Saarland University
Faculty of Natural Sciences and Technology I
Computer Science Department
Bachelor's Program in Computer Science

Bachelor's Thesis

MD5 collisions on multimedia files

submitted by

Holger Bornträger

Supervisor	Prof. Dr. Michael Backes
Advisors	Prof. Dr. Michael Backes
	Dr. Dominic Unruh
Reviewers	Prof. Dr. Michael Backes
	Dr. Dominic Unruh

Statement

Hereby I confirm that this thesis is my own work and that I have documented all sources used.

Saarbrücken, March 14, 2007

Holger Bornträger

Declaration of Consent

Herewith I agree that my thesis will be made available through the library of the Computer Science Department.

Saarbrücken, March 14, 2007

Holger Bornträger

Contents

1 Abstract **4**

2 Introduction **4**

3 Assumptions **4**
3.1 Packet based multimedia files 4
3.1.1 Construction 4
3.1.2 Fault tolerance 5

4 MD5 **5**
4.1 Constuction of MD5 6
4.2 Properties of MD5 6
4.3 Collisions on MD5 6
4.3.1 General collisions on MD5 6
4.3.2 Meaningful collisions on MD5 8

5 Collisions on multimedia files **8**
5.1 Usable collisions 8
5.1.1 General collision properties 8
5.1.2 Usable collisions for packet based multimedia files 8
5.2 Creating colliding multimedia files 8
5.2.1 Prerequisites 8
5.2.2 Basic idea 9
5.2.3 Interleaving the files 9
5.2.4 Attaching the collision 10
5.3 Why it works 11
5.3.1 Interleaving 12
5.3.2 Attaching the collision 12
5.4 Summary 13

6 Proof of concept - MPEG - Audio **13**
6.1 The MPEG standard 13
6.2 MPEG audio packet 13
6.2.1 MP3 - MPEG 1 Layer 3 14
6.3 ID3 tags version 1 and 2 14
6.3.1 ID3v1 14
6.3.2 ID3v2 15
6.4 Algorithm 15
6.5 Usable collisions for MPEG-audio 15
6.5.1 Creating usable collisions 15
6.5.2 Filter 17
6.6 Preparing the audio files 17
6.7 Program 18
6.7.1 Workflow 18
6.7.2 Performance 19
6.8 Result 19
6.9 Source 19
6.10 Limitations 19
6.10.1 Player 19

6.10.2 Seeking . . . 19
6.10.3 Filesize . . . 20

7 MPEG - Video **20**
7.1 Basic Construction of an MPEG video stream . . . 20
7.2 Collision with basic method . . . 21
7.3 Alternative method . . . 21

8 Conclusion **21**
8.1 PHMpegHeader . . . 22
8.1.1 Public interface . . . 22
8.2 PHMpegPacket . . . 23
8.3 Public Interface . . . 23
8.4 PHCleanedMpegPacket . . . 24
8.4.1 Public interface (additions) . . . 24
8.5 PHMpegFile . . . 24
8.6 Public interface . . . 24
8.7 PHMpegCollision . . . 25
8.7.1 Public Interface . . . 25
8.8 PHMpegInterleaver . . . 26
8.9 Public interface . . . 26
8.10 PHMd5CollisionAttacher . . . 26

List of Figures

1 A basic data packet 5
2 One round of MD5 7
3 Interleaving packets 9
4 Interleaving packed based multimedia files 9
5 Prepending a valid packet 10
6 Prepending an invalid packet 10
7 Collision data 10
8 Attaching a collision to a stream 11
9 Attaching a collision to a stream 2 11
10 Prepending a valid header 11
11 License of Stevens collision finding program 16
12 Usage of program 18
13 Construction of an MPEG video stream 20

1 Abstract

A generic break for MD5 is known some time, but also some other formats can be broken. In these cases, conditional branch structures are used to decide what content of the file is used. In this paper, MD5-collisions for some multimedia-formats are shown. As most of these formats have no branch structures, here a method will be shown, that uses the fault tolerance systems. It shows how two files can be interleaved in such a way, that a constructed packet at the beginning of the file allows to choose between the contents of the two files. It is also shown, how to construct this initial packet in such a way, that the multimedia files collide.

2 Introduction

There are some general methods to construct MD5-collisions on several formats: postscript, applications ... . These collisions are created by merging both possible outputs in one file. The executed/displayed data is chosen by a conditional statement, that contains the generic collision.
The method used for multimedia files is similar in the way, that the constructed files contain the information of both files. The difference to other formats is, that most multimedia-formats don't support jumps or branches. This is why this behavior is simulated using the error correction system. Single packets or frames of the multimedia file are interleaved, so that every other packet is invalid, and thus not played. This leads to the possibility to only play back half of the data, while keeping all information in the file. The generic collisions on MD5 are prepended to the file in such a way that, depending on the collision, one of the interleaved streams is played.

3 Assumptions

Not all multimedia file formats allow the construction used in this paper. To be considered usable for the methods described here, file formats have to meet several requirements.

3.1 Packet based multimedia files

The format has to be packed based. Packet based file formats are used to achieve good fault tolerance for streaming or sending files, as well as the possibility to start playing the file at any position (i.e. play a stream from it's current position).

3.1.1 Construction

A basic packet based format divides the information, be it audio or video, in small chunks. Every chunk gets it's own header, that carries all information needed, to interpret its content data, i.e. bitrate, information about the codec, ... (Fig. 1). This means, every packet can be played on it's own. A packet based multimedia file or stream is just a chain of such packets. This construction allows streaming of the media, as an internet radio or internet TV, as no

Figure 1: A basic data packet

entrypoint is needed. Playback can start at any position in the stream, without knowledge of prior packets, as all information needed is stored in every single packet.

For the methods described in this document, it is also required, that the length of the packets is fixed either by the format, or by the information in the header. This happens for many formats, as it simplifies parsing as well as decoding of the packets. It removes the requirements of removing all possible headers from the content data.

3.1.2 Fault tolerance

The construction as described in 3.1.1 allows a good fault tolerance, by simply ignoring damaged packets. As a packet only contains some miliseconds worth of data, a missing or not played packet is normally not perceived by the user.
Additionally the stream is not delayed, by waiting for packets, or re requesting packets. This also removes the need for the server to store packets that is has already sent. In most cases, waiting for a corrected packet would have more impact on the quality than just dropping the data.
A simple, yet effective, error correction algorithm is thus, playing valid packets and dropping all information of the file, until the next valid packet header is found. Therefore the header contains some sort of sync information, normally the first bytes, that allows to identify the next header in the stream. This sync information can also be the "magic number" that identifies the stream-type. To allow skipping of packets, every packet header must contain information about the length of the packet, to allow to seek for the end of the packet. Alternatively the stream may use a fixed packet-size.
This form of fault tolerance is especialy usefull for web-streams such as online-radios, as missing packets don't have to be resent, and no forward error correction is required.
Additionally it is required, that the data is fault tolerant when it comes to minor editing, such as changing one bit or byte. This is true for almost any multimedia format, as minor changes to the data only slightly affect the data itself. It might decrease the quality of the audio or video information, but it doesn't.

4 MD5

MD5 (Message-Digest algorithm 5), is a hash-algorithm, that produces 128 bit hashes.

4.1 Constuction of MD5

This section will not describe MD5 in all detail, but tries to give an overview about how it actually works.
MD5 is an algorithm that works in 16 rounds, covering a 512 bit message block. Every round consists of 16 similar operations.
Each operation works as follows (Fig 2):

1. Divide the 128 bit message in 4 blocks of 128 bit A, B, C, D .
2. Calculate $F(B, C, D)$ and add to $A \ mod \ 2^{32}$.
3. Add state $M_i \ mod \ 2^{32}$.
4. Add round constant $K_i \ mod \ 2^{32}$.
5. Shift by given amount per round .
6. Add $B \ mod \ 2^{32}$.

The state M_i is a given 32 bit value, for the first round of the algorithm, and then determined, by all previous rounds of the algorithm (also carried over to the next 512 bit block).
There are four different functions for F, one used for each block of 16 operation rounds (thus $16 * 4 = 64$ rounds).

4.2 Properties of MD5

As seen in 4.1, MD5 only does one pass over a file. i.e. after processing one 512 block of data, this data is not used anymore for the following rounds, except for the state, that it influenced.
This property means, that two files with identical MD5 hash will also hash to the same value, when identical data is appended to both files, i.e. if `File A` and `File B` both hash to value h_1, after appending `File X` to `File A` and `File B`, `File AX` and `File BX` hash to the same value h_2.
Additionally, (not stated in the construction), both files must have the same size, as MD5 includes the length of the data into the padding of the last packet.
This property is used to construct meaningful collisions, given general (random) collisions on MD5.

4.3 Collisions on MD5

4.3.1 General collisions on MD5

Several fast methods are known to create collisions on MD5 [?, ?, ?]. These collisions however are mostly random data. These methods use tunnels in the MD5 algorithm. This means, there are several known conditions, that, applied correctly, make it possible, to create data, that has a predictable behavior when fed into the hash algorithm.
One fast method of creating collisions on MD5 is by Vlastimil Klima[?], another one by Stevens[?].
Both create collisions by choosing data with certain properties, so that some bits determine the outcome of the algorithm predictably, i.e. there have to be

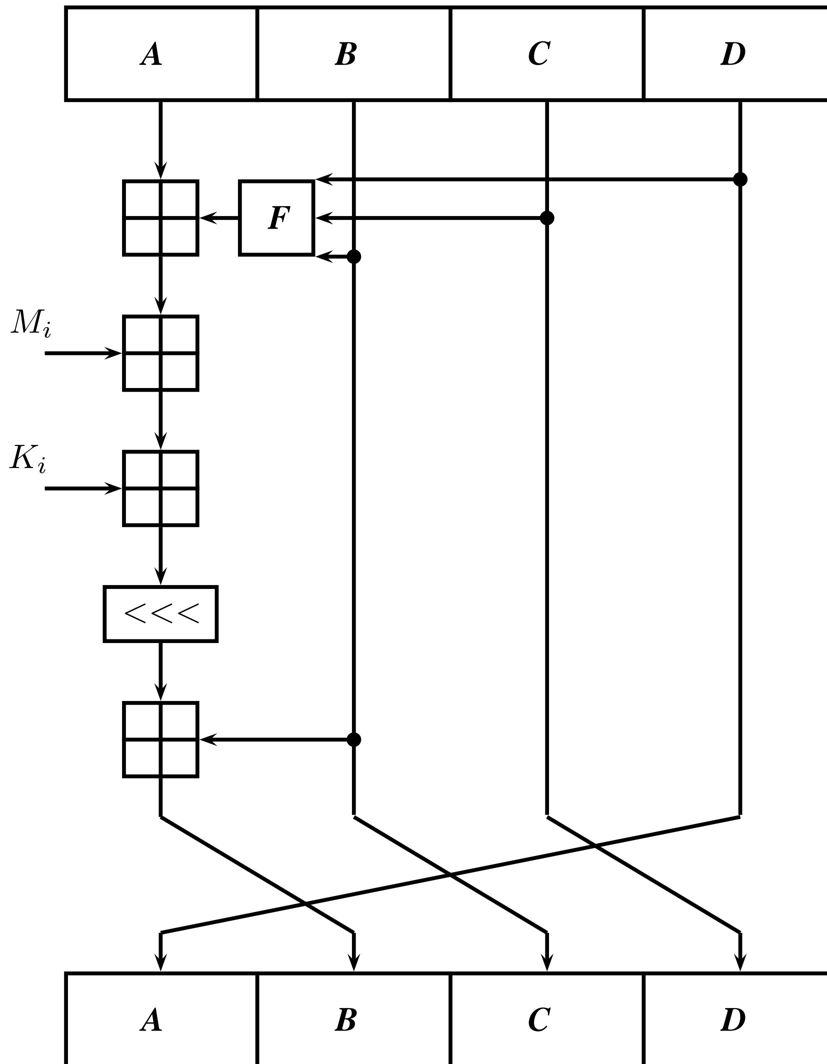

Figure 2: One round of MD5

two blocks of data, that only differ in these bits, and hash to the same value.

As only 5 bits change, the colliding blocks can be easily determined by brute force.

4.3.2 Meaningful collisions on MD5

Methods have been shown, to provide collisions on postscript documents[?] as well as applications[?]. These collisions are realized by using conditional branches provided by the specification of the document i.e. `if then else` statements.
Basically they create a pair of colliding data blocks, then use a conditional statement, that says:
`if block = collision_block_1 then do first else do second`.
If the files collide for the part up to the end of `block`, they collide completely, following directly from the properties in 4.2.

5 Collisions on multimedia files

5.1 Usable collisions

As there are no simple branch conditions available for most multimedia formats, several special conditions must be met by the collisions, to be considered usable for the method described in this paper.

5.1.1 General collision properties

Collisions on MD5 created with Stevens algorithm are 128 byte long, plus a potential initialization file, but they only differ in 5 bits. The rest of the collisions is rather random, although it has to fulfill some properties required by algorithm.

5.1.2 Usable collisions for packet based multimedia files

For the method described here, the collisons meet several conditions.

- The collision must contain a valid packet header, suitable for the stream.
- The header must contain one of the changing bits.

In practice, the first condition is normally met, by encoding the multimedia files in a way, that corresponds to the collision, as finding an appropriate collision for one special stream requires a sequence of several bytes to be met in the collision block, which is harder to achive.

5.2 Creating colliding multimedia files

5.2.1 Prerequisites

- an MD5 collision meeting the requirements of 5.1.2
- two files with the same header as in the collision blocks

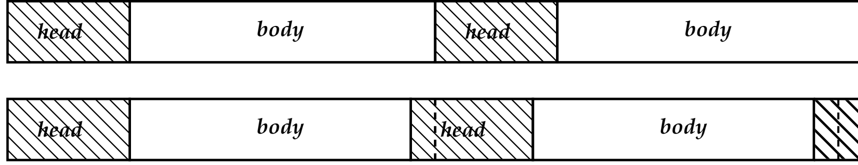

Figure 3: Interleaving packets

Figure 4: Interleaving packed based multimedia files

5.2.2 Basic idea

The final result will be constructed in a way, similar to those used for postscript documents or executables. However, as most multimedia formats don't contain any branch structures, this has to be realized by other means.
The basic concept is, that we interleave the packets in such a way, that only every other packet is played by the player. This is done, by invalidation every other packet, if the previous one was played, causing the error correction to skip this invalidated block. This method will create some sort of condition, that can be used, to create meaningful collisions on multimedia files.
In this case, the generic collision block is used, to switch between the two packet streams, and thus influencing, what content is played.

5.2.3 Interleaving the files

As stated in 3, for the method described here, we need multimedia formats, that contain all information required in the packet header of the individual packets. Also, it is important, that the packets either have a fixed length, or the length is computed from the packet header.

The construction as in figure 3 works as follows:
Every packet is shortened by a fixed amount of bytes. This leads to the fact, that the header of the next packet starts at a position, that is in the body of the previous packet. As a result, all packets overlap.

As the second packet in this construction becomes invalid, it should be dropped by the error correction. Though the error correction doesn't see it as an invalid packet but as random data. Dropping data up to the next header again, causes the third packet to be played correctly.
Using this effect, two files can be interleaved in an alternating manner (Fig. 4), so that on playback, there will be almost no difference to the first file, hiding the second file contained in the first one.

With this construction, a conditional element can be introduced in a format that didn't contain such an element in it's design, by using the error correction mechanisms. This is done by prepending and overlapping an addtitional packet. If this packet is valid (Fig. 5), it will be played back, and the first packet of the stream, belonging to the first file, is not played back. If this happens, the

Figure 5: Prepending a valid packet

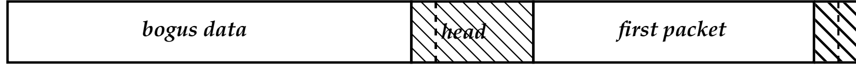

Figure 6: Prepending an invalid packet

second packet is played, skipping all the packets of the first file. Following this chain, only packets from the second file will be played.
On the other hand, if the prepended packet is not valid (Fig. 6)(i.e. does not contain a valid header), the packet will be ignored and the first valid packet is the first packet of the first file.

5.2.4 Attaching the collision

Up to this point, we only combined the two original files into one larger file. With the "switch-packet" prepended however, these two files do not collide on MD5.
Now, from the given collisions on MD5 (see 5.2.1 and 5.1), two sets of data are constructed that:

- let both files have the same MD5 hash (collision)
- one of them causes packets from file 1 to be played
- the other causes packets from file 2 to be played

Figure 7 shows one of the two collision blocks, namely the one that contains a valid packet header. As all collisions created have a fixed length of 128 bytes, they can fit in a stream packet as a whole (128 byte is too short for virtually any multimedia file format).

Such an collision block can be prepended to the file. As of the properties of MD5, if the beginnings of two files collide, and the rest of the file is identical, then either the files are a collision on MD5 or both files are identical (See 4.2). The collision can now be prepended to the interleaved file as in figure 8. The collision block is padded in such a way, that the length from the header up to the end of the padding is just the size of a normal packet minus one byte. The missing byte is just where the so created packet overlaps with the first packet of the actual stream. Although the data in this packet does not contain any meaningful audio-data, it is a valid packet, and thus played back. From this,

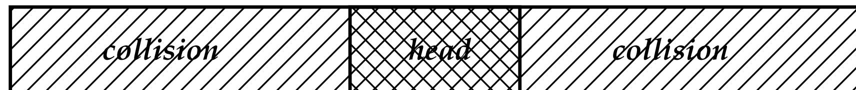

Figure 7: Collision data

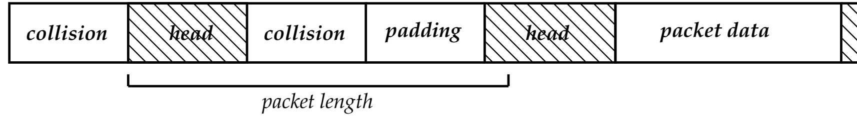

Figure 8: Attaching a collision to a stream

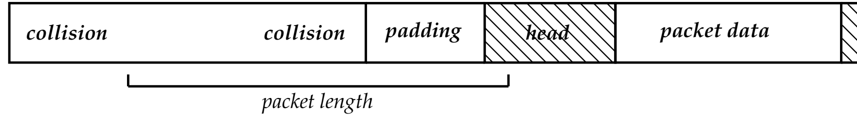

Figure 9: Attaching a collision to a stream 2

the first byte of the first real packet is missing, so the first packet is not played. As described in 5.2.3, this causes the playback of the second file.
The collision could also be the one without a valid header (Fig. 9). In this case, the first packet and thus the first file should be played.

As the files now start with two different blocks that form a collision on MD5, keeping the rest of the file identical, both files will now have the same checksum. On playback, the files will have completely different output (identically to either file one or file two).
This is a collision on MD5 for two multimedia files.

There is an additional method, to improve the compatibility of this construction with other players, but it makes it harder to find a usable collision, as it fixes the medias parameters and then searches a usable collision.
What can be done, is to add the beginning of a valid packet, s the part of the collision that is in front of the header is contained in this packet. This allows the file to run on players that require a valid header at the beginning of the file. The construction would be, first a valid header for a packet of the stream, followed by some filling data and then the collision (Fig. 10). From there the construction is as shown above.

5.3 Why it works

Now a more exact analysis will be given, why the methods given in 5.2 are working.

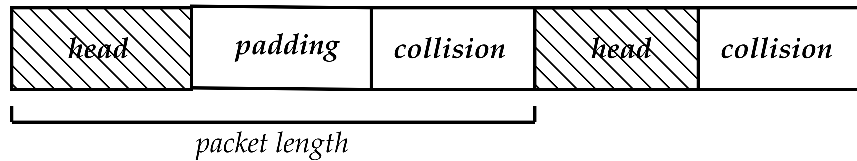

Figure 10: Prepending a valid header

5.3.1 Interleaving

A parser with high fault tolerance should basically work in the following steps:

1. drop bytes until valid packet header is found
2. interpret packet header
3. read all bytes that belong to the packet
4. repeat until end of file.

This simple system allows to drop damaged packets on internet streams, as well as in damaged files, with nearly no loss of data, as each packet only contains very limited amounts of data, that are not needed to play the following packets.

Assuming a parser with this properties, the construction in 5.2.3 works as follows:
Right at the beginning of the file, the parser finds the first packet header. It interprets the header, and calculates the length of the header. Then it reads remaining packet data (i.e. the number of bytes calculated). In the modified file, this includes the first bytes of the header of the second packet. This packet is send to the decoder.
Now, trying to read the second packet, the parser doesn't find a valid header, as the header of the second packet is missing it's first bytes. So the parser assumes there is some random data to be dropped, and does so. The next valid packet header that can be found, is the header of the third packet, i.e the complete second packet/first packet of the second stream is droped. The same process is done for all following packets, thus dropping every other packet, stripping all the data from the second file.

5.3.2 Attaching the collision

Again, the same parser as in 5.3.1 is assumed. Now with the prepended collision, the start of the parsing is slightly different. There are two possible beginnings of the file:
Either the file starts with a valid header, as the beginning of the collision is wrapped in a valid packet, or it starts with the beginning of the collision.
This only differs in the point, that in the first case, the parser reads a complete bogus packet, that is played back, then finds the header inside of the collision. In the second case, the parser would start finding random data, thus dropping it up to the first valid header.
From the point, where the header contained in the collision block is, there are two possible ways to go.

Valid header in collision If the collision file with the valid header is used, the parser reads the header, interprets it and thus reads the calculated number of bytes. As of the construction in 5.2.4, the bogus packet of the header in the collision, is shorter then it should be. This causes the parser to interpret the beginning of the first real packet as part of the bogus packet. Now the mechanics explained in 5.3.1 kick in again. The first packet has no valid header (due to

the missing bytes), and is dropped entirely by the parser. The second packet is the first valid packet played. The rest of the construction again follows 5.3.1.

no valid header in collision If no valid header is present in the collision, all of it's data is dropped, as the parser can't find a packet it belongs to. This causes the header of the first packet to be the first valid header in the stream. Starting right from this point, everything works exactly as in 5.3.1.

5.4 Summary

By using the fault tolerance of players, two packet based multimedia files can be interleaved in such a way, that on playback only data from one of them is used. When adding a matching collision block to such a file, the appropriate stream can be chosen. Due to the properties of MD5, the two files (one for each of the two parts of the two colliding blocks) hash to the exactly same value, but have different output when played back.

6 Proof of concept - MPEG - Audio

The MPEG[source needed] format is a packet based format for video and audio data. It fulfills the requirements as in 3.1. MPEG files consist of packets, identified by packet headers. They allow for multiplexing audio and video, as well as audio and video only streams. The MPEG standard does not provide the possibility for conditional or unconditional jumps, but must be played sequentialy. Only the player may implement seeking functionality. Only MPEG-audio will be discussed here in detail, for MPEG-video, the packet headers are different, but it allows for the same methods.

6.1 The MPEG standard

The MPEG standard[**?**] describes no file format for audio or video. Instead only the packet format is described. This is possible, due to the fact, that every packet contains all information needed for playback. Streams and files have the same format, they both are just a sequence of packets.
The MPEG standard contains no default method for fault tolerance or error correction, so the implementation of these features is up to the author of the player.

6.2 MPEG audio packet

The basic construction of an MPEG audio packet consists of a header and the audio data itself.
In the MPEG standard, these packets are called frames, this paper will call them packets, for consistency of the document.
For now only the header is of interest. It consists of 32 bits/4 bytes. The construction is (ignoring the unofficial standard MPEG version 2.5 which uses the last frame sync bit as version indicator):

- 12 bit FrameSync (all bits set)

- 1 bit MPEG version (version 1 or 2)
- 2 bit audio layer (1, 2 or 3)
- 1 bit flag if header CRC check is present
- 4 bit bitrate index
- 2 bit sampling frequency
- 1 bit flag if packet is padded
- 1 bit flag free for personal use (private bit)
- 2 bit channel mode (mono, stereo, joint stereo, dual stereo)
- 1 bit flag data is copyrighted
- 1 bit flag if media is original media or copy
- 2 bit for emphasis (used when decoding the data).

The information given in the header allows to compute the size of the packet, as for every standard, there is a fixed formula for the length the packet has to have. For example in case of MP3 this formula would be

$$FrameLengthInBytes = 144 * BitRate/SampleRate + Padding.$$

This allows a player to read one complete packet, then look for the next packet. Some data of the header is only additional information for the player/user. Such fields are the private bit and the information about copyright and original media. Other parts are fixed for all packets of all streams, for example the 12 bit FrameSync (11 bit for MPEG 2.5 which is not an official standard).

6.2.1 MP3 - MPEG 1 Layer 3

The most widely known application of the MPEG file or stream format is MP3, which is MPEG version 1 audio layer 3, an audio format.
It uses the packet and packet header format as shown in 6.2.

6.3 ID3 tags version 1 and 2

ID3 tags are the common method to attach additional information, such as author, title... to an MPEG audio file. The two versions differ in format as well as integration into the file.

6.3.1 ID3v1

Version 1 of the ID3 standard is attached to end of the file, thus giving no problem when interleaving the files. The ID3 tag, if present, must be the last 128 bytes, so only one tag is possible. The tag has to be identical for both files in the end, to allow the creation of the collision. It can be simply appended to the interleaved files in the end.

6.3.2 ID3v2

Version 2 is attached to the beginning of the file. If a ID3v2 tag has to be present in the created file, then the collisions must be created with this tag as initial vector. This makes it impossible to create the collisions before the data is known. While this doesn't harm the method, it slows down the process of collision finding. If possible, the ID3v2 tag could be replaced with version 1.

6.4 Algorithm

The algorithm follows strictly the principle from 5.2. For the special case of MPEG-audio the packets only have to overlap by one byte (the first byte of the sync).

6.5 Usable collisions for MPEG-audio

For MPEG-audio, the collisions have to contain a valid 4 byte MPEG audio header. Additionally this header can only be correct in one of the two collisions. Only the first 12 bits are the same in every packet header, the rest of the bits create a valid MPEG header for almost any possible combination. For this reason only collisions where the changing bit is contained in the first 12 bits are considered usable.
There are some additional requirements for the collisions to be considered usable. If a changing bit is in the first 12 bits, the rest of the packet has to be valid, i.e. contain valid values for all the fields. For instance, a bitrate index of $0x0$ is not valid. additionally some combinations of emphasis, channel mode and bitrate are not valid for MPEG audio.
As a final requirement headers that have the CRC flag set are not considered usable, as this would require an additional 16 bit CRC following the header, that also must be contained correctly in the collision. As this case is rare, it was not considered when searching usable collisions.

6.5.1 Creating usable collisions

The collision files where created using brute force. Steven's implementation of his method, described in [**?**], was used to create collision pairs. These collision pairs were passed to a filter, that analyzed them for MPEG headers, to decide if the collision is usable. The collision program was run in parallel several times, to use the full computing power of the server it ran on (A system with two dual-core Xeon CPU's).
The collision files were passed to a filter, that decided, if there were potentially usable collisions in them.
In this step only a very simple filter is used, that checked for the following constraints:

- 12 bit frame sync is valid (i.e. `0xFFF`)
- the CRC flag is not set
- it is either MPEG version 1 or 2 and layer 1,2 or 3
- a valid bit rate and sampling was chosen.

```
Version
=======
version 1.0.0.5, April 2006.

Copyright
=========
M. Stevens, 2006. All rights reserved.

Disclaimer
==========
This software is provided as is. Use is at the user's risk.
No guarantee whatsoever is given on how it may function or
malfunction.
Support cannot be expected.
This software is meant for scientific and educational
purposes only.
It is forbidden to use it for other than scientific or
educational purposes.
In particular, commercial and malicious use is not allowed.
Further distribution of this software, by whatever means, is not
allowed
without our consent.
This includes publication of source code or executables in
printed form,
on websites, newsgroups, CD-ROM's, etc.
Changing the (source) code without our consent is not allowed.
In all versions of the source code this disclaimer, the copyright
notice and the version number should be present.
```

Figure 11: License of Stevens collision finding program

The filter also writes all usable collisions it finds to a log file.
A perl script runs the collision finder, takes the output and passes it to the filter. The script is also used, to send mails containing the current status of the system, especially if usable collisions are found, to a pre-defined email address. This is useful, as the collision finder runs over several days on a server.

Steven's implementation As of the license of stevens code, it only allows the use of the software as is, and only for educational purposes.
The license also does not allow to redistribute the code without their consent. (11). For this reason, neither source nor compiled program of Steven's implementation are distributed with the source of the programs described in 6.5.2, and 6.7, as till the deadline, no reply to the request was received. The source of the program can be downloaded from the hashclash website [?], for the use according to the license.

Klima's Implementation Klima's implementation is under a similar license as Steven's, also not allowing the distribution of the source or the program. Additionally it is only for Microsoft Windows, as it requires Windows only headers, and would have to be changed to run on linux servers or other operating systems.

Choice of program Steven's version was chosen for several reasons:

- The implementation is portable, using the platform independent boost library.
- It is fast, which improves the performance greatly, when used in the brute force search, as many runs are needed.
- The source is clean and well readable.

Especially the last point is an advantage, as Klima states in his source:

```
Note: I am very sorry, the program is very bad.
      I am not a programmer.
      On the other hand, there is a possibility to improve it
      and to speed up the collision search.
```

6.5.2 Filter

As the simple filter from 6.5.1 does not check if the mpegMPEG header is really valid, but only if it meets some constraints, an additional filter is needed, that parses the files, checks the headers and also outputs human readable information about it. This information is needed to be able to encode the audio data to the same format. The program takes one collision pair, and the position where the valid MPEG header is. The program is mostly build from the same components used for the interleaving program. It uses the same header class as the interleaver, passes the 4 bytes of the header (8.1) into it, and checks if the header is valid.
The header is written to `stdout` in a human readable form. Again a perl script is used to automatically feed all possibly usable collisions into the program. This script also writes the output to a logfile, that can be searched for a header that fits the needs of the application, i.e. a useful bit rate or a specific MPEG version...

6.6 Preparing the audio files

To allow better compatibility, the audio files are first encoded with an encoding that matches the one found in the chosen collision. Although MPEG audio allows variable bit rate, at least the MPEG version and layer have to be constant through the file.
This is achieved by re-encoding the audio files. For the test runs, the free LAME mp3 encoder was used, as it allows setting most header flags separately by the user, though only encoding as MP3 (MPEG version 1 audio layer 3) is supported.

```
Usage:
mpegInterleave <mpegFile1> <mpegFile2> <collisionFile1>
               <collisionFile2> <collisionPosition> [outFile]
```

Figure 12: Usage of program

6.7 Program

The program was written in C++ using only standard header files, for compatibility reasons. It takes six arguments (five mandatory, one optional) and creates the colliding files from them (Fig. 12). Besides the two audio files prepared as in 6.6 and the collision pair, the program also requieres the offset of the header in the collision file. Last optional parameter is a name for the output files.

6.7.1 Workflow

First both MPEG files are parsed, and stored as lists of packets 8.5 (also 8.2). The audio data is not decoded but copied as is, only the packet headers are parsed into a header structure as in 8.1. While parsing the packets, they are also "cleaned", i.e. except for the headers, every occurrence of `0xFF` in the packet is replaced by `0xFE`. This is done, to keep the player from jumping when reading structures that look like headers while seeking the beginning of the next packet. If not done, players simply try to read the packet, thus leaving the current audio stream played. In most Cases this leads to nearly identical output of both files.

The collision files are parsed in their own structure (8.7), containing all information needed, such as if they contain a valid packet header and the position of the header (also the incomplete header where it would be). In this process they also parse the contained header, if available, to allow to calculate the length the packet should have.
After all data is parsed, the construction of the colliding files itself begins.

The interleaver (8.8 shortens all packets by one byte (the last), then creates a new list of packets, by alternating between the two input lists. This basically creates a MPEG file that corresponds to 5.2.3. As the output of the interleaver is a MPEG file structure (8.5), it could be output at this position, generating a file that on playback sounds like the first input file.
If one of the two files is larger then the other, the shorter file is padded with it's last packet (this could also be changed to some bogus packet).

At this point only one file was created, as the collision is the only thing differing in the output files.
The next step is to attach the collisions. For this, the collision blocks are padded by the collision attacher (**??**), so that the length from the beginning of the contained header (or where it would be, dependend if the collision contains a valid header) to the end of the block is exactly the length the packet should have minus one byte (for the overlapping). The padded collisions are then encapsulated into MPEG packet structures (8.2) and then prepended to the interleaved

MPEG file.
This step is repeated for the second collision block.

Finally the results of this operations are written to disk, as two files, one for each part of the collision pair. These files collide on MD5.

6.7.2 Performance

The whole process takes less then half a second on a 2GHz notebook, for two 4 minute songs, i.e. if the collision is present, the files can be created very efficiently. This especially means, that with given collisions, or a collision database, such collisions can be created on the fly.

6.8 Result

After running the program on the files, the two files indeed collided on MD5. Some players did not play the files. Others played it well. XMMS implements the described in 5.3.1. The playback results are a near perfect match of the two original files. This inconsistent behavior of the players is mainly caused by the lack of a definition of the parser for such cases in the MPEG standard [?] (See 6.1).

6.9 Source

The source of the programs (except the parts that can't be included due to license restrictions) are on CD, and will hopefully be availabe.

6.10 Limitations

As this method relies on the implementation of the player, there are some limitations in the functionality.

6.10.1 Player

As stated in 6.1, the error correction and fault tolerance is completely up to the player for plain MPEG streams and files. This leads to an inconsistent behavior of the created files across different players.
Some players, like iTunes don't play the file at all, others play them well. An example for the latter case is XMMS, that delivers a "perfect" result.

6.10.2 Seeking

Another drawback occurs when seeking in the file. Normally seeking is done by jumping to an other position in the file. This may lead to a switch of the streams, when the first valid packet found after changing the position in the file is a packet from the other stream.

Figure 13: Construction of an MPEG video stream

6.10.3 Filesize

A third problem is the filesize. The files basicly double their size when beeing combined. As multimedia files are rather large in their original form, the increased size can be noticed by the user.

7 MPEG - Video

Although the construction of MPEG 2 video is similar to that of MPEG audio, it is not possible to construct a simple collision with the methods above.

7.1 Basic Construction of an MPEG video stream

Simplified, a MPEG stream is build of 3 types of packet. There are actually some more, but for the matter given, the simplified version with these three types is enough.

- Sequence header, containing most of the information needed for decoding the pictures, such as ascpect ratio resolution and framerate.
- Picture, a picture consists of several slices of data, the picture header contains information necessary to decode the image.
- Slice, a slice contains a part of the actual image information, it's header is only needed to locate the beginning of the slice.

The construction is according to figure 13. The stream must begin with a sequence header, identified by the prefix `0x000001B3`, specifying frame rate, aspect ratio and resolution of the video. Without this information, playback of the video is impossible.
After this initial information, normally, a group of pictures starts (header: `0x000001B8`. As the group of picture header does not contain information vital for the playback, it is ignored here.
The content of a group of pictures, in this case the rest of the file, consists of pictures. The picture header, (`0x00000100`) is needed, for it's sequence number as well as to know where a picture begins or ends.
Each slice of a picture (`0x00000101 - 0x000001AF`) contains a part of the information of the current picture.
This minimum setting is played by most MPEG Video players. It should also be said, that slices do not seem to have a minimum length, so the video also plays with shortened slices (even if some bytes are removed at the end of each slice, this only affects the picture quality.

7.2 Collision with basic method

As stated in 7.1, shortening slices by one or several bytes, only decreaces image quality, without dropping the aditional frame. Therefore, it is impossible to construct a choice mechanism as described in **??**. The only level possibly allowing collisions would be picture level. However, there are methods to allow limited support for collisions.
It has to be said, that the construction does not follow the requirements in 3, as it doesn't consist of individually playable packets, but requires the sequence header to be playable.

7.3 Alternative method

While it is impossible to hide one video completely with the given method, it is possible to have one video dominant. This is done, by interleaving two files with different resolutions. The video with the same resolution as stated in the sequence header, is more dominant in the resulting video, as the other pictures will be disrupted. due to the fact that they do not align with the resolution.
By containing the header similar to the way done in **??**, it would be possible to create two colliding videos with different content. However, as the Sequence header is 11 bytes long, it is hard to find a usable collision for this scenario.

8 Conclusion

The methods in this paper allow the construction of colliding multimedia files on MD5. The same method can also be applied with any other hash algorithm that only does a single pass over the data, i.e. that has the same properties as described for MD5 in 4.2. Although only the exact methods for MPEG are specified, many other packet based formats may be usable in this way.
With regard to the hash algorithm, MD5 is not the only algorithm allowing for this type of attack. Every algorithm that only does one pass over the data, similar to MD5, can be used in the same way, as soon as it is broken for the general case, providing random collision pairs.
For some hash algorithms, this can most likely be done right now [**?**]:

- SHA0
- HAVAL
- MD4
- RIPEMD
- and as shown here MD5.

Also, as almost all hash algorithms do only one pass over a file, this method should be adaptable for most hash functions as soon as they are broken.

Annex A - Documentation of the source

This section contains a reference to the Classes of the programs.

8.1 PHMpegHeader

A class to manage MPEG audio headers.
The class can parse 4 byte MPEG audio headers and makes the information available through it's interface.

8.1.1 Public interface

PHMpegHeader() The default constructor, creating an empty header. This header is not considered a valid header.

PHMpegHeader(unsigned char* header) A constructor, that takes a 4 byte array of unsigned characters and tries to interpret it as an MPEG header. The byteheader is kept along the parsed information. This removes the need for a later synthesis of the header.

Parameter

- `header` a pointer to a `unsigned char` array of length 4 containing an MPEG audio header in byte format.

PHMpegHeader(const PHMpegHeader &two) The copyconstructor for `PHMpegHeader`.

Print(std::ostream &o) The function prints the header in a human readable form to `o`.

Parameter

- `o` a stream.

InterpretHeader(unsigned char* header)

Parameter

- `header` a pointer to a `unsigned char` array of length 4 containing an MPEG audio header in byte format.

This method takes a 4 byte array and interprets it as a MPEG audio header. The byteheader is kept along the parsed information. This removes the need for a later synthesis of the header.

GetByteHeader()

Returns

- returns a pointer to a `unsigned char`, pointing to a copy of the byte-header.

Returns a pointer to a 4 byte representation of the header;

IsValidHeader() Returns true if the header is a valid MPEG audio header.

Returns

- `bool` a boolead value, indicating if the header is valid.

GetPacketSize() The size of the whole packet (based on the information in the header).

Returns

- `unsigned int` the length of the packet.

== and != `==` and `!=`, the equal and the not equal operators are implementet for the class `PHMpegHeader`

8.2 PHMpegPacket

This class encapsulates a complete MPEG audio packet including header.

8.3 Public Interface

PHMpegPacket(PHMpegHeader _header,unsigned char* _body,unsigned int _size) Constructor with three arguments, creates a new packet from a packet header, the packet data as an array and the size of the packet. The default method for creating `PHMpegPackets`

Parameter

- `_header` a `PHMpegHeader`, the header of the packet.
- `_body` a pointer to an array of `unsigned char` containing the packet
- `_size` an `unsigned int` the size of the packet in bytes.

PHMpegPacket(const PHMpegPacket &packet) The copyconstructor for `PHMpegPacket`.

GetBytePacket() Returns a pointer to a copy of the packet data.

Returns

- `unsigned char*` a pointer to a copy of the packet data.

GetSize() The size of the packet in bytes.

Returns

- `unsigned int` the size of the packet in bytes.

GetHeader() The header of the packet as a header object.

Returns

- `PHMpegHeader` the header of the packet.

Print(std::ostream &o) Writes the byte version of the packet! to the stream.

Parameter

- `o` a reference to a stream.

SetSize(unsigned int newsize) Sets the size of the packet, used to shorten the packets.

Parameter

- `newsize` a `unisgned int`, the new size of the packet.

8.4 PHCleanedMpegPacket

Inheriting from `PHMpegPacket` (8.2), adds a functionality to replace all `0xFF` in the body of the packet (not in the header for obvious reasons).

8.4.1 Public interface (additions)

Clean() Replaces all occurrences of `0xFF` with `0xFE` except in the packet header.

8.5 PHMpegFile

This class contains a whole MPEG audio stream, parsed in packets.

8.6 Public interface

PHMpegFile() Default construcor used to get an empty file.

PHMpegFile(const PHMpegFile &one) The copyconstructor for `PHMpegFile`.

PHMpegFile(std::string fileName) Reads the file `fileName` and parses it's contents into the newly created `PHMpegFile`.

Parameter

- `fileName` a `std::string` the path to the file.

ResetPosition() Resets the position of the internal pointer used for traversing the file.

GetNextPacket() Returns the current packet and advances the pointer.

Returns

- `PHMpegPacket` The current packet in the file.

bool Eof() True if the last packet has been read.

Returns

- `bool` last packet read.

AddPacket(PHMpegPacket packet) Appends the `PHMpegPacket packet` to the end of the File.

Parameter

- `packet` a `PHMpegPacket` to be added to the file.

WriteToFile(std::string fileName) Writes the content of the `PHMpegFile` to the file `fileName`.

Parameter

- `fileName` a `std::string` with the path to the file to write the contents to.

GetHeader()
Returns the header of the first packet of the file.

Returns

- `PHMpegHeader` The header of the first packet of the file.

8.7 PHMpegCollision

A class to parse and contain a collision.

8.7.1 Public Interface

PHMpegColisionstd::string fileName,int position Reads a MD5 collision from the file `fileName` and tries to parse a header at positio `position`.

Parameter

- `fileName` a `std::string`, the path to the collision file.
- `position` a `int`, the position of the MPEG header in the collision.

HasValidHeader() True if the collision file contained a valid MPEG header.

Returns

- `bool` flag if the header is valid.

GetHeader() Returns the header contained in the collision.

Returns

- `PHMpegHeader` the header contained in the collision.

unsigned int GetPosition() Returns the position where the header in the collision starts (should start).

Returns

- `unsigned int` the position of the header in the collision

8.8 PHMpegInterleaver

Contains the facilitys to interleave two `PHMpegFile`.

8.9 Public interface

Interleave(PHMpegFile &one, PHMpegFile &two) Takes two `PHMpegFile` and interleaves their packets, while reducing their length by one (and thus overlapping the packets by one byte).

Parameter

- `one`, a `PHMpegFile`, the first of the two files to interleave.
- `two`, a `PHMpegFile`, the second of the two files to interleave.

Returns

- `PHMpegFile` containing the two interleaved `PHMpegfile`s.

8.10 PHMd5CollisionAttacher

Attaches a collision to an `PHMpegFile`

Public interface

PHMpegFile Attach(PHMpegFile &file,PHMpegCollision &collision) Creates a `PHMpegPacket` from a `PHMpegCollision` and prepends it to the `PHMpegFile`.

Parameter

- `file` a reference to a `PHMpegFile`, the interleaved file to prepend the collision to
- `collision` a reference to a `PHMpegCollision`, containing the collision data to be attached.

Returns

- `PHMpegFile` the file with the collision prepended.

References

[1] Information technology - coding of moving pictures and associated audio for digital storage media at up to about 1,5 mbit/s. INTERNATIONAL STANDARD ISO/IEC 11172, 08 1993. Available from: http://www.iso.org.

[2] M. Daum and S. Lucks. Hash collisions (the poisoned message attack) "the story of alice and her boss". Available from: http://th.informatik.uni-mannheim.de/people/lucks/HashCollisions/.

[3] V. Klima. Tunnels in hash functions: Md5 collisions within a minute,. Cryptology ePrint Archive, Report 2006/105, 2006. Available from: http://eprint.iacr.org/2006/105.

[4] J. Randall and M. Szydlo. Collisions for sha0, md5, haval, md4, and ripemd, but sha1 still secure. Available from: http://www.rsa.com/rsalabs/node.asp?id=2738 [cited April 11 2008].

[5] P. Selinger. Available from: http://www.mathstat.dal.ca/~selinger/md5collision/ [cited April 11,2008].

[6] M. Stevens. Fast collision attack on md5. Cryptology ePrint Archive, Report 2006/104, March 2006. Available from: http://eprint.iacr.org/2006/104.pdf.

[7] M. Stevens, A. Lenstra, and B. de Weger. Hashclash. Available from: http://www.win.tue.nl/hashclash/ [cited April 11, 2008].

[8] X. Wang and H. Yu. How to break md5 and other hash functions. EUROCRYPT, 2005. Available from: http://www.infosec.sdu.edu.cn/paper/md5-attack.pdf.