Performance Issues about Context-Triggered Piecewise Hashing

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. A hash function is a well-known method in computer science to map arbitrary large data to bit strings of a fixed short length. This property is used in computer forensics to identify known files on base of their hash value. As of today, in a pre-step process hash values of files are generated and stored in a database; typically a cryptographic hash function like MD5 or SHA-1 is used. Later the investigator computes hash values of files, which he finds on a storage medium, and performs look ups in his database. Due to security properties of cryptographic hash functions, they can not be used to identify similar files. Therefore Jesse Kornblum proposed a similarity preserving hash function to identify similar files. This paper discusses the efficiency of Kornblum's approach. We present some enhancements that increase the performance of his algorithm by 55% if applied to a real life scenario. Furthermore, we discuss some characteristics of a sample Windows XP system, which are relevant for the performance of Kornblum's approach.

Key words: Digital forensics techniques and tools, context-triggered piecewise hash functions, fuzzy-hashing, efficiency of `ssdeep`, subtleties of fuzzy-hashing.

1 Introduction

The amount of data gathered within a computer forensic acquisition process is growing rapidly. As of today, an investigator has to deal with several terabytes of raw data. His crucial task is to distinguish relevant from non-relevant information, which often resembles to look for a needle in a haystack. In most of the cases there is an automated preprocessing, which tries to filter out some irrelevant files to reduce the amount of data the investigator has to look at by hand.

This preprocessing groups the files into three categories: known-to-be-good, known-to-be-bad and unknown files. For instance, system files of the operating system or binaries of a common application like a browser are said to be known-to-be-good and need not be inspected within an investigation. The working steps are quite simple: hash the file, compare the resulting fingerprint against a set of fingerprints and put it in one of the categories. The most common set/database of

non-relevant files is the *National Software Reference Library* (NSRL, [1]) maintained by the US National Institute of Standards and Technology (NIST).

For use in practice an important property of a hash function is ease of computation. Differences in terms of computation complexity have significant effort on the practical usefulness of hash functions. Although the preprocessing is done with computational power, it is very time-consuming. For instance, we consider an installation including operating system files and some applications (e.g. email client, browser, office). Typically such an installation comprises tens of thousands of files. We compare the resulting slowdown of the preprocessing time if two different hash functions of different performance are used. In our example, h_2 denotes a hash function being 25% slower than a hash function h_1. For example, if h_1 can process a file of our sample image on average in $50ms$, then h_2 requires on average $62.5ms$ for processing a file. If there are $50,000$ files of the operating system and $100,000$ personal files like pictures, mp3s, we have to preprocess $150,000$ files. The total runtime of the preprocessing using h_1 requires

$$150,000 \cdot 50ms = 7,500s = 2h\ 5min\ , \tag{1}$$

the corresponding total runtime of h_2 is

$$150,000 \cdot 62.5ms = 9,375s = 2h\ 36min\ . \tag{2}$$

Thus the generation of all hash values will differ about half an hour. Of course, the real processing time depends on the file size, hard disk speed and processor.

Due to the security requirements of cryptographic hash functions, this proceeding has one drawback: if a single bit in the input changes, the output behaves pseudo randomly. Thus comparing the similarity of files using a cryptographic hash function is not possible.

Therefore Jesse Kornblum [2] proposed in 2006 a method, which he calls context-triggered piecewise hashing, abbreviated as CTPH. Kornblum's CTPH approach is based on a spam detection algorithm due to Andrew Tridgell [3]. The main idea is to compute cryptographic hashes not over the whole file, but over parts of the file, which are called *segments* or *chunks*. CTPH is currently the only opportunity for similarity hashing and thus the only opportunity to find similar files based on hash values. In the following we denote a similarity preserving hash function a *fuzzy hash function*. Hence, we rate CTPH as one possible implementation for fuzzy-hashing.

Over the last years Kornblum's idea about fuzzy-hashing has been investigated several times with respect to both performance (e.g. [4], [5]) and security (e.g. [6]). The performance improvements often come with some drawbacks, e.g. [5] increased the performance by decreasing the flexibility and [4] increased the security by decreasing the performance.

1.1 Contributions and Organisation of this Paper

Currently context-triggered piecewise hashing is one of the few approaches for fuzzy-hashing, i.e. the forensic investigator is able to find similar files based

on fingerprints on the byte level (we do not consider semantic fingerprinting). We present our performance improvement of 55% of CTPH with respect to efficiency in Sec. 5.2 within a real-life scenario. In contrast to previously proposed enhancements (see Sec. 5.1), our improvement is independent of the underlying hash functions. For instance, it is possible to replace the used hash functions by other/cryptographic ones as it is done in [4]. Besides the efficiency improvement we also present a way to identify manipulated files, which could be the act of an active adversary. In addition we show some general characteristics about a common operating system and its files, which are important in the context of finding similar files.

The rest of the paper is organised as follows: In the subsequent Sec. 1.2 we introduce notation and terms, which we use throughout this paper. Then, in Sec. 2 we shortly review related work. In Sec. 3 we sketch the current use of hash functions within computer forensics. Next, we discuss in Sec. 4 the foundations of context-triggered piecewise hashing, which are necessary to understand our performance improvement. The core of our paper is then given in Sec. 5, where we present the issues of our improvement, which will be evaluated in Sec. 6. Sec. 7 concludes our paper.

1.2 Notation and Terms used in this Paper

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

- h denotes a cryptographic hash function (e.g. MD5, SHA-1, RIPEMD-160).
- BS denotes a byte string of length m: $BS = B_0B_1B_2 \cdots B_{m-1}$
- bs denotes a bit string of length M: $bs = b_0b_1b_2 \cdots b_{M-1}$
- PRF refers to a pseudo-random function. Kornblum denotes this function as a rolling hash function.
- A *chunk* is a sequence of bytes, for which a hash character is computed (i.e. the byte string between two trigger points).
- A *trigger point* is the final byte within a chunk.
- A *trigger sequence* is a sequence of bytes BS, where $PRF(BS)$ hits a certain value, the *trigger value*. The default length of a trigger sequence are 7 bytes.
- A *block size* is a modulus used to determine trigger sequences. Block sizes are of the form $b_{min} \cdot 2^k$ with a minimal block size b_{min} (typically $b_{min} = 3$) and a non-negative integer k. b denotes the block size.

2 Related Work

Considered quite rough, there are currently two mechanisms for so called fuzzy-hashing, which are based on the same design: Split a file into pieces/chunks, hash each chunk and combine all chunk hashes to the final fuzzy-hash. These approaches differ in terms of fixed vs. variable chunk sizes. In 2002 Nicholas Harbour developed `dcfldd`[1], which extends the well-known disk dump tool `dd`.

[1] `http://dcfldd.sourceforge.net`; visited 31.05.2011

[7] presents a tool called `md5bloom`, which is based on MD5 and bloom filters. It can be used to compare whole hard disks on the block level.

A pioneer for chunks of variable size is Jesse Kornblum with his `ssdeep` ([2]), where the file is divided into chunks by a sliding hash function that is only based on the seven current bytes. A similar idea is given in [8] where a file is divided based on "identifying statistically-improbable features".

Having a closer look at Kornblum's approaches, there are few papers about the security of `ssdeep`. [4] changed the used hash functions and extended it by bloom filters to represent the signature, which increases the security, but decreases the performance. [9] combined it with other statistical file characteristics and improved the false positive rate, but at the cost of efficiency. [5] is the only one dealing with efficiency, but binds `ssdeep` to a homomorphic hash function. More details about related improvements to our work are given in Sec. 5.1.

3 The Usage of Hash Functions in Computer Forensics

In this section we give an overview of the usage of hash functions in computer forensics. First, we describe in Sec. 3.1 the use of cryptographic hash functions. Up to now this is the most common usage. Next, we give a short introduction to context-triggered piecewise hashing (CTPH) in Sec. 3.2.

3.1 Cryptographic Hash Functions and their Application in Computer Forensics

This section introduces the term of a cryptographic hash function, the basic properties of such a function, and their use in computer forensics in the context of a whitelist and a blacklist, respectively.

Let $\{0,1\}^*$ denote the set of bit strings of arbitrary length, and let $bs \in \{0,1\}^*$. If we write h for a hash function then according to [10], h is a function with two properties:

- *Compression*: $h : \{0,1\}^* \longrightarrow \{0,1\}^n$, $n \in \mathbb{N}$ (e.g. $n = 160$).
- *Ease of computation*: Computation of $h(bs)$ is 'fast' in practice.

In practice bs is a 'document' (e.g. a file, a volume, a device). The output of the function $h(bs)$ is referred to as a *hash value*, *digest* or *fingerprint*. Sample security applications of hash functions comprise storage of passwords (e.g. on Linux systems), electronic signatures (both MACs and asymmetric signatures), and whitelists / blacklists in computer forensics.

As of today the most popular use case of cryptographic hash functions within computer forensics is detecting known files. In order to detect these files based on their fingerprints, the computer forensic investigator must have a database at hand, which comprises at least a referrer to the input file and its hash value. If he finds this hash value on a storage medium within an investigation, he is convinced that the referred file is present on the medium. In computer forensics

hash values are typically computed over the payload of a file (i.e. hash functions are applied on the file level). Hence known files can be identified very efficiently.

Dependent on the assessment of the file, he proceeds as follows:

1. *Whitelist*: If the file is known-to-be-good, the investigator can fade out the file from further investigation. The hash database is then referred to be a *whitelist*. Whitelists are used in computer forensics to get data reduction, i.e. only files, which are not on the whitelist, are inspected by hand. We denote the use of a whitelist within computer forensics as *whitelisting*.
2. *Blacklist*: If the file is known-to-be-bad, the investigator looks at the file by hand and checks, if the file actually serves as evidence for e.g. possession of child abuse pictures. The hash database is then referred to be a *blacklist*. We denote the use of a blacklist within computer forensics as *blacklisting*.

3.2 Context-Triggered Piecewise Hashing

The origin of context-triggered piecewise hashing, within the meaning of homologous files, was made in 1999 by Andrew Tridgell and his `rsync`-checksum. This algorithm used context-triggered piecewise hashing to more efficiently find updates of files (e.g. during a backup process). Later Tridgell developed a context-triggered piecewise hashing based algorithm to identify mails, which are similar to known spam mails. He called his software `spamsum` (see [11]).

Jesse Kornblum modified spamsum to cope with files and released `ssdeep`[2] in 2006 [2]. He calls his approach *Context-Triggered Piecewise Hashing (CTPH)*. We discuss Kornblum's algorithm in detail in Sec. 4.

Up to now CTPH is promoted to be able to detect similar files on the byte level. Probably the most common use case for context-triggered piecewise hashing in the forensic process is the use of context-triggered piecewise hashing within *blacklists*.

4 Foundations of CTPH

This section introduces the concept of context-triggered piecewise hashing (CTPH) as proposed by Jesse Kornblum [2] in 2006. We summarise the properties of Kornblum's approach that are relevant for understanding the remainder of this paper.

As mentioned above, the origin of Kornblum's idea goes back to Andrew Trigdell's spamsum algorithm [11]. Unlike `dcfldd` the blocks are not fixed-sized and will be denoted as *chunk* or *segment*. Each chunk is determined by a pseudo random function PRF as follows: A window of a fixed size s (we assume $s = 7$ bytes throughout this paper) moves through the whole input, byte for byte, and generates a pseudo random number at each step. Let

$$BS_p = B_{p-s+1}B_{p-s+2}B_p \tag{3}$$

[2] `http://ssdeep.sourceforge.net`; visited 30.12.2010

Algorithm 1 Pseudocode of the rolling hash

h_1, h_2, h_3, c, n are unsigned 32-bit integers, initialised to zero
window is an array of length *size* (default $size = 7$)

to update the rolling hash for a byte c

```
h2 = h2 − h1
h2 = h2 + size · c                    ▷ h2 is the sum of the bytes times the index
h1 = h1 + c
h1 = h1− windows[n mod size]          ▷ h1 is the sum of the bytes in the window
window [n mod size] = c
n = n + 1
h3 = h3 << 5                          ▷ h3 mostly needed to cope with large block sizes values
h3 = h3 ⊕ c
return (h1 + h2 + h3)
```

denote the byte sequence in the current window of size s at position p within the file and let $PRF(BS_p)$ be the corresponding rolling hash value. If $PRF(BS_p)$ hits a certain value, the end of the current chunk is identified. We call the byte B_p a *trigger point* and the current byte sequence BS_p a *trigger sequence*. The subsequent chunk starts at byte B_{p+1} and ends at the next trigger point or EOF.

Kornblum denotes the PRF as a rolling hash. The structure of the rolling hash function as proposed in [2] allows to compute the value $PRF(BS_{p+1})$ cheaply from the previous rolling hash value $PRF(BS_p)$. If B_p is not a trigger point, the next processed byte sequence is $BS_{p+1} = B_{p-s+2}B_{p-s+3}B_{p+1}$. Kornblum updates the value $PRF(BS_{p+1})$ by removing the influence of B_{p-s+1} and considering the new byte B_{p+1}. Algorithm 1 shows the pseudocode of the rolling hash used by Kornblum in `ssdeep`. As there are only low-level operations, Kornblum's PRF is very fast in practice.

In order to define a hit for $PRF(BS_p)$, Kornblum introduces a modulus, which he calls a *block size*. If b denotes the block size, then the byte B_p is a trigger point if and only if $PRF(BS_p) \equiv -1 \bmod b$. If PRF outputs equally distributed values, then the probability of a hit is reciprocally proportional to b. Thus if b is too small, we have too many trigger points and vice-versa. As Kornblum aims at having 64 chunks, the block size has to be approximately $b_{init} \approx \frac{N}{S}$ where S is the desired number of chunks with a default value of 64, and N is the file size in bytes. To receive an equal block size for similar sized files he generates the initial block size b_{init} as follows:

$$b_{init} = b_{min} \cdot 2^{\lfloor \log_2(\frac{N}{S \cdot b_{min}}) \rfloor} , \tag{4}$$

where the minimum block size b_{min} is set to 3. Although we will not discuss this formula, it should be mentioned, with respect to Kornblum's work, that the calculation of the block size as given in [2], Eq. (4) is not conform with his implementation [12] (floor vs. ceiling operation in the exponent),

$$b_{init} = b_{min} \cdot 2^{\lceil \log_2(\frac{N}{S \cdot b_{min}}) \rceil} . \tag{5}$$

```
$ dd if=/dev/urandom of=random bs=1 count=24000

$ ls -la random
-rw-r--r-- 1 user user 24000 2011-06-01 07:58 random

$ ssdeep random
ssdeep,1.0--blocksize:hash:hash,filename
384:exQOElbn4N0TSNVyCCvIiebjYKKjoKUTDeueZdTmvk1ac9slOjRXMImRHgnY5:
   A8ZQVy6jYKKElPeXZdT1NaHJ5,"/home/user/random"
```

Fig. 1. A sample ssdeep output

Once a chunk is identified a cryptographic hash value over this chunk is computed. Let BS denote this chunk and h the cryptographic hash function. Then $h(BS)$ is a bit string of length n. However, to save space, Kornblum only makes use of the least significant 6 bits of $h(BS)$, i.e. the 6 rightmost bits. We denote this output by $LS6B(h(BS))$. Kornblum then identifies $LS6B(h(BS))$ with a Base64 character. We refer to this Base64 character as the *Base64 hash character* for the currently processed chunk. Kornblum's hash value for a file is simply the concatenation of all Base64 hash characters.

Since the block size is used for determining the chunks and depends on the length of the input, only `ssdeep` hash values with the same block size can be compared. To be a little bit more flexible two different block sizes are used: b_{init} and $2b_{init}$. If there are too few Base64 hash characters for block size b_{init} (i.e. at most $\frac{S}{2} - 1 = \frac{64}{2} - 1 = 31$), Kornblum sets $b \leftarrow \frac{b_{init}}{2}$ and the whole process is repeated.

A sample output of `ssdeep` is given in Fig. 1. The `ssdeep` hash is computed over the file `random` with its $24,000$ bytes. The number at the beginning of the output of `ssdeep` is the block size used to trigger the PRF. In our example the block size is 384, which can be computed using Eq. (4) or estimated by $\frac{24,000}{64} = 375$. Then the two `ssdeep` hash values comprising the Base64 hash characters for block size 384 and $2 \cdot 384 = 768$ are printed, respectively. Finally, we see the path and name of the processed file.

5 Performance Improvement of CTPH

This section discusses the performance of `ssdeep`, where performance means efficiency. [13] is an online article about Kryder's Law (based on Moore's Law) where it is written that "the doubling of processor speed every 18 months is a snail's pace compared with rising hard-disk capacity". This fact also influences our willingness to delete something from our disc. To process this huge amount of data we need fast hardware and software.

As mentioned in the introduction, there are several ideas to enhance `ssdeep` with respect to both efficiency (e.g. [5]) and security (e.g. [4]). In addition there was a security analysis of CTPH in [6].

Sec. 5.1 will explain the existing improvements in detail followed by our new idea in Sec. 5.2.

5.1 Existing Improvements

In [4] the authors applied two main changes. First, they used a new PRF and second, they changed the hash function for processing each chunk. Instead of using the exiting PRF based on adler32, Roussev et al. used another simple polynomial hash function called `djb2` which is defined as follows:

$$h_0 = 5381, \quad h_{k+1} = 33h_k + c_k \bmod 2^{32}, \text{ for } k \geq 0\ , \tag{6}$$

where c_k denotes the k^{th} character of the input. In his paper Roussev shows that `djb2` and `MD5` yield similar results concerning the chunk sizes. Both hash functions were tested against several files and their basic stochastic values have been compared. As a result Roussev states that it is not necessary to use a cryptographic hash function for the PRF. But `djb2` has the disadvantage compared to original one that each window has to be processed from scratch. Remember, the original PRF allows to compute the value for the following window by removing the influence of the last character and add the new one, in contrast to `djb2`, where we need a loop, which processes all characters[3]. So his changes influence the efficiency in a negative way.

Furthermore, Roussev used the cryptographic hash function MD5 for processing each chunk instead of the FNV-Hash. The least significant 11 bits of the MD5 hash serve as input for a Bloom filter to represent the final signature. Even though this modification slows down[4] `ssdeep`, it increases the security aspects (e.g. difficulty to find collisions or second-preimages). Hence we consider this change to be very useful.

On the efficiency side [5] showed that it is possible to improve the straight forward processing of `ssdeep`. Once again, `ssdeep` reads the file byte by byte, generates the PRF and the FNV-Hash and in the very last step it examines the signature. If it is too short, i.e. shorter than 32 characters, everything is dropped and the file is processed again using an adapted block size $b \leftarrow \frac{b}{2}$. [5] showed[5] that this happens in 38% of cases and therefore modified `ssdeep` "to generate intermediate hashes using numbers in the geometric progression with factor 4 as block size, generate hashes with other block sizes used in spamsum by rehashing the intermediate hashes to decrease the scan passes and hashing passes. [...] With current block size b, we compute two traditional hashes h and H at block level b and $4b$" and count the trigger points of the block sizes b, $2b$, $4b$ and $8b$. Furthermore, the authors stated that they used FNV as a homomorphic hash function and that it is possible to create the hashes for $2b$ by using the hashes of b. An example is given in Fig. 2. Block size b is the smallest one and

[3] The multiplication $33h_k$ can be implemented by $(h_k << 5) + h_k$. Therefore all previous characters influence h_{k+1} and an iterative processing is not possible

[4] Because FNV only requires one multiplication and one XOR, it is faster than MD5.

[5] It's the result of a test with 1575 different files from Linux and Windows systems.

therefore triggers most often. Having a look at the chunks for $2b$, they consist of the first three chunks of b, which are represented by the FNV-hashes $h1$, $h2$ and $h3$. If we have a homomorphic hash function,we can create the chunk hash for $2b$ by combining $h1$ to $h3$. Actually, we don't think that FNV could be used in such a way.

If the final signature for b_{init} is too short than b is set to $\frac{b_{init}}{4}$ and h is set to H. Now only H needs to be computed again.

recorded block size	b	b	2b	4b	b	8b	4b	2b
pieces (b)	h1	h2	h3	h4	h5	h6	h7	h8
pieces (2b)	1			2	3		4	5
pieces (4b)	H1				H2		H3	H4
pieces (8b)	1						2	

Fig. 2. hash signature generation process of [5]

The drawback of this approach is that a combination of hashes, i.e. use the hashes for b and combine them to hashes for $2b$ might be only possible with FNV-Hash. If we use any cryptographic hash function such as MD5, we have to do more computations and lose the efficiency advantage.

5.2 Our Enhancements of `ssdeep`

Our improvement aims at three main points:

1. Each file should be processed only once and thus we have a runtime directly proportional to the input length.
2. The implementation should be flexible so that we can change the PRF and chunk hash function h.
3. It should be able to determine an untypical behaviour of trigger sequences, which may be caused by an active adversary.

Our main idea is to process the file and count the trigger sequences for all reasonable block sizes (the term *reasonable* is explained below). In the next step we read the file again and set the block size b to the largest value that yields at least 32 signature characters. Further details about the enhancement are given below.

An important point with respect to security is the restriction of the signature length of `ssdeep`. Kornblum asserts $32 \leq length \leq 64$. However, he does not give any justification. We deem the lower boundary to be useful in order to be

able to make statements about similarity. However, the upper boundary is a weakness and was exploited in [6]. Maybe a reason to set this boundary was to satisfy the compression conditions for hash functions. Nevertheless some attacks are not possible if we ignore this condition.

Implementation Details We use the original software for our improvement and insert our ideas. This means that all shown performance improvements are only based on some algorithm changes and not on some implementation issues.

We consider a reasonable block size to be of order of magnitude of Kornblum's proposal for b_{init}. To be more concrete, we allow $b \in \{2b_{init}, b_{init}, b_{init}/2, b_{init}/4\}$. The file is read byte by byte and the PRF is computed for each byte. If we found a trigger sequence for one of the four reasonable block sizes, we save the offset and increase a counter for this block size. As a reminder a trigger sequence is found, if $PRF(BS) \equiv b-1 \bmod b$. In most of the cases we only have to check one if-condition as BS can only be a trigger sequence for b if it is a trigger sequence for $b/2$, too.

For the second run the file is read again byte by byte. Now we use the stored offsets to determine each chunk and run the chunk hash function h. As a result we preserve the flexibility to change PRF and the hash function.

Untypical Behaviour of trigger sequences In [6] the authors demonstrated that an active adversary can manipulate a file and bypass blacklisting and whitelisting. For this the authors exploited a peculiarity that an `ssdeep` signature can have at most 64 characters. The attack randomly generates trigger sequences and inserts them at the beginning of the file.

Generally, we would expect that no file has significantly more than 64 trigger sequences for their initial block size b_{init}. In general there are two extreme examples:

- Low entropy files: They are expected to have long runs of a specific byte, e.g. 0-byte-sequences. As this will not cause a triggering, there will be too less trigger sequences instead of too much. Example files are *.doc or *.bitmap.
- High entropy files: They are expected to have very variable byte-sequence. A well-tested PRF is assumed to produce approximately 64 trigger sequences. Example files are *.jpg, *.zip or a truecrypt container.

A possible manipulation can be detected by comparing the trigger sequences counter against a certain threshold. Of course, this mechanism could be improved by considering the distribution of the trigger sequences, i.e. we would expect that all chunks have a similar length.

Assessment of our improvement As it is described above, the modification needs to read the file two times: one run for receiving the amount of trigger sequences including their offsets and one run for building the hash value for each junk. As the amount of computations, i.e. building the PRF and FNV-Hash, is similar in both `ssdeep` versions, there should be no significant difference. The main disadvantage is that our proposal reads the file two times from the hard disk / cache / RAM.

On the other side, the new algorithm is superior if there are not enough trigger sequences. Thus, the original version needs to do two complete runs, which means: reading the file from the hard disk, generate the PRF, compute h for each identified chunk. If this is the case, we expect the new version to be faster. As then both algorithms needs to read the file from the hard disk two times, we expect to have an improvement over 50%.

If the file has even less trigger sequences (e.g. the block size needs to be quartered) and therefore needs to be read more often, we expect a great performance difference between both algorithms: $b \leftarrow \frac{b_{init}}{4}$ results in 33% runtime, $b \leftarrow \frac{b_{init}}{8}$ in 25% runtime.

Thus, two new questions raise up, which will be answered in the next section:

- How often does `ssdeep` on average adapt the block size?
- What is an average size for all files on a hard disk?

6 Experimental Results

In this section we discuss our experimental setup to show the efficiency advantage of our modified `ssdeep` version. First, in Sec. 6.1 we describe our test environment and some specific characteristics of a our sample operating system. Then in Sec. 6.2 we present the practical relevance of our enhancement on base of a 500 MiB file. Finally, in Sec. 6.3 we discuss the performance advantage of our enhancement with respect to a real-life scenario.

6.1 Working Environment and File System Analysis

In order to receive trustful results, which mirror a real-life scenario, we set up a system running Windows XP Service Pack 3 including some basic applications and user specific files. More precisely, we assume that nowadays nearly every personal computer has at least an office suite, a browser and a PDF-Viewer installed. We therefore set up OpenOffice 3.3, Mozilla Firefox 4.01 and Acrobat Reader 10.0.1. In order to work on some user specific files, we additionally insert about

- $1,000$ images of size of a few KB to 1 MB,
- 20 MP3 audio files covering music from Strauß and Bach of size of 4 to 11 MB, and
- several free available PDF files.

Even though this only reflects a very small system, it allows estimations what happens if a hard disc image would have about 200 GB or more.

Our sample image has a total file size of 3.84 GB, containing $15,036$ files and $1,574$ folders. It is stored as a `dd`-image. Our working environment is based on Ubuntu 10.04 Linux installed in a virtual machine.

Amount and Size of Files Our sample image comprises 15,036 files and 3,936,437,740 bytes, which leads to an average file size of

$$\frac{3,936,437,740 \text{ bytes}}{15,036 \text{ files}} = 261,800.86 \frac{\text{bytes}}{\text{file}} = 255.7 \frac{\text{KiB}}{\text{file}} . \tag{7}$$

Assuming a modern work station with 4 GiB main memory, we easily can save our `dd` image into RAM. Therefore the disadvantage that a file needs to be read two times from a persistent storage medium can be neglected.

Block Size Changes We use the previously declared system for our efficiency analysis of the original `ssdeep` implementation. As already investigated by [5] we expect that there is a change of the block size within the `ssdeep` processing in 38% of the cases. The results of our sample test environment of 15,036 files are given in Table 1.

1 time	2 times	3 times	4 times	5 times	6 times and more
10,125	3,944	645	176	32	114
67.3 %	26.2 %	4.3%	1.3 %	0.2 %	0.8%

Table 1. Distribution of block size changes

Our test image has the property that 10,125 files only need to be processed once, which means that approximately 33% needs to be processed more than one time. This is in conformance with the claim given in [5].

However, if we examine the relationship between the file size and the amount of trigger sequences, we find an interesting relationship. The files, which only need one run (i.e. each of these files has enough trigger sequences) have a total size of 1,302,435,802 bytes (i.e. an average file size of 127 KiB). On the other hand all the rest has a total size of 2,634,001,938 bytes (i.e. an average file size of 535 KiB and thus about 4 times larger than one-processed files). In general we can say that mostly large files need more than one run. One possible answer would be that large files, with some exceptions, often contain long 0-byte-sequences, which do not trigger the PRF. However, this statement needs some more research.

6.2 Efficiency Improvement

Next, we have a first practical test of our performance assumptions from Sec. 5 and if they are true. We compare the original `ssdeep` version to our modified one for three different files of size 500 MiB, respectively, where

– file1 has enough trigger sequences for b_{init},
– file2 has only enough trigger sequences for $b_{init}/2$, but not for b_{init}, and

– file3 has only enough trigger sequences for $b_{init}/4$, but for no larger block size.

The results of our performance comparison are given in Table 2. The time has been measured using the Linux command `time -p` (the flag `-p` is used to obtain an output containing the three different times[6]). Explanations from the man page of time say:

– user: Total number of CPU-seconds that the process used directly (in user mode), in seconds (e.g. running through an array and do something).
– sys: Total number of CPU-seconds used by the system on behalf of the process (in kernel mode), in seconds (e.g. reading a file from the hard disc).

Additionally, in Table 2 we add both timings to get the overall CPU time denoted by sum.

	500 MiB $b \leftarrow b_{init}$	500 MiB $b \leftarrow \frac{b_{init}}{2}$	500 MiB $b \leftarrow \frac{b_{init}}{4}$
original `ssdeep`	user: 2.76 sys: 4.89 sum: 7.65	user: 5.45 sys: 10.06 sum: 15.51	user: 7.53 sys: 15.47 sum: 23.00
modified `ssdeep`	user: 1.50 sys: 7.60 sum: 9.10	user: 1.46 sys: 7.58 sum: 9.04	user: 1.74 sys: 7.52 sum: 9.26
$\frac{\text{modified sum}}{\text{original sum}}$	1.19	0.58	0.40
estimated $\frac{\text{modified sum}}{\text{original sum}}$	1.00	0.50	0.33

Table 2. CPU time to process different 500MiB files with `ssdeep` and our modified improved version.

The *user*-time results essentially from processing the input, e.g. generating the PRF and the FNV-Hash. In contrast the *sys*-time is mostly influenced by reading/buffering the file. It is eye-catching that the *sys*-time for the modified version (7.60 sec.) is faster than 2 times the original one ($2 \cdot 4.89$ sec.). Our guess is that this results from the caching mechanism of modern systems.

Furthermore, it is surprising that the *user*-time differs so much. Due to our implementation change, we need to do less computations in our modified version, e.g. do not generate all trigger sequences, have less if-conditions to check.

Having a look at the results for the original version, we recognize that there is a linear increase; a halving block size increases the runtime by exactly one run. The modified version has a constant run time, but is therefore approximately 20% slower compared to the original version, if there is only one run. In addition we expected an improvement of 50% if the block size $b \leftarrow \frac{b_{init}}{2}$ is used; the practical relation is 58% in our test.

[6] As we assume the 'real-time' for useless, we drop it.

Overall we can say that the improvement depends on the files we investigate, which will be discussed in the next section.

6.3 Performance Comparison Using our Sample 4 GB Image

In this section we concentrate on a real-world scenario and compare the runtimes of both `ssdeep` versions, the original and our modified one. Our performance test makes use of our small disc image from Sec. 6.1. For both versions we perform two runs. Additionally, as a benchmark we measure the run time to compute the SHA-1 values of all files using `sha1sum`. As `sha1sum` does not have a recursive flag, we used the following command:

```
find /my/path -type f -print0 | xargs -0 sha1sum
```

The results are given in Table 3.

	first run	second run
original `ssdeep`	user: 202.96 sys: 52.01 sum: 254.97	user: 201.94 sys: 56.02 sum: 257.96
modified `ssdeep`	user: 88.58 sys: 26.29 sum: 114.87	user: 86.04 sys: 29.19 sum: 117.23
$\frac{\text{modified sum}}{\text{original sum}}$	0.45	0.45
sha1sum	user: 26.98 sys: 12.96 sum: 39.94	user: 26.57 sys: 13.46 sum: 40.04

Table 3. Run times (in seconds) to compute hash values of all files on our sample image

Comparing the runtime of our modified `ssdeep` version to the original one yields an improvement of approximately 55% for a complete image (because the runtime of the modified version is only about 45% of the original version). Looking back at section 6.1 we can say that about 2.6 GB needs to be processed more than one time which explains the large performance improvement.

However, compared to the cryptographic hash function SHA-1, both `ssdeep` versions are severely slower.

7 Conclusion

Currently context-triggered piecewise hashing is the only approach for fuzzy-hashingFuzzy-hashing might get more important if besides FTK other commercial tools support it. We have discussed the performance issues of CTPH as

proposed by Jesse Kornblum. As a conclusion the performance of CTPH could be easily improved by nearly 55% with respect to a real-life scenario. This efficiency improvement is important for the future since the amount of data is increasing and investigators are overworked. Even though CTPH does not withstand an active adversary, it can support an investigation process and give some first clues, especially if the owner is not a computer specialist.

Additionally, we presented some interesting facts about properties of an operating system and its files with respect to CTPH. For example our sample Windows XP system consists of thousands of small files mostly smaller than 262 KB. Due to the fact that `ssdeep` processes most of the small files only one time, we assume a high entropy and therefore a lot of information within these files.

Acknowledgement: We thank Christian Dichelmüller for providing the sample Windows XP image.

References

1. National Institute of Standards and Technology, "National Software Reference Library," http://www.nsrl.nist.gov, July 2011.
2. 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
3. A. Tridgell, "Spamsum," Readme, 2002. [Online]. Available: http://samba.org/ftp/unpacked/junkcode/spamsum/README
4. V. Roussev, G. G. Richard, and L. Marziale, "Multi-resolution similarity hashing," *Digital Investigation 4S*, pp. 105–113, 2007.
5. L. Chen and G. Wang, "An efficient piecewise hashing method for computer forensics," *Proceedings of the International Workshop on Knowledge Discovery and Data Mining*, pp. 635–638, 2008.
6. H. Baier and F. Breitinger, "Security aspects of piecewise hashing in computer forensics," *6th International Conference on IT Security Incident Management & IT Forensics*, May 2011.
7. V. Roussev, Y. Chen, T. Bourg, and G. G. Rechard, "md5bloom: Forensic filesystem hashing revisited," *Digital Investigation 3S*, pp. 82–90, 2006.
8. V. Roussev, "Data fingerprinting with similarity digests," *Internation Federation for Information Processing*, vol. 337/2010, pp. 207–226, 2010.
9. K. Seo, K. Lim, J. Choi, K. Chang, and S. Lee, "Detecting similar files based on hash and statistical analysis for digital forensic investigation," *2nd International Conference on Computer Science and its Applications, 2009. CSA '09.*, pp. 1–6, December 2009.
10. A. Menezes, P. Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1997.
11. A. Tridgell, "Spamsum," Readme, 2002. [Online]. Available: http://samba.org/ftp/unpacked/junkcode/spamsum/README
12. J. Kornblum, "ssdeep," Sourcecode and Documentation, September 2010. [Online]. Available: http://ssdeep.sourceforge.net/

13. C. Walter, “Kryder’s law.” [Online]. Available: http://www.scientificamerican.com/article.cfm?id=kryders-law&ref=sciam