Security Aspects of Piecewise Hashing in Computer Forensics

Harald Baier
Center for Advanced Security Research Darmstadt
and Hochschule Darmstadt
64295 Darmstadt, Germany
harald.baier@cased.de

Frank Breitinger
Department of Computer Science
Hochschule Darmstadt
64295 Darmstadt, Germany
frank.breitinger@stud.h-da.de

Abstract

Although hash functions are a well-known method in computer science to map arbitrary large data to bit strings of a fixed length, their use in computer forensics is currently very limited. 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. This approach has several drawbacks, which have been sketched in the community, and some alternative approaches have been proposed. The most popular one is due to Jesse Kornblum, who transferred ideas from spam detection to computer forensics in order to identify similar files. However, his proposal lacks a thorough security analysis. It is therefore one aim of the paper at hand to present some possible attack vectors of an active adversary to bypass Kornblum's approach. Furthermore, we present a pseudo random number generator being both more efficient and more random compared to Kornblum's pseudo random number generator.

Keywords

Fuzzy hashing, context-triggered piecewise hash functions, security analysis, whitelisting, blacklisting, computer forensics, anti-forensics.

I. 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. As cryptographic hash functions yield unique fingerprints of arbitrary large input, they are used in computer forensics to identify known data by their fingerprint. 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. Thus their fingerprint is computed in advance and labeled as a non-relevant fingerprint.

While this proceeding is well-established and straightforward, it has one main drawback. Cryptographic hash functions meet several security requirements, which are well-suited for their use in cryptography (e.g. to digitally sign a file). As a consequence the fingerprints of a cryptographic hash function behave pseudo randomly if one bit is changed. Comparing similarity of files using cryptographic hash functions is therefore not possible in that way.

This, however, restricts to the byte level. If a semantic analysis of data is allowed, similarity of contents is often possible. As an example, think of acoustic fingerprinting, where similar sound parts of a music track may be found within a large database to identify the composition ([2]).

However, deciding about similarity of files on the byte level is more efficient. Therefore Jesse Kornblum [3] 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 [4]. The main idea is to compute cryptographic hashes not over the entire file, but over parts of it, which are called *segments* or *chunks*.

The reference implementation of Kornblum's CTPH algorithm is called `ssdeep`. Although Kornblum's paper [3] is about 5 years old, a thorough security analysis is missing. Some minor efficiency improvements have been proposed (e.g. [5]), but there is no research on robustness against anti-forensics of `ssdeep` and its underlying algorithms.

We point out that although we only discuss the relevance of Kornblum's approach with respect to computer forensics, variants of `ssdeep` may be used in biometrics, malware detection, and intrusion prevention, too.

A. Contributions and Organisation of this Paper

While context triggered piecewise hashing seems to be reasonable for detecting similar files, if no anti-forensic measures are present, our main contribution is to show that Kornblum's approach does not withstand an active adversary. Currently, context triggered piecewise hashing is a candidate for so-called blacklisting, i.e. the forensic investigator is able to find files being similar to known-to-be-bad files using CTPH. We show how to circumvent such a blacklist although the non-detected file is similar to a known-to-be-bad file.

Additionally, we improve the pseudo random number generator used within the `ssdeep` implementation of Kornblum with respect to both efficiency and randomness. As a consequence the performance of `ssdeep` is enhanced with respect to both speed and stochastic properties of CTPH.

We stress that although Kornblum denotes his CTPH approach to be a fuzzy hash function, we do not agree with this statement. The main reason is that Kornblum's context triggered piecewise hashing method relies on cryptographic hash functions. Using this design, a fuzzy property will not be reachable. We therefore avoid the term *fuzzy* when dealing with context triggered piecewise hashing.

The rest of the paper is organised as follows: In the subsequent Sec. I-B we introduce notation and terms, which we use throughout this paper. Then, in Sec. II we sketch the current use of hash functions within computer forensics. Next, we discuss in Sec. III the foundations of context triggered piecewise hashing, which are necessary to understand our anti-forensic attacks. In the following Sec. IV we discuss software packages, which provide CTPH functionality. Then in Sec. V we show how to improve Kornblum's pseudo random number generator. The core of our paper is then given in Sec. VI, where we present our anti-forensic attacks to circumvent a CTPH-based blacklist. Sec. VII concludes our paper.

B. 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 is a byte string of length m: $BS = B_0B_1B_2 \cdots B_{m-1}$.
- bs denotes a bit string of length M: $bs = b_0b_1b_2 \cdots b_{M-1}$.
- If two bit strings bs_1 and bs_2 are given, their XOR relationship is written as $bs_1 \oplus bs_2$.
- Where reasonable spaces in strings are denoted by $\llcorner\!\lrcorner$.
- PRTF refers to a pseudo random trigger function. Kornblum calls this function a *rolling hash function*.
- F_1 denotes the original version of a file (i.e. the non-manipulated version).
- F_2 refers to the manipulated version of F_1 within an anti-forensic attack.
- A *chunk* or *segment* is a sequence of bytes within a file, 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 $PRTF(BS)$ 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.

II. 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. II-A the use of cryptographic hash functions. Up to now this is the most common usage. Then we describe in Sec. II-B a block based approach, which remedies some drawbacks of the use of cryptographic hash functions. Finally, we give a short introduction to context triggered piecewise hashing (CTPH) in Sec. II-C.

A. 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 and if n is a positive integer, then according to [6], h is a function with two properties:

- *Compression*: $h : \{0,1\}^* \longrightarrow \{0,1\}^n$.
- *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* or a *digest*. 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.

For use in cryptography, we have to impose further conditions:

1) *Preimage Resistance*: Let a hash value $H \in \{0,1\}^n$ be given. Then it is infeasible **in practice** to find an input (i.e. a bit string bs) with $H = h(bs)$.
2) *Second Preimage Resistance*: Let a bit string $bs_1 \in \{0,1\}^*$ be given. Then it is infeasible **in practice** to find a second bit string bs_2 with $bs_1 \neq bs_2$ and $h(bs_1) = h(bs_2)$.

3) *Collision Resistance*: It is infeasible **in practice** to find any two bit strings $bs_1, bs_2 \in \{0,1\}^*$ with $bs_1 \neq bs_2$ and $h(bs_1) = h(bs_2)$

These security conditions have an important consequence regarding the output of a hash function: Let bs and $h(bs)$ be given. If bs is replaced by bs', $h(bs')$ behaves pseudo randomly, i.e. we do not have any control over the output, if the input is changed. This effect is called *avalanche effect*. As a consequence if only one bit in bs is changed to get bs', the two outputs $h(bs)$ and $h(bs')$ look 'very' different. More precisely every bit in $h(bs')$ changes with probability 50%, independently of the number of different bits in bs'. Sample cryptographic hash functions are given in Table I.

Name	MD5	SHA-1	SHA-256	SHA-512	RIPEMD-160
n	128	160	256	512	160

Table I
SAMPLE CRYPTOGRAPHIC HASH FUNCTIONS AND

As of today the most popular use case of cryptographic hash functions within computer forensics is detecting known files. The idea is quite simple: As cryptographic hash functions behave pseudo randomly, if the input is manipulated, hash values serve as a unique and very short fingerprint of an arbitrary large input bit string. 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.

In order to detect known files on base of their fingerprints, the computer forensic investigator must have access to a database, which comprises at least a referrer to the input file and its hash value. If he finds a match of a hash value of a file within an investigation to a hash value in the database, he is convinced that the referred file is actually present on the storage medium.

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 it actually is illicit (e.g. a child abuse picture). The hash database is then referred to be a *blacklist*.
   We denote the use of a blacklist within computer forensics as *blacklisting*.

As it is a challenging task to generate a capacious whitelist or blacklist, often global databases are used. However, if a file is known-to-be-good or known-to-be-bad, respectively, depends on regional law. Therefore hash databases have to be adapted according to national legal frameworks.

```
"SHA-1","MD5","CRC32","FileName","FileSize","ProductCode","OpSystemCode","SpecialCode"
"AC91EF00F33F12DD491CC91EF00F33F12DD491CA","DC2311FFDC0015FCCC12130FF145DE78","14CCE9061FFDC001",\\
"WORD.EXE",1217654,103,"T4WKS",""
```

Figure 1. A sample entry of the NIST Reference Data Set (RDS)

The most famous basis for generating a whitelist is the *National Software Reference Library* (NSRL, [1]). Within its Reference Data Set (RDS)[1] each file entry comprises the SHA-1 fingerprint (Secure Hash Algorithm 1, [7]), the MD5 fingerprint (Message Digest Algorithm 5, [8]), the CRC-32 checksum, the file name and its length. A sample entry of the RDS for a well-known text editing programme is given in figure 1. NIST points out that its RDS is not a whitelist, as it also contains entries of files, which may be known-to-be-bad in some countries (e.g. steganographic tools, hacking scripts). However, there is no illicit data within the RDS, e.g. child abuse images.

Although this proceeding is well-established and straightforward, it has one main drawback. Due to the avalanche effect of cryptographic hash functions, changing one irrelevant bit circumvents the blacklist although the modified file is almost identical to the known-to-be-bad version.

In addition to the RDS, there are also non-RDS hash datasets, that may be used within a computer forensic investigation. For example there are data sets for SHA-256, 'MD5 of the first 4096 bytes' of a file or entries for *ssdeep algorithm (aka "fuzzy hashes")*[2].

[1]http://www.nsrl.nist.gov; visited 07.01.2011
[2]http://www.nsrl.nist.gov/ssdeep.htm; visited 07.01.2011

B. Block Based Hashing

A first approach to overcome the shortcomings of the use of cryptographic hash functions on the byte level is due to Nicholas Harbour. In 2002 he developed a programme called `dcfldd`[3], which extends the well-known disk dump tool `dd`. At that time, Nicholas Harbour worked for the Defense Computer Forensics Laboratory (DCFL) of the US Department of Defense.

The aim of `dcfldd` is to ensure integrity on the sector level during imaging. The software splits the input data into sectors or blocks of a fixed length (e.g. sectors of size 512 bytes) and computes for each of these blocks the corresponding cryptographic hash value. Thus his approach is also called *block based hashing*. `dcfldd` outputs are rather large: If we use for instance SHA-1 and the default block size of 512 bytes, then every 512 byte block is represented by its corresponding 20 byte SHA-1 value. As a consequence a `dcfldd` hash requires about 4 per cent of the file's memory. For instance, a 4 MiB image is represented by a sequence of SHA-1 hash values of length 163.840 bytes, which is rather long.

A key property of Harbour's method is that a flipping bit in the input only affects the hash output of the corresponding block. However, if we use `dcfldd` in combination with a whitelist or a blacklist, `dcfldd` does not withstand a trivial anti-forensic measure: Shifting the bytes by inserting or deleting a byte in the first block leads to a completely different sequence of hash values. This means that `dcfldd` is not alignment robust, where alignment robustness means [9]: *It is important that shifting does not affect the complete hash value, i.e. resistant against the addition or deletion of bytes.*

C. Context Triggerd Piecewise Hashing

In 1999 Andrew Tridgell invented context triggered piecewise hashing, within the meaning of homologous files, in the scope of his `rsync`-application. 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`.

Jesse Kornblum modified spamsum to cope with files and released `ssdeep`[4] in 2006 [3]. He calls his approach *Context Triggered Piecewise Hashing (CTPH)*. In contrast to `dcfldd` Kornblum divides a file into blocks depending on their content. We discuss Kornblum's algorithm in detail in Sec. III.

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*. However, we will show in Sec. VI how to circumvent CTPH by keeping the semantics of a file.

III. Foundations of CTPH

This section introduces the concept of context triggered piecewise hashing (CTPH) as proposed by Jesse Kornblum [3] in 2006. We summarize 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 [9]. Unlike `dcfldd` the blocks are not fixed-sized and will be denoted as *chunk* or *segment*. Each chunk is determined by a pseudo random trigger function PRTF (Kornblum calls this function a *rolling hash function*). A PRTF gets a byte string as input and outputs an integer. It proceeds 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

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

denote the byte sequence in the current window of size s at position p within the file and let $PRTF(BS_p)$ be the corresponding rolling hash value. If $PRTF(BS_p)$ hits a certain value, the end of the current chunk is identified. We call the byte B_p a *trigger point* or *trigger value* 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`.

Pseudocode of Kornblum's rolling hash function as proposed in [3] is given in Algorithm 1. It allows to compute the value $PRTF(BS_{p+1})$ cheaply from the previous rolling hash value $PRTF(BS_p)$. 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 $PRTF(BS_{p+1})$ by removing the influence of B_{p-s+1} and adding the new byte B_{p+1}. As there are only low-level operations, Kornblum's PRTF is very fast in practice. However, we will show in Sec. V how to improve Kornblum's PRTF.

In order to define a hit for $PRTF(BS_p)$, 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 $PRTF(BS_p) \equiv -1 \bmod b$. If PRTF outputs equally distributed

[3]http://dcfldd.sourceforge.net; visited 04.01.2011

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

Algorithm 1 Pseudocode of the rolling hash

$h1, h2, h3, c$ are unsigned 32-bit values, initialized to zero
$window$ is an array of $size$

to update the rolling hash for a byte c

```
h2 = h2 - h1
h2 = h2 + size * c          ▷ h2 is the sum of the bytes multiplied by a constant
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 is used to get large outputs
h3 = h3 ⊕ c
return (h1+h2+h3)
```

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 depends on the file size as given in Eq. (2), where b_{min} is the minimum block size with a default value of 3, S is the desired number of chunks with a default value of 64, and N is the file size in bytes:

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

We will not discuss this formula in detail, but the relation $b \approx \frac{N}{S}$ is obvious. Thus we expect to have about S chunks.

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 of a file is simply the concatenation of all Base64 hash characters.

Since the block size is used for determining the chunks, 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 and $2b$. If there are too few Base64 hash characters for block size b (i.e. at most $\frac{S}{2} - 1 = \frac{64}{2} - 1 = 31$), Kornblum sets $b \leftarrow \frac{b}{2}$ and the whole process is repeated.

```
$ ssdeep Msdosdrv.txt
ssdeep,1.0--blocksize:hash:hash,filename
384:6A+A46SBSZHJEi4gMOzscKThLKxmdokp72mzdfdM72l3zefMENY2PDr20sypztHc:
   KQx+AecKumvlAN20sY0yX5uR,"Msdosdrv.txt"
```

Figure 2. A sample ssdeep output

A sample output of `ssdeep` is given in figure 2. The `ssdeep` hash is computed over the file `Msdosdrv.txt`. The integer at the beginning of the output is the block size b. In our example the block size is 384. Then the two `ssdeep` hash values comprising the Base64 hash characters for block size 384 and $2 \cdot 384 = 768$ are printed. Finally, the name of the processed file is given.

IV. Forensic Software for Hash Functions in Computer Forensics

Automated forensic analysis methods are gaining more and more attention because of the increasing amount of data within an investigation. There are many free and commercial tools to facilitate the work of the investigators. In Germany tools like Perkeo[5] and Artemis[6] are used for blacklisting and accepted in court. Perkeo is a special data scanner for child pornography whose basic database is maintained by the German Federal Criminal Police Office (BKA). However, both tools make use of cryptographic hash functions. In order to avoid a detection from traditional hash functions, criminals might change some extraneous header information, e.g. metadata within a file like an author or a comment field.

[5]http://www.perkeo.net/; visited 15.01.2011

[6]http://www.seed-forensics.de/software/artemis; visited 15.01.2011

Next we will have a look at three commercial tools for forensic investigators found in [10, p11]. All of these tools have an interface to import the RDS of NIST. We shortly discuss their ability of block based hashing and context triggered piecewise hashing.

1) *EnCase* by Guidance Software[7] is probably the most common forensic tool. EnCase does not come with the functionality to identify similar files on base of CTPH. The EnCase Cybersecurity suite has the possibility to identify similar malware using an entropy near-match analyser technology as said in [11], but this functionality isn't available for forensics.
   However, an EnScript due to Alexander Geschonneck [12] implements similar features as with `dcfldd`. This script aims at finding partial file matches. It searches in non-allocated clusters for fragments of files and compares them against previously created hash-sets.
2) *X-Ways Forensics* by X-Ways Software Technology AG[8] seems not to implement CTPH, too. However, there is a similar functionality at the semantic level for pictures (i.e. image recognition) as stated in [13, post #118]: *Known pictures can be recognized even if they are stored in a different file format, resized, if the colors or the quality are different or they have been edited, etc.*
3) The *Forensik Toolkit (FTK)* by AccessData[9] *contains a fuzzy hash utility* as it's said in [14]. The method is advertised to save time for investigators, to match parts of files to the original one or to identify similar files. It seems that the FTK implementation of context triggered piecewise hashing in FTK actually is `ssdeep`.
   Despite the positive assessment, the manufacturers warn to rely entirely on their context triggered piecewise hashing as they recommend: *[...] investigator will still be required to make the final decision on whether certain documents in a case are similar or not [14, p9].*

V. Generating Random Numbers for CTPH

As explained in Sec. III `ssdeep` makes use of a pseudo random trigger function PRTF to determine the trigger points. In order to have a good distribution of trigger points, the PRTF should follow an equal probability distribution.

In this section we sketch a new PRTF for context triggered piecewise hashing. We give a short discussion that our PRTF has both, a better performance and randomness than Kornblum's approach.

We test both PRTF s for the following files:

- *Msdosdrv.txt*[10] is a text file from the windows operating system with 45.582 bytes. The file is also part of the reference data set (RDS) of NIST.
- *hacker_siedlung.jpg* is a self-made picture with 68.650 bytes. We decided for jpg since it is widely spread on the internet and has an easily changeable header.
- *hacker_siedlung.bmp* is the same picture like before, but this time in the bitmap format with 959.754 bytes. We decided for bmp since this format uses a simple encoding.
- *gesetz.pdf*[11] is the german tax law as a pdf file with a size of 314.077 bytes.

Our PRTF is given in Algorithm 2. The main two differences of our approach compared to Kornblum's one are first the initial values for the registers $h1$ and $h2$, and second the right-shift-operation on register $h2$. $h2$ is used to mutate the higher bits of the rolling hash (`c << 24`) and $h3$ for the lower bits. Due to the multiplication in the return-value, $h1$ influences all 32 bits, too.

In order to test the randomness of a PRTF, we first used an existing framework of NIST that is available in [15]. This framework is a test suite for the validation of pseudo random number generators for cryptographic applications consisting of 15 different tests: Frequency (Monobit) Test, Frequency Test within a Block, Runs Test, tests for the Longest-Run-of-Ones in a Block, and so on.

We processed each of the four files as follows: We slide the position p through the file and write the concatenation of the bit strings $PRTF(BS_p)$ into a file (for a definition of BS_p see Eq. (1)). This bit string then was used as an input for the NIST framework. In summary most tests fail for all files. This is due to the design of $PRTF$, which outputs bit strings of length 32 instead of a single bit.

Therefore we implemented two own tests to compare pseudo random functions:

[7] http://www.guidancesoftware.com/; visited 30.11.2010
[8] http://www.x-ways.de/; visited 30.11.2010
[9] http://accessdata.com/; visited 30.11.2010
[10] http://support.microsoft.com/kb/234868; visited 10.01.2011
[11] http://www.ag.ch/sar/output/651-100.pdf; visited 15.11.2010

Algorithm 2 Pseudocode of the new rolling hash

$h1, h2, h3, c$ are unsigned 32-bit values
$h2 = 0x81a5c9f3$
$h3 = 0xa51fbc31$
$window$ is an array of $size$

to update the rolling hash for a byte c

```
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

h2 = (h2 >> 5) ⊕ (c << 24)             ▷ h2: right-shift; new byte influences higher bits
h3 = (h3 << 5) ⊕ c                     ▷ h3: left-shift; new byte influences lower bits

return h3 ⊕ h2 ⊕ (h1*0x7fffffff)       ▷ 0x7fffffff = 2^31 − 1
```

	Msdosdrv.txt		hacker_siedlung.jpg		hacker_siedlung.bmp		gesetz.pdf	
	old	new	old	new	old	new	old	new
expected value	59.35	59.35	44.69	44.69	39.05	39.05	51.12	51.12
standard dev.	124.39	120.81	6.83	6.65	52.57	50.62	53.07	54.69
min. amount	19	17	22	23	8	9	23	20
max. amount	2031	1984	84	95	1202	1198	2055	2181

Table II
TEST 1: NEW ROLLING HASH VS. OLD ROLLING HASH

1) Let b be the block size of the processed file. Our first test checks randomness with respect to the block size b. It counts how often each of the values $0 \bmod b$, $1 \bmod b$, ..., $b-1 \bmod b$ appears. Our aim is to have an equal distribution. Thus each value shall occur N/b times (where again N denotes the file size in bytes).
2) Our second test determines the number of second preimages. With a uniform distribution and different windows, only few second preimages of a value $PRTF(BS_p)$ should appear, if the file size is reasonable (say up to some MiBs). The workflow is as follows: For each position p within a file, our test computes $PRTF(BS_p)$, stores it in a database, and looks up if there is a k, $0 \leq k \leq p-1$ with $PRTF(BS_p) = PRTF(BS_k)$. If this is true and we additionally have $BS_p = BS_k$ then no second preimage is found. Otherwise we increase the counter of preimages of $PRTF(BS_p)$. Since all of our four files are significantly smaller than 2^{32} bytes, we expect to have a small number of second preimage collisions.

Table II compares the random behavior of both PRTFs with respect to our test 1. Row 1 to 3 from Table II are the basic stochastic values. We conclude that our approach is superior to Kornblum's one except for the file *gesetz.pdf*. The last two rows of Table II are the minimum / maximum amount of a value $k \bmod b$, i.e. a 19 signifies that there are 19 positions p within a file yielding $PRTF(BS_p) \equiv k \bmod b$. Both rolling hash functions yield similar results.

Table III shows a comparison of both rolling hash functions with respect to our test 2. Row 1 of Table III is equal to the file size in bytes. The row *1 time* is the number of unique rolling hash values, *2 times* is the amount of rolling hash values having two preimages within the respective file, and so on. *Repetitions* is the amount of repeated byte sequences. We conclude that our proposal of a PRTF induces significantly less second preimage collisions.

Finally, we look at the efficiency of both rolling hash functions. In Table IV we see that the new rolling hash function is faster. This test is based on a 50 MiB and a 100 MiB file from `/dev/urandom` and considers exclusively the rolling hash function.

The disadvantage of the new rolling hash function is that it is not as flexible as the old one. If there is a change of the window size, all shift-operations and the multiplication have to be adjusted.

VI. ANTI-FORENSICS ON CTPH

In Sec. II we described the usage of context triggered piecewise hashing in computer forensic investigations. This section describes anti-forensic measures based on a blacklist, i.e. how to circumvent a blacklist, which is based on Kornblum's

	Msdosdrv.txt		hacker_siedlung.jpg		hacker_siedlung.bmp		gesetz.pdf	
	old	new	old	new	old	new	old	new
possible values	45582	45582	68650	68650	959.754	959.754	314077	314077
1 time	20844	20888	68581	68583	617285	617632	271186	271202
repetitions	24694	24694	65	65	342034	342034	42869	42869
2 times	22	0	2	1	216	44	11	3
3 times	0	0	0	0	1	0	0	0

Table III
TEST 2: NEW ROLLING HASH VS. OLD ROLLING HASH

	old	**new**
100 MiB	2 sec 527519 msec	1 sec 814968 msec
	2 sec 204430 msec	1 sec 892178 msec
	2 sec 222300 msec	1 sec 817345 msec
	2 sec 476963 msec	1 sec 850076 msec
	2 sec 263307 msec	1 sec 809587 msec
50 MiB	1 sec 659221 msec	0 sec 891254 msec
	1 sec 159536 msec	0 sec 895482 msec
	1 sec 68835 msec	0 sec 911760 msec
	1 sec 63804 msec	0 sec 898916 msec
	1 sec 60671 msec	0 sec 900254 msec

Table IV
PERFORMANCE: NEW ROLLING HASH VS. OLD ROLLING HASH

hashes. We give a proof of concept that shows how to avoid automated detection by a blacklist-based CTPH approach. As in Sec. V our proof of concept to successfully manipulate files takes the following four file classes into account:

1) Text files in ASCII encoding.
2) Images in jpg format.
3) Images in bmp format.
4) pdf files (partly).

We point out that we do not take the semantic level into account (e.g. comparison of the perceptual impression of an image).

In Sec. VI-A we analyse the trigger sequences and describe the possibility to pre-compute global trigger sequences. We make use of these global trigger sequences to manipulate different files with a complexity of $O(1)$. Next, we concentrate on anti-blacklisting attacks. From a criminal's point of view, bypassing a blacklist is an important issue. We therefore describe our anti-blacklisting approach in Sec. VI-B.

A. Pre-Computation of Trigger Sequences

In this section we describe how to pre-compute trigger sequences, which we use in the subsequent sections for our anti-blacklisting attacks. Our aim is to find a set of trigger sequences, which are independent of the file at hand.

Trigger Sequence	**Base64 Hash Character**	**Trigger Sequence**	**Base64 Hash Character**
`AAAD?Hp`	9	`AAAV?Hf`	l
`AAAD?Og`	v	`AAAf?Ft`	p
`AAAD?QI`	7	`AAAr?xj`	V
`AAAJ?MW`	P	`AAAx?Fj`	l
`AAAJ?PJ`	F	`AAAx?OC`	n
`AAAJ?V0`	Z	`AAAx?tx`	5

Table V
SAMPLE PRE-COMPUTED GLOBAL TRIGGER SEQUENCES AND THEIR CORRESPONDING BASE64 HASH CHARACTERS

The basic idea is as follows: The modulus (i.e. the block size) for determining the trigger points is of the form $b = b_{min} \cdot 2^k$. As Kornblum's implementation of `ssdeep` makes use of $b_{min} = 3$, we first estimate an upper bound of k and then compute a trigger sequence for this maximal block size.

More precisely, we assume that the majority of the files, which we investigate, does not exceed the file size $N = 15$ MiB $= 15 \cdot 2^{20}$ Bytes. Using Kornblum's default values $b_{min} = 3$ and $S = 64$, Eq. (2) yields

$$\begin{aligned} k_{max} &= \left\lfloor \log_2 \left(\frac{N}{S \cdot b_{min}} \right) \right\rfloor \\ &= \left\lfloor \log_2 \left(\frac{15 \cdot 2^{20}}{2^6 \cdot 3} \right) \right\rfloor \\ &= \left\lfloor \log_2 \left(5 \cdot 2^{14} \right) \right\rfloor \\ &= 16 \, . \end{aligned}$$

Our aim is to find a byte sequence $BS := B_0 B_1 \cdots B_6$ of length 7 with an output $m := PRTF(BS) = 3 \cdot 2^{16} - 1$. Then BS will trigger for all block sizes $b = 3 \cdot 2^k$ with $0 \leq k \leq 16$, i.e. for all files of file size at most 15 MiB. We call such a byte sequence a *global trigger sequence*.

As a proof, we only have to show $m \equiv -1 \bmod b$. However, if $b = 3 \cdot 2^k$ with $0 \leq k \leq 16$, we have

$$m = 3 \cdot 2^{16} - 1 = 3 \cdot 2^k \cdot 2^{16-k} - 1 = b \cdot 2^{16-k} - 1 \equiv 0 \cdot 2^{16-k} - 1 = -1 \bmod b \, . \tag{3}$$

Table V shows a sample set of twelve global trigger sequences, which serve as triggers for any file of size ≤ 15 MiB. If BS denotes such a global trigger sequence of length 7, the column *Base64 Hash Character* in Table V represents the corresponding `ssdeep` hash character $LS6B(h(BS))$ as explained in Sec. III.

The computation of global trigger sequences is very efficient in practice. For instance, finding the twelve sample trigger sequences listed in Table V took us about 2 seconds in our virtual machine. The reason is that the computation of $PRTF(BS)$ is very fast and that we expect to find a hit after $\frac{b}{2}$ trials for BS, where b denotes our block size $b = 3 \cdot 2^{16} = 196608$ (again we assume that $PRTF$ yields equally distributed values and we have to find a byte sequence BS with $PRTF(BS) \equiv -1 \bmod b$).

B. Anti-Blacklisting

Anti-blacklisting means that we are able to efficiently generate *false negatives* from Kornblum's point of view. To explain this let a known-to-be-bad file F_1 be given. Our manipulation of F_1 yields a file F_2, which is semantically similar to F_2 (e.g. if F_1 is an image, the visual impression of F_1 and F_2 are almost identical for a human being). However, F_2 is assessed as non-similar by Kornblum's similarity algorithm and thus a false negative.

We present two anti-blacklisting approaches in what follows:

1) *Editing between trigger points*: The key point is that we preserve the location of the trigger points within a file and change one byte in each chunk. Then the corresponding traditional hash values change with a high probability as explained below.
   This anti-blacklisting approach is suitable for text files and bitmap images.
2) *Adding trigger points*: This attack takes advantage of an implementation weakness of Kornblum's approach. Let l be the number of Base64 characters in Kornblum's hash string. Then Kornblum's `ssdeep` implementation requires $l \leq 64$. If `ssdeep` processes a file and has already output 63 characters, than all further trigger points are ignored and the final Base64 character corresponds to the whole final chunk of the file, starting after the 63^{th} trigger point and ending at EOF.
   This approach is easily applicable to jpg files or pdf files. We denote by F_1 such a file. In order to get the corresponding manipulated file F_2, we insert as many trigger sequences as reasonable in the file header (e.g. 63) of F_1. This does not affect the semantic of the contents.
   In order to hide the similarity between F_1 and F_2, we randomly choose global trigger sequences yielding a different Base64 hash string than for F_1.

1) Editing Between Trigger Points: The idea of this anti-blacklisting attack is to manipulate one byte within each chunk, but to preserve the location of all trigger points. As we do not want to change the location of the trigger points and Kornblum only makes use of trigger sequences of length 7, this attack is applicable, whenever a chunk comprises at least 8 bytes. We point out, that our approach preserves the file length.

We assume, that the traditional hash function h yields equally distributed values. Then the Base64 hash characters are equally distributed, too (i.e. the Base64 characters, which are output by the function $LS6B \circ h$, follow an equal distribution). The codomain of $LS6B \circ h$ contains $64 = 2^6$ elements. Thus if we manipulate a random byte in a chunk, a change of this byte leads to a different Base64 hash character with probability $\frac{1}{64} = 2^{-6}$. This probability does not dependent on the size of the chunk.

```
$ cp Msdosdrv.txt Msdosdrv.txt.backup

$ ./anti-blacklisting-ebtp -ascii -7 Msdosdrv.txt
block size: 768
7 -> WINDOW: ->   122  -> changed:   to _
 chunk-size: 13172    modified character offset: 13165
14 -> WINDOW: -> an EMS -> changed: r to R
 chunk-size: 15467    modified character offset: 28632
21 -> WINDOW: ->     Nu -> changed:   to _
 chunk-size: 9928    modified character offset: 38560
-------------bs-changed----------------
block size: 384
7 -> WINDOW: ->ave Cur -> changed: S to s
 chunk-size: 4816    modified character offset: 4809
14 -> WINDOW: ->  0;88  -> changed:   to _
 chunk-size: 4435    modified character offset: 9244
21 -> WINDOW: ->   122  -> changed:   to _
 chunk-size: 3921    modified character offset: 13165
28 -> WINDOW: ->adapter -> changed:   to _
 chunk-size: 4207    modified character offset: 17372
35 -> WINDOW: ->moves\n -> changed:   to _
 chunk-size: 2328    modified character offset: 19700
42 -> WINDOW: ->u use D -> changed: o to O
 chunk-size: 2010    modified character offset: 21710
49 -> WINDOW: ->=altreg -> changed: A to a
 chunk-size: 2548    modified character offset: 24258
56 -> WINDOW: -> an EMS -> changed: r to R
 chunk-size: 4706    modified character offset: 28964
63 -> WINDOW: ->  Preve -> changed:   to _
 chunk-size: 3323    modified character offset: 32287
done

$ ssdeep Msdosdrv.txt.backup Msdosdrv.txt -p
$ ssdeep Msdosdrv.txt.backup Msdosdrv.txt -l
ssdeep,1.0--blocksize:hash:hash,filename
384:6A+A46yByZHJEb4gMOzs8KThLKxGdokp7WGzdfdM7Wl3zefMkNY2PDrW0syJztHc:
   KQx+5e8KuGPlgNW0s40yX5uR,"Msdosdrv.txt.backup"
384:6A+A46SBSZHJEi4gMOzscKThLKxmdokp72mzdfdM72l3zefMENY2PDr20sypztHc:
   KQx+AecKumvlAN20sY0yX5uR,"Msdosdrv.txt"
```

Figure 3. Anti-blacklisting attack 1 – Editing between trigger points for ASCII files

When implementing this anti-blacklisting attack, we have to ensure that changing some bytes in the original file F_1 does not significantly affect the visual impression of the manipulated file F_2 for the person looking at F_2. We explain how to successfully manipulate ASCII files and bitmap image files in what follows. However, for jpg images or pdf files this approach is not applicable.

- The manipulation of ASCII files is quite simple. Let $BS = B_0B_1 \cdots B_{m-1}$ denote the currently processed chunk. We assume $m \geq 8$. Let i denote the index of the byte we manipulate. This byte satisfies the following two properties: First, B_i represents a printable character. Second, i is maximal with the additional property $i \leq m - 8$. Although the changes can take place anywhere in the chunk, we opted for the end because it can be efficiently implemented.
  The replacement of characters is implemented as an array-lookup. If B_i is a lower case letter, we replace it by the corresponding upper case letter and vice-versa. If B_i is a special character, we replace it by a similar looking character, e.g. a space will be an underscore, a slash will be a black-slash. If the resulting Base64 hash character remains the

same, we switch to the next smaller index i.
If we are not successful for all $i \geq 0$, we simply set $B_0 \leftarrow B_0 \oplus 1$. Similar approaches are well-known from avoiding spam-mail detection.
In order to optimize our attack in practice, we exploit another weakness in Kornblum's `ssdeep` implementation. This 'feature' allows us to only manipulate every 7^{th} chunk as we see from Kornblum's following quote [16, fuzzy.c, line370]: *We only accept a match if we have at least one common substring in the signature of length ROLLING_WINDOW [size of window = 7]. This dramatically drops the false positive rate for low score thresholds while having negligible affect on the rate of spam detection.*
So we only have to change every 7^{th} Base64 hash character to avoid automatic detection by `ssdeep`. Thus, if the Kornblum hash comprises the maximum number of 64 Base64 characters, we have to manipulate 10 chunks (the 7^{th}, 14^{th}, 21^{st}, ..., 63^{rd}).
This anti-blacklisting attack is realized through a C-implementation called `anti-blacklisting-ebtp` on an Ubuntu 10.04 virtual machine. We next explain a sample output of our software in figure 3. We modify the ASCII file *Msdosdrv.txt* to get a false negative, i.e. the manipulated file is not assessed to be similar to *Msdosdrv.txt* by `ssdeep`. First, we copy the file *Msdosdrv.txt* to get a backup copy named *Msdosdrv.txt.backup*. The backup serves as the original file F_1 and the file *Msdosdrv.txt* will be our false negative F_2.
We invoke our anti-blacklisting software with two options. The flag `-7` indicates that we only want to change every 7^{th} chunk as described above. The option `-ascii` specifies that our software has to use the rules for manipulating an ASCII file.

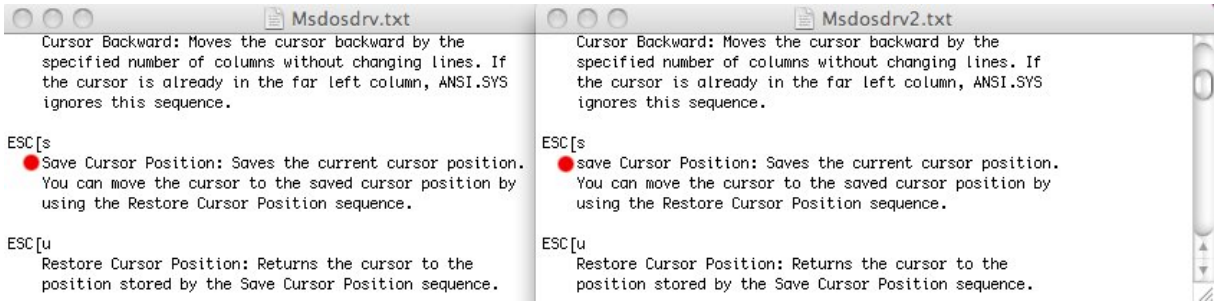

Figure 4. Manipulation of an ASCII text file

Fig. 3 then shows that the initial block size is 768 bytes as computed by Eq. (2). We manipulate every 7^{th} chunk. For instance, our software identifies the trigger sequence of the 7^{th} chunk to be the window `␣␣␣122␣` and changed the space `␣` at offset 13165 within the 7^{th} chunk to an underscore. However, as the initial block size does not yield enough trigger sequences, the block size is halved and the changes are reseted.
Let us next consider the changes for the block size 384 bytes. Our software identifies the trigger sequence `ave␣cur` of the 7^{th} chunk and changed the capital `S` to the corresponding lower case letter `s` at offset 4809 within the 7^{th} chunk. Our software then manipulates every 7^{th} chunk in a similar manner.
Finally, we test if `ssdeep` assesses the files *Msdosdrv.txt* and *Msdosdrv.txt.backup* to be similar. We make use of the pretty matching mode of `ssdeep` for these two files, which is invoked by the flag `-p`. However, as `ssdeep` does not give any output, it identifies a non-match, i.e. a false negative.
For convenience, we additionally output the `ssdeep` Kornblum hashes of *Msdosdrv.txt* and *Msdosdrv.txt.backup*, at the end of figure 3, respectively. The `-l` flag suppresses the absolute file path, i.e. it forces `ssdeep` to output the file name without any directory names. Comparing the two hashes we determine that only every seventh character had changed.
In order to demonstrate our manipulation, Figure 4 shows an excerpt of the original file on the left side and the manipulated file on the right side. We see, that the manipulation of the 7^{th} chunk is a barely perceptible change to the eye.

- The *editing between trigger points* anti-blacklisting attack for ASCII files can simply be transferred to bitmap files. Often bitmap files do not use any kind of compression – they typically store the colour of each pixel within one byte (8 bit colour depth) or within a group of 3 bytes (24 bit colour depth: 1 byte for red, green, and blue, respectively). Besides the file header, each byte therefore represents the hex value of a colour of a pixel.
  Therefore a nearly 'invisible' manipulation can be done by incrementing or decrementing the hex value by one. If B

```
$ cp hacker_siedlung.bmp hacker_siedlung.bmp.backup

$ ./anti-blacklisting-ebtp -hex -7 hacker_siedlung.bmp
block size: 12288
7 -> WINDOW: ->0E 0E 11 0F 0F 0F 0F  -> changed: 10 to 11
 chunk-size: 244fc    modified character offset: 244f5
14 -> WINDOW: ->25 21 20 23 1E 1D 24  -> changed: 20 to 21
 chunk-size: 9245    modified character offset: 2d73a
21 -> WINDOW: ->7A 78 78 7D 78 79 7B  -> changed: 78 to 79
 chunk-size: 28dbe    modified character offset: 564f8
28 -> WINDOW: ->9C 9F 8E 9A 9E 93 9F  -> changed: 8D to 8C
 chunk-size: 188b0    modified character offset: 6eda8
35 -> WINDOW: ->1F 23 1E 1F 22 1D 1E  -> changed: 1E to 1F
 chunk-size: 230cb    modified character offset: 91e73
42 -> WINDOW: ->24 24 32 30 30 46 41  -> changed: 26 to 27
 chunk-size: 18236    modified character offset: aa0a9
49 -> WINDOW: ->25 18 21 24 1A 1F 20  -> changed: 23 to 22
 chunk-size: 24232    modified character offset: ce2db
56 -> WINDOW: ->DC DF EA E8 CA D7 D5  -> changed: DF to DE
 chunk-size: 15d47    modified character offset: e4022
done

$ ssdeep hacker_siedlung* -p -l
$ ssdeep hacker_siedlung.bmp hacker_siedlung.bmp.backup -l
ssdeep,1.0--blocksize:hash:hash,filename
12288:GrxpWErqprJp3415jQd+PS9Q5oCitcoMNaBpeEvNEB7cQ0c6ePmbLikD+Jd:
   23qZYLQd+PMQKlioXvNEVEiJJd,"hacker_siedlung.bmp"
12288:GrxpWEYqprJp3715jQd+wZ9Q5oCttcoMNa6peEvNEO7cQ0c61PmbLik8+Jd:
   2kqZbLQd+w7QKaiowvNEY5icJd,"hacker_siedlung2.bmp"
```

Figure 5. Anti-blacklisting attack 2 – Editing between trigger points for BMP files

denotes the byte we want to manipulate, we therefore simply set $B \leftarrow B \oplus 1$. For instance, the byte $B = 0xE1$ is changed to $0xE1 \oplus 1 = 0xE0$.

Our manipulation workflow for bitmap files is therefore quite simple. Let $BS = B_0 B_1 \cdots B_{m-1}$ denote the currently processed chunk. As above, we assume $m \geq 8$, which is reasonable due to the length of bitmap files. We then set $B_{m-8} \leftarrow B_{m-8} \oplus 1$.

Fig. 5 shows a sample application of our software `anti-blacklisting-ebtp` to manipulate a bitmap image. We first copy the file *hacker_siedlung.bmp* to get a backup copy and then manipulate the file *hacker_siedlung.bmp* using our anti-blacklisting software. The flag `-7` forces our tool to change only every 7^{th} chunk. The option `-hex` specifies that our software has to use the rules for manipulating a bitmap file.

We then see that our software changes again every 7^{th} chunk. Finally, `ssdeep` can not identify a match. Both bitmap files are printed in figure 6.

2) Adding Trigger Points: This anti-blacklisting attack is based on the fact that a Kornblum hash value will be at most 64 characters long. We mentioned above that trigger points are ignored, if there are too many of them. As an example we assume to have 100 trigger points in our file. Then the first 63 trigger points invoke the traditional hash function h as usual and the LS6B of these hash values are concatenated to the Kornblum hash, respectively. The next 36 trigger points are ignored, only the last one will call the traditional hash function and generate the Base64 hash character for the final 37 chunks.

Many file formats, including jpg and pdf, allow small changes in the header. On Linux systems the header information can be modified with *jhead*[12] for jpgs or with *pdftk*[13] for pdfs. Our *adding trigger points* attack inserts a string consisting

[12]http://www.sentex.net/~mwandel/jhead/; visited 14.01.2011

[13]http://www.pdflabs.com/tools/pdftk-the-pdf-toolkit/; visited 14.01.2011

Figure 6. Manipulations of BMP-Files

```
$ cp hacker_siedlung.jpg hacker_siedlung.backup.jpg

$ jhead -ci trigger-sequences hacker_siedlung.jpg
Modified: hacker_siedlung.jpg

$ ssdeep hacker_siedlung* -l
ssdeep,1.0--blocksize:hash:hash,filename
1536:WZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZwLVoUY:
   WZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZp,"hacker_siedlung.jpg"
1536:ELVoUaX+ns+6iAuLNdElzt/CclGbn2OCFN8DXg1BSXHaL++:
   E3l6ew33lG2OMBSXa6+,"hacker_siedlung.backup.jpg
```

Figure 7. Attack2 - Adding trigger points JPG-files (simple version)

only of trigger sequences. From Section VI-A we know ASCII-based global trigger sequences and the corresponding Base64 hash characters. Again, given a file F_1 our target is to find a false negative F_2.

In a first, very simple variant of our attack, we proceed as follows. We assume to have a numbering for Base64 characters, that is for $0 \leq i \leq 63$ let C_i denote the i-th Base64 character. Furthermore, let BS_i be a corresponding global trigger sequence, i.e. $LS6B(h(BS_i)) = C_i$. We then choose a random index i and insert the corresponding byte sequence BS into the original file as often as required.

The disadvantage of this simple version is that the resulting `ssdeep` hash is very conspicuous if a human investigator looks at it as shown in figure 7. This listing shows a manipulation of the file *hacker_siedlung.jpg*. In listing 7, we first create a backup of the original file. We assume to have a file *trigger_sequences*, which is a text file containing the concatenation of the same trigger sequence 58 times. In our example we make use of the trigger sequence $BS = AAAJ?V0$ with $LS6B(h(BS)) = Z$. We insert the contents of the *trigger_sequences* file into our jpg image using the tool `jhead`. As claimed, the `ssdeep` hashes of the original and manipulated file do not yield a match.

In order to generate inconspicuous manipulated files, we proceed as follows. For $0 \leq k \leq 57$ we choose a random index i_k and insert the corresponding byte sequence BS_{i_k}. The probability that both the Base64 hash character of the original and manipulated file coincide is $\frac{1}{2^6}$. As `ssdeep` requires a common subsequence of at least Base64 hash characters in both Kornblum hashes, the probability of automated detection of our adding trigger points attack by `ssdeep` is $\frac{1}{2^{6 \cdot 7}} = 2^{-42}$, which is negligible.

As we have to insert at most 63 such trigger sequences, this attack yields a file expansion of at most $63 \cdot 7 = 441$ bytes. In all, this approach is very fast. Again we point out that inserting 58 trigger sequences is enough.

We point to a possible behaviour of this approach: Concatenating several global trigger sequences may lead to some unexpected trigger points within a global sequence. However, as we only want to generate a different Kornblum hash for our manipulated file, additional trigger points does not yield to automated detection by `ssdeep`.

In order to avoid this static approach, trigger sequences can be easily found with a brute-force attack on-the-fly, too. On a modern PC finding such values takes less then a second for small files. As mentioned above the generation for 12 global trigger sequences took about 2 seconds within our PC environment.

This issue can be used to optimize our attack with regard to the file growth as follows. We insert our first trigger sequence and then test which additional character(s) will cause a new triggering. Typically, 1-2 shifts of the window are enough to receive the next trigger point. This approach needs approximately $7 + 57 \cdot 1.5 = 92.5$ additional bytes, instead of $58 \cdot 7 = 406$ additional bytes, which we need by inserting 58 independent global trigger sequences.

```
$ cp hacker_siedlung.jpg hacker_siedlung.backup.jpg

$ ./generateTriggerSequenceForFile -jpg hacker_siedlung.jpg 58
[1] generation of file 'trigger-sequences'
[2] Checking for a match of original and manipulated file ---> no match
[3] call jhead -ci trigger-sequences hacker_siedlung.jpg
Modified: hacker_siedlung.jpg
done

$ ls -la hacker_siedlung*
-rw-r--r-- 1 user user 69061 2011-01-12 14:36 hacker_siedlung.jpg
-rw-r--r-- 1 user user 68650 2011-01-12 14:32 hacker_siedlung.backup.jpg

$ ssdeep hacker_siedlung* -p
$ ssdeep hacker_siedlung* -l
ssdeep,1.0--blocksize:hash:hash,filename
1536:cpZ51VIAET7HSBZJhMfLbuuVV1wssHaq66y6pwwk0o7HDGlddNwrHr/Pa55YLVo9:
   cPVGHNVV1mHaq66yewLHDri55Y3l6ew5,"hacker_siedlung.jpg"
1536:ELVoUaX+ns+6iAuLNdElzt/CclGbn2OCFN8DXg1BSXHaL++:
   E3l6ew33lG2OMBSXa6+,"hacker_siedlung.backup.jpg"

$ xxd hacker_siedlung.jpg | less
0000000: ffd8 ffe0 0010 4a46 4946 0001 0101 0048  ......JFIF.....H
0000010: 0048 0000 ffdb 0043 0005 0304 0404 0305  .H.....C........
0000020: 0404 0405 0505 0607 0c08 0707 0707 0f0b  ................
...
0000090: 1e1e 1e1e 1e1e 1e1e 1e1e 1e1e 1e1e fffe  ................
00000a0: 01a1 4141 4141 4158 2b41 4141 413f 706f  ..AAAAAX+AAAA?po
00000b0: 4141 4141 3f77 6641 4141 413f 7948 4141  AAAA?wfAAAA?yHAA
...
0000220: 5574 4f41 4141 4155 7a35 4141 4141 566a  UtOAAAAUz5AAAAVj
0000230: 4f41 4141 4157 4473 4141 4141 574b 6a0a  OAAAAWDsAAAAWKj.
0000240: 0aff c000 1108 01c9 02bc 0301 2200 0211  ............"...
```

Figure 8. Attack2 - Adding trigger points in jpg files

Next we explain the working steps of our software `generateTriggerSequenceForFile`, which is our second tool for anti-blacklisting:

- First, our software generates the file *trigger_sequences*. This file holds the concatenation of 58 trigger sequences for the original file.
- Second, despite the low probability, our software invokes `ssdeep` to check if both Kornblum hashes yield a match. If there is a match, we go back to step one.
- Third, depending on the file type, an extern software is used to include the trigger sequences into the file header. For jpg files, we call *jhead*, for pdf files we make use of *pdftk*.

Listing 8 shows a sample output of our tool to create an inconspicuous, but also different Kornblum hash for a jpg file. To include the trigger sequences from the file *trigger_sequences* into the jpg header, we invoke the *jhead* command. This software allows to modify the header segments of jpg files including exif information from the command line. The comment segment can be changed by using the *jhead* option `-ci <name-of-comment-file>`.

The `ls`-command at figure 8 shows the file growth of 411 bytes, which are due to $58 \cdot 7 = 406$ bytes for the trigger sequences and additional 5 bytes to update the segment information of the jpg header. The `ssdeep` hashes of the original and manipulated images differ as we conclude. Using the hex dump of the manipulated file *hacker_siedlung.jpg*, it is obvious that the trigger sequences have a very low offset (they start at offset $0xa2 = 162$). In this example, we do not have control over the bytes 0 to 161 of the manipulated file. However, it is very unlikely that within these 162 already 7 `ssdeep` hash

```
InfoKey: Comment
InfoValue: <sequence of trigger points>
```

Figure 9. pdftk metadata file

```
$ ./generateTriggerSequenceForFile -pdf gesetz.pdf 58
[1] generation of file 'trigger-sequences'
[2] Checking for a match of original and manipulated file ---> no match
[3] call pdftk gesetz.backup.pdf update_info trigger-sequences output gesetz.pdf
done

$ ssdeep gesetz* -l -p
$ ssdeep gesetz* -l
ssdeep,1.0--blocksize:hash:hash,filename
6144:nXC0Gn+dk4awUYz3vWZJ79o1zO4aj9G8YJvKIc/Ky:
   XFxrXFbkJ7umj9ysL/Ky,"gesetz.backup.pdf"
6144:US8OqktJ5aHqIhfv2THAFI9cN4Z8WPYJP6iJrbuVV1mHn55v:
   UbA/Exx0HAOn8f0sOVV1G55v,"gesetz.pdf"

$ xxd gesetz.pdf | less
0000000: 2550 4446 2d31 2e35 0a25 e2e3 cfd3 0a31  %PDF-1.5.%.....1
...
004b120: 6279 2031 5433 5854 292f 436f 6d6d 656e  by 1T3XT)/Commen
004b130: 7428 4141 4141 4158 2b41 4141 413f 706f  t(AAAAAX+AAAA?po
004b140: 4141 4141 3f77 6641 4141 413f 7948 4141  AAAA?wfAAAA?yHAA
004b150: 4141 4366 6d41 4141 4143 6f46 4141 4141  AACfmAAAACoFAAAA
...
004b2b0: 5574 4f41 4141 4155 7a35 4141 4141 566a  UtOAAAAUz5AAAAVj
004b2c0: 4f41 4141 4157 4473 4141 4141 574b 6a29  OAAAAWDsAAAAWKj)
004b2d0: 2f4d 6f64 4461 7465 2844 3a32 3031 3130  /ModDate(D:20110
...
```

Figure 10. Attack2 - Adding trigger points PDF-files

characters have been triggered.

pdf and jpg files have similar characteristics in terms of header information, so this attack can partly be used for pdf files. In this case the counterpart for *jhead* is *pdftk*. Unlike *jhead* the new programme requires a file structure for *trigger_sequences* as given in figure 9.

A sample application of our tool `generateTriggerSequenceForFile` to manipulate a pdf file is given in figure 8. The output shows the downside of manipulating pdf files compared to jpg files: In figure 8 our trigger sequences are inserted at the high offset `0x004b120`. Since including additional header information using pdftk changes some more bytes in the header, this attack is still successful.

VII. Conclusion

We have discussed security issues of context triggered piecewise hashing as proposed by Jesse Kornblum. As a conclusion CTPH seems to be reasonable for detecting similar files, if no anti-forensic measures are present. However, we have proposed active anti-blacklisting attacks on `ssdeep` hashes and practically proven that Kornblum's approach does not withstand an active adversary against a blacklist.

Additionally, we discussed efficiency and randomness of the pseudo random trigger function. We improved the pseudo random number generator used within the `ssdeep` implementation of Kornblum with respect to both properties.

Further anti-forensic work will show that also anti-whitelisting on CTPH whitelists is possible. The main conclusion of the paper at hand is thus that in order to come to a fuzzy hash function, new approaches are necessary.

REFERENCES

[1] National Institute of Standards and Technology, "National Software Reference Library," http://www.nsrl.nist.gov, January 2010.

[2] J. Haitsma and T. Kalker, "A highly robust audio fingerprinting system," in *ISMIR*, 2002.

[3] 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

[4] A. Tridgell, "Spamsum," Readme, 2002. [Online]. Available: http://samba.org/ftp/unpacked/junkcode/spamsum/README

[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] A. Menezes, P. Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1997.

[7] SHS, "Secure hash standard," 1995.

[8] R. Rivest, "The md5 message-digest algorithm," 1992.

[9] A. Tridgell, "Spamsum," Readme, 2002. [Online]. Available: http://samba.org/ftp/unpacked/junkcode/spamsum/README

[10] E. Huebner and S. Zanero, *Open Source Software for Digital Forensics*. Springer, February 2010, vol. 1.

[11] G. Software, "Network-enabled incident response and endpoint data control through cyberforensics," http://www.realediscovery.com/WorkArea/DownloadAsset.aspx?id=1000000404, 2010.

[12] A. Geschonneck, "Ceic 2008: Block based hash analyse mit encase," http://computer-forensik.org/2008/04/30/ceic-2008-block-based-hash-analyse-mit-encase/, April 2008.

[13] X-Ways, "Winhex & x-ways forensics newsletter archive," http://www.winhex.com/winhex/mailings/index.html, July 2010.

[14] D. Hurlbut, "Fuzzy hashing for digital forensic investigators," AccessData, Tech. Rep., January 2009. [Online]. Available: http://www.accessdata.com/downloads/media/Fuzzy_Hashing_for_Investigators.pdf

[15] A. Rukhin, J. Soto, J. Nechvatal *et al.*, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," National Institute of Standards and Technology, http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html, Tech. Rep., April 2010.

[16] J. Kornblum, "ssdeep," Sourcecode and Documentation, September 2010. [Online]. Available: http://ssdeep.sourceforge.net/