:b1:11:ac:3e:f9:3d:c3:6e:6a:9a:1b:2e:ef:17:52:2e:32:e8:71:
1f:29:41:be:80:72:bb:46:c8:7f:fa:d6:a2:04:3d:80:54:dc:a8:e7:9a:3b:a5:
36:79:c8:8b:41:dd:00:76:9a:59:cf:88:19:1f:26:f
total hash length: 191 digits
```

Fig. 5. Hash value of a segment. The `bbHash` of the whole file is listed in Fig. 4.

4.4 Run Time Performance

The run time performance is the main drawback of `bbHash` which is quite slow compared to other approaches. `ssdeep` needs about 0.15 seconds for processing a 10MiB file where `bbHash` needs about 117 seconds for the same file. This is due to the use of building blocks as external comparison data structures and the computation of their Hamming distance to the currently processed input byte sequence. Recall that at each position i we have to build the Hamming distance of 16 building blocks each with a length of 1024 bits. To receive the Hamming distance we XOR both sequences and count the amount of remaining ones (bitcount($bb_k \oplus BS_i$)). To speed up the counting process, we precomputed the amount of ones for all sequences from 0 to $2^{16}-1$ Bits. Thus we can lookup each 16-Bit-sequence with a complexity of O(1). But since we have $N \cdot \frac{l_{bit}}{16} = 16 \cdot \frac{1024}{16} = 1024$ lookups at each processed byte of the input it is quite slow.

Although the processing in its first implementation is quite slow which results from a straight forward programming, this problem is often discussed in literature and there are several issues for improvements. For instance [17] states that their algorithm finds all locations where the pattern has at most t errors in time $O(L_f\sqrt{t \log t})$. Compared against our algorithm which needs $O(L_f \cdot l)$ time that's a great improvement. As we make use of N building blocks, we have to multiply both run time estimations by N. For a next version we will focus on improving the run time performance.

4.5 Attacks

This section is focusing on attacks with respect to both forensic issues blacklisting and whitelisting. From an attacker's point of view anti-blacklisting/anti-whitelisting can be used to hide information and increase the amount of work for investigators.

Anti-blacklisting means that an active adversary manipulates a file in a way that fuzzy hashing will not identify the files as similar – the hash values are too different. We rate an attack as successful if a human observer cannot see a change between the original and manipulated version (the files look/work as expected). If a file was manipulated successfully then it would not be identified as a known-to-be-bad file and will be categorized as unknown file.

The most obvious idea is to change the triggering whereby the scope of each change depends on the Hamming distance. For instance, at position i the Hamming distance is 450, then an active adversary has two possibilities:

1. He needs to change at least 11 bits in this segment to overcome the threshold t and kick out this trigger sequence from the final hash.
2. He needs to manipulate it in a way that another bb has a closer Hamming distance.

In a worst case each building block has a Hamming distance of 460 and a 'one-bit-change' is enough to manipulate the triggering. In this case an active adversary approximately needs to change $\frac{L_f}{100}$ bits, one bit for each position i. Actually a lot of more changes needs to be done as there are also positions where the Hamming distance is much lower than 460. This is an improvement compared to `sdHash` where it is enough to change exactly one bit in every identified feature. Compared to Kornblum's `ssdeep` this is also a great improvement as [6] states that in most of the cases 10 changes are enough to receive a non-match.

Our improvement is mainly due to the fact that in contrast to `ssdeep` and `sdhash` we do not rely on internal triggering, but on external. However, the use of building blocks results in the bad run time performance as discussed in Sec. 4.4.

Anti-whitelisting means that an active adversary uses a hash value from a whitelist (hashes of known-to-be-good files) and manipulates a file (normally a known-to-be-bad file) that its hash value will be similar to one on the whitelist. Again we rate an attack as successful if a human observer couldn't see a change between the original and manipulated version (the files look/work as expected).

In general this approach is not preimage-resistance as it is possible to create files for a given signature: generate valid trigger sequences for each building block and add some zero-strings in between.

The manipulation of a specific file to a given hash value should also be possible but will result in an useless file. In a first step an active adversary has to remove all existing trigger sequences (result in approximately $\frac{L_f}{100}$). Second, he needs to imitate the triggering behavior of the white-listed file which will cause a lot of more changes.

5 Conclusion & Future Work

We have discussed in the paper at hand a new approach for fuzzy hashing. Our main contribution is a tool called `bbHash` that is more robust against anti-blacklisting than `ssdeep` and `sdHash` due to the use of external building blocks. The final signature has about 0.5% of the original file size, is not fixed and can be adjusted by several parameters. This allows us to compare very small parts of a file against its origin which could arise due to file fragmentation and deletion.

In general there are two next steps. On the one side the run time performance needs to be improved wherefore we will use existing approaches (e.g. given in [17]). On the other side we have to do a security analysis for our approach to give more details about possible attacks. Knowing attacks also allows further improvements of `bbHash`.

References

1. C. Walter, "Kryder's law," 2005. [Online]. Available: http://www.scientificamerican.com/article.cfm?id=kryders-law&ref=sciam
2. NIST, "National Software Reference Library," August 2011. [Online]. Available: http://www.nsrl.nist.gov
3. S. Maddodi, G. Attigeri, and A. Karunakar, "Data Deduplication Techniques and Analysis," in *Emerging Trends in Engineering and Technology (ICETET)*, nov. 2010, pp. 664–668.
4. M. Turk and A. Pentland, "Face recognition using eigenfaces," in *Computer Vision and Pattern Recognition, 1991. IEEE*, jun 1991, pp. 586 –591.
5. J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," in *Digital Investigation*, vol. 3S, 2006, pp. 91–97. [Online]. Available: http://www.dfrws.org/2006/proceedings/12-Kornblum.pdf
6. H. Baier and F. Breitinger, "Security Aspects of Piecewise Hashing in Computer Forensics," *IT Security Incident Management & IT Forensics*, pp. 21–36, May 2011.
7. V. Roussev, "Building a Better Similarity Trap with Statistically Improbable Features," *42nd Hawaii International Conference on System Sciences*, vol. 0, pp. 1–10, 2009.
8. ——, "Data fingerprinting with similarity digests," *Internation Federation for Information Processing*, vol. 337/2010, pp. 207–226, 2010.
9. A. Menezes, P. Oorschot, and S. Vanstone, *Handbook of Applied Cryptography.* CRC Press, 1997.
10. A. Tridgell, "Spamsum," Readme, 2002. [Online]. Available: http://samba.org/ftp/unpacked/junkcode/spamsum/README
11. L. Chen and G. Wang, "An Efficient Piecewise Hashing Method for Computer Forensics," *Workshop on Knowledge Discovery and Data Mining*, pp. 635–638, 2008.
12. V. Roussev, G. G. Richard, and L. Marziale, "Multi-resolution similarity hashing," *Digital Investigation 4S*, pp. 105–113, 2007.
13. K. Seo, K. Lim, J. Choi, K. Chang, and S. Lee, "Detecting Similar Files Based on Hash and Statistical Analysis for Digital Forensic Investigation," *Computer Science and its Applications (CSA '09)*, pp. 1–6, December 2009.
14. F. Breitinger, "Security Aspects of Fuzzy Hashing," Master's thesis, Hochschule Darmstadt, February 2011. [Online]. Available: https://www.fbi.h-da.de/organisation/fachgruppen/fachgruppe-it-sicherheit/studien-und-abschlussarbeiten.html
15. C. Sadowski and G. Levin, "Simhash: Hash-based similarity detection," http://simhash.googlecode.com/svn/ trunk/paper/SimHashWithBib.pdf, December 2007.

16. V. Roussev, Y. Chen, T. Bourg, and G. G. Rechard, “md5bloom: Forensic filesystem hashing revisited,” *Digital Investigation 3S*, pp. 82–90, 2006.
17. A. Amir, M. Lewenstein, and E. Porat, “Faster algorithms for string matching with k mismatches,” in *11th annual ACM-SIAM symposium on Discrete algorithms*, ser. SODA ’00. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2000, pp. 794–803. [Online]. Available: http://dl.acm.org/citation.cfm?id=338219.338641