University of Amsterdam
Master Software Engineering

Master's Thesis

Binary Differencing for Media Files

Vladimir Komsiyski

July 2013

Supervisors: Jurgen Vinju
Jeroen van den Bos

Abstract

Modern digital asset management systems face the problem of handling hundreds of thousands of media files. These include multiple versions of the same image file or image editing software document, leading to substantial disk space requirements and long transfer times to and from the clients of the system. In an attempt to reduce the used storage and transmission times, we look into encoding only the differences between two versions of the same file and use them as a substitute for further processing. This work investigates several such approaches and evaluates their benefits and limitations. The variety of different file types and their nature makes it very difficult to identify the best performing method for all file types and further improvement or combination of approaches must be sought.

Contents

Chapter 1

Introduction

1.1 Problem Description and Motivation

The quantity of existing media content grows with rate of *GB/s*. For example, hundreds of billions of photos are taken each year. In addition, the size of the newly created files is greatly increased by the quality of the capturing hardware. Recent cameras can produce High Definition video and images of size that was unthinkable a few years ago.

While the average person can manage the private collection of pictures, movies and audio, the media industry face a new problem when dealing with the vast quantity of data.

Digital asset management systems are used in the international media industry to manage large quantities of media content. Millions of photos, videos, articles, document and presentation files are used by newspaper and magazine publishers, production companies and marketing or sales departments.

Publishing and media editing software is heavily used to modify such files to prepare them for release in a product. Graphics, audio and video editing naturally results in multiple versions of the same file. Traceability and back-up requirements force storing the old version in addition to the new one, which can result in hundreds of different versions of the same file, kept either on a server, or locally. Thus, managing such large quantities of media content is very storage expensive.

In addition to the considerable amount of disk space requirements, the growing file sizes cause the transfer of a newly created file to the server to take a relatively long time. Retrieving an old version from the server also results in a long data transfer time when dealing with large files.

For illustration, a typical publishing company has rates of 10,000 incoming images per hour on average. Most of them are standard JPEG files with sizes around 2 *MB*, which results in bandwidth of about 45 *Mbit/s* - this is 4 times the maximum bandwidth of the wireless standard 802.11b. Furthermore, peaks of up to 50,000 incoming images per hour can be observed, which increases the required maximum bandwidth to at least 200 *Mbit/s* for a single publishing company. In addition, the average rates impose storage requirements of several hundred *GB* added each day. With the continually growing bandwidth and storage capacity it seems to be a matter of time until the problem eventually solves itself, but the size media files and production material is just as likely to show similar growth in the years to come.

In this work we will address these problems by investigating possible solutions for reducing the transfer time and storage requirements of versioned media files.

1.2 Outline

Naturally, reducing the required disk space involves storing less files or minimizing the file sizes. As discussed, availability of all versions of a file is required and does not allow removing any of them. Reducing file size is done by various compression techniques that sometimes result in loss of data. Lossless compression, on the other hand, does not significantly reduce the file sizes. Different compression techniques are discussed in more detail in Chapter 2.

We will exploit the fact that the original file is accessible by both the server and the client. Since any editing takes place on the client, it needs to first retrieve the last (or any) version of the file in interest. The server stores all versions to date, so both the server and the client have access to the file prior to modification.

This allows for the client to only compute a *patch* file - the difference between the original and the new version of a file. Transmitting only the changes made instead of the whole newly created version can dramatically decrease the file transfer time. The patch can then be applied to the original file on the server to recover and store the new version, or it can be stored as-is as a replacement of the file itself, and applied only when this specific version is requested by a client, thus efficiently storing multiple versions of the same file.

In order for this approach to be useful, the time for creating a patch and transferring it to the server has to be less than the time it would take to send the whole file. Furthermore, receiving a patch and applying it has to take less time than simply receiving the new file. The current tools used for this kind of differencing are discussed in Chapter 3.

The fact that the nature of the media files and the most common modifications performed on them are known allows us to optimize the current tools in an attempt to achieve smaller patch sizes and faster run-time. Taking this to a next level: making the assumption that the server has already stored millions of files, similar data blocks might already exist on it. By using a fingerprint algorithm, it may be possible to locate any possible matches for the data blocks in the file being uploaded. This will allow even new files to be uploaded by just transferring a fingerprint and then only uploading data block, for which no matches were found on the server.

The custom approaches are described in Chapter 4, and evaluated along with the off-the-shelf tools with respect to saved disk space and transfer time reduction in Chapter 5. Chapter 6 summarizes the findings and concludes this work.

1.3 Scope

Numerous media types and file formats exist. There are hundreds of different formats only for graphics files. Investigation of audio, video, vector graphics, documents, etc. is well beyond the scope of this work and we will limit ourselves to raster graphics. Still, our approach can be extended for an arbitrary file type, not limited only to media.

Since versioning of images comes naturally with editing, files used by graphics editing and publishing software will be investigated along with the most common graphics formats.

Taking this into consideration, we try to answer the question of whether and when differencing of media files is useful by first looking into the performance of current binary differencing tools on such files, how the images are typically edited, and what remains common among them.

1.4 Related Work

The idea of using file differencing in order to reduce network traffic is relatively recent. The *bsdiff* and *bspatch* tools were created by Colin Percival in the early 2000s for FreeBSD security update compression. Currently they are successfully used by FreeBSD and OS X to transfer software updates to client machines, where the precomputed patches are applied and the new version of software is retrieved.

Mozilla also makes use of these tools in their FireFox web browser updates [11, 15, 16]. Google took this approach one step further with *Courgette*, being developed to reduce the size of the Chromium web browser updates [17]. It uses knowledge of the specifics of the compiled application updates to optimize the performance of the binary differencing tool *bsdiff*, achieving 90% reduction of the size of the patches. The same idea is employed in the current work - performing binary differencing with a custom method extending an off-the-shelf tool, rather than directly with the tool. This allows to input knowledge of the target files to achieve better performance, which is essentially the same idea as in *Courgette*.

Samteladze and Christensen conducted research with a similar objective [16]. Their aim was to reduce the network traffic for smart phone apps. They also used *bsdiff* and showed that downloading only the difference between the current and the new version of an app can reduce traffic by 48% on average.

Their work, however, is focused on executable files. In all of the mentioned research a binary differencing tool is used to reduce the size of executable files similar to their previous versions and thus allowing for the generation of a patch. Knowledge of the executable file structure provides ground for improvement and better results and most of the tools are tuned specifically for differencing executables.

Although *bsdiff* seems to be the most used binary differencing tool, there are numerous other tools, also implementing optimizations for compiled source code differencing. *xdelta* is designed with respect to the generic differencing format *VCDiff* [8, 9, 20] and is a general purpose tool, while many of its derivatives like *edelta* [4] and *zdelta* [18] focus specifically on the performance on executables.

The media files have a very different nature and the successful optimizations already done are practically useless for our purposes. Thus, with the underlying idea remaining the same, different approach will be sought in this work.

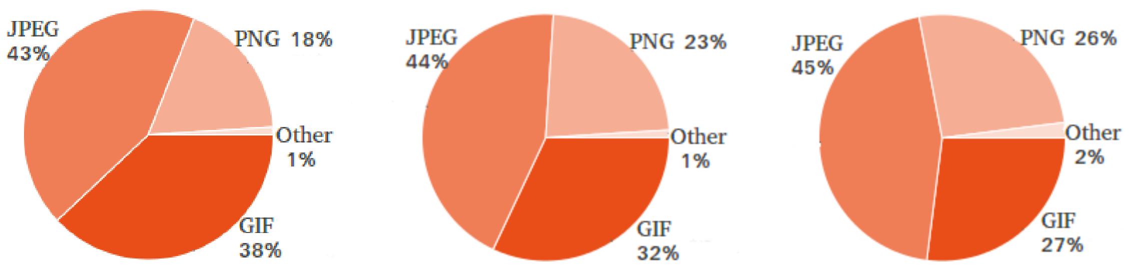
Chapter 2

Media Files

2.1 File Formats

Media files come in vast abundance of various formats. We can categorize the most common types of media as audio, video and graphics, separating the last in vector and raster graphics. We will discuss the common features and the differences among all types and formats but the focus of this work is raster graphics, referred to from here on simply as graphics.

Graphics alone come in a great number of different formats. The most commonly used ones are Portable Network Graphics (PNG), Graphics Interchange Format (GIF) and JPEG compressed files (standard created by the Joint Photographic Experts Group). PNG was designed as a replacement for GIF and TIFF and is steadily taking over as the most widely used lossless graphics format, as shown on Figure 2.1. Although it does not support animation, it offers better compression and more features.



(a) 1 April 2011 **(b)** 1 April 2012 **(c)** 1 April 2013

Figure 2.1: Internet image requests by format according to the HTTP Archive. The trend of substituting GIF with PNG images is visible from the data for the three consecutive years.

The lossy JPEG is the most appropriate format for photographs and is the most commonly used format on the World Wide Web. Many other formats like JFIF, JPEG-LS, JBIG, TIFF, etc. are closely related to JPEG (since JPEG is a compression method rather than a file format).

These are the reasons for the choice of PNG and JPEG - representatives of the two main classes of raster graphics: lossless and lossy. This work will place focus on these two types of graphics files.

Virtually all media formats define data representation, compression method, and a container format. The files have clear and well defined structure, usually consisting of header data and several substructures containing information about the image and the actual data. These substructures are called differently in different format specifications (*chunks* in PNG, *segments* in JPEG, *image file directories* in TIFF, etc.). We will refer to any such substructures as *blocks* for consistency.

The data representation differs among different formats. PNG supports indexed, RGB, and grayscale images with and without transparency. Indexed images define a palette and each pixel essentially contains only an index from the palette instead of full color specification. RGB pixels are encoded with three values for the red, green and blue channels, and optionally a fourth, alpha (transparency) channel. The color space in JPEG is YCbCr, which is effectively a way of encoding RGB information by luminance, blue-difference and red-difference chrominance components.

Although a generic approach is preferred to a number of format-specific ones because extensibility beyond the few investigated formats like PNG and JPEG is desired, it may prove unfeasible due to the differences in data representation among the various formats.

2.2 Data Compression

Regardless of the data representation method, a general purpose differencing tool should be able to encode the color changes between two similar images, based on the raw binary data. Performance on indexed images also should be fair, provided the palette remains the same. However, in almost all graphics formats (including PNG and JPEG) the raw data is compressed using some standard to reduce the file size. Different media types use different compression methods.

GIF uses a dynamic dictionary method for compressing data. It scans the image row by row, discovering pixel correlations within a row, but not between rows. This treatment of a two-dimensional image as one-dimensional space makes GIF compression inefficient and is another reason for its substitution by PNG as the most commonly used lossless format.

The PNG compression is *DEFLATE* [5]. It was originally designed as part of the Zip file format and the most commonly used implementation is part of the *zlib* open source library. The data blocks in the PNG file are effectively compressed data blocks, corresponding to successive blocks of input data. Each block is compressed using a combination of the LZ77 algorithm and Huffman coding. LZ77 is a dictionary algorithm based on the sliding window technique. Its compression is improved when followed by a statistical method like Huffman encoding [15]. Each block therefore consists of two parts: a pair of Huffman code trees that describe the representation of the compressed data part, and a compressed data part [6].

JPEG uses a lossy form of compression based on the discrete cosine transform (DCT), which converts the original data into a frequency domain, discarding high-frequency information like sharp transitions in intensity and color hue [15]. This results in data loss. The compression level can be changed in order to trade off quality for file size. There is an optional lossless mode defined in the JPEG standard; however, this mode is not widely supported in products.

Because of the compression, binary differencing tools like *bsdiff* perform very poorly on PNG and JPEG files (detailed results are presented in Chapter 5). Even a small local modification of the pixel values can propagate changes throughout the compressed data, resulting in an entirely different file, as far as the tool is concerned.

However, since PNG compression is lossless, two images can be decompressed then differenced based on the raw data. The patch produced in this way can then be applied to the original (again decompressed) image to yield the new version without any data loss.

This approach, though, may prove undesirable for JPEG images. Because of their lossy compression, such images suffer from generation loss where repeatedly encoding and decoding progressively loses information and degrades the image. One solution to this problem would be to store an image losslessly (as PNG) and converted to JPEG format only for distribution or preview, so that there is no generation loss. However, this can be impractical because of the larger file size of losslessly compressed images. Another solution is presented in Chapter 4.

2.3 Image Modification

Versioned files come naturally with modification. After making changes to an image, the users want to be able to revert them at any later point in time. This requires storing of all versions of a single file, boosting storage requirements. Dealing with millions of sizable files, like in the publishing industry for example, is a growing concern.

Images are modified with image editing software. Adobe Photoshop is among the most commonly used software for image modification, and the file formats associated with it will be investigated along with PNG and JPEG. GIMP is its free open-source equivalent. Adobe InDesign is a publishing software application also capable of image manipulation. It can create flyers, posters, magazines, newspapers and books, and is therefore widely used by publishers.

The knowledge of the most common changes made to images using such software gives insight on how to best encode them. Newspaper publishers often perform modifications like cropping or scaling an image to fit it on a page alongside the text. Cropping imposes no difficulty in differencing, as the local pixel information in the cropped region remains the same as the original. Scaling, however, results in an entirely different image on pixel level because of the interpolation used when calculating the new pixel values.

Magazine editors usually modify images in a different way. *Image layers* describe different elements and settings of an image while not affecting other parts and facilitate high level modifications and are therefore commonly used by all users of image editing software. Layers are stacked on top of one another and the user can edit them or modify their transparency without changing the other layers. They are most commonly used to add text or company logos, or create effects by compositing multiple images or adding vector graphics shapes.

Common image modifications range from local ones like red-eye reduction to global ones like changing the light's tones and influencing the contrast. When pixels are only locally changed the difference between the original and the new images remains small, consisting only of the modified pixel values. However, adjusting image lightness for example, effectively results in changes throughout the image, making it more difficult to encode the modification efficiently.

Fortunately, image editing software like Adobe Photoshop support non-destructive editing. It allows for making changes without overwriting the original image data. This way the image quality does not degrade after multiple edits. The changes are instead stored as separate layers that do not contain any apparent content, but rather hold adjustments that affect the layers below. This allows for any transformations and effects without directly editing image pixels, greatly facilitating differencing, as only these setting changes need to be encoded. Of course, once an image is exported, all layer information is lost and therefore this reasoning is valid only for the editing software files.

GIMP's XCF files and Adobe Photoshop's PSD files are basically containers for properties and image layers that include the image data. The Adobe InDesign INDD file specification is not available to the public, which further constrains a format-specific approach. Publishing companies maintain large number of such files along with the exported images.

The pixel data in the GIMP's XCF files is either uncompressed, or encoded via Run-Length Encoding (RLE). The PSD files separate the layer information from the raw data, which can be also uncompressed, RLE-encoded, or Zip-compressed. RLE is a simple approach replacing a number of consecutive occurrences with the single value and the length of the run [15]. This method employs the high likelihood that neighboring pixels have the same color. In Chapter 5 we investigate the impact of such encoding on the differencing results.

Chapter 3

Binary File Differencing Tools

The primary use of file differencing is distributing software updates over a network. A target file is compressed with respect to a reference file to produce a patch, normally much smaller in size. Differencing is also used in revision control systems for backing up old versions of files.

There are many binary differencing tools offering various trade-offs with respect to runtime, memory usage and patch size. The performance of any tool largely depends on the differences between the target and the reference files. Naturally, more different files result in larger patches.

We can classify the current (platform independent) binary differencing tools roughly into three families: *xdelta* and its derivatives, *bsdiff*, and *rdiff*. Although other tools (such as *ExeDiff*, optimized for digital UNIX Alpha executables [2]) with comparable performance exist, we constrain ourselves to open-sourced, platform independent tools.

3.1 *VCDiff* and *xdelta*

xdelta is a fast, linear time and space differencing algorithm, based on the *VCDiff* generic differencing format specified by van Hoff [8, 9, 20]. Its underlying principle is to append the target file to the source file and compress the file acquired in this way using the LZ77 or similar algorithm on, only writing upon reaching the target file. Considering many parts of the original file are repeated in the new one, the target is very efficiently compressed because the source file has already been fully processed.

In this sense data compression is a special case of file differencing. If no source file is provided, the result is simply the LZ77-compressed target file.

The generated patch files consist of sequence of instructions, these being `ADD`, `RUN`, and `COPY`. The `ADD` instruction specifies a sequence of bytes to be appended to the target file. `RUN` is used when a single byte is repeated multiple times, and `COPY` indicates a previous occurrence of byte sequence by its pointer and length to be appended to the target [9].

To illustrate the *VCDiff* method, an example (as provided in [9]) is presented. Assume the source and target window (or file) are represented as strings.

Source: $a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l\ m\ n\ o\ p$
Target: $a\ b\ c\ d\ w\ x\ y\ z\ e\ f\ g\ h\ e\ f\ g\ h\ e\ f\ g\ h\ e\ f\ g\ h\ z\ z\ z\ z$

The *VCDiff* instructions encoding the difference are then:

```
COPY  4, 0
ADD   4, w x y z
COPY  4, 4
COPY  12, 24
RUN   4, z
```

To reconstruct the target window, the decoder simply processes the instructions. The first instruction copies the substring $a\ b\ c\ d$ to the target file immediately after the source. Then the four bytes $w\ x\ y\ z$ are appended and the substring at index 4 ($e\ f\ g\ h$) is added. All this results in the encoder's buffer to have this form:

$$\underbrace{a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l\ m\ n\ o\ p}_{\text{source}} | \underbrace{a\ b\ c\ d\ w\ x\ y\ z\ e\ f\ g\ h}_{\text{target}}$$

Now the index 24 indicated by the fourth instruction is in the target file. Considering the source has size 16 and currently the target has size 12, the instruction points to the beginning of the substring $e\ f\ g\ h$ at the end of the buffer and the number of the bytes to be copied is 12. Thus, only the four available bytes are copied three times to achieve this. This illustrates how the target file can also be used as reference during the decoding. Finally, the last four z bytes are added to yield the reconstructed target file.

Other tools like *edelta* and *zdelta* are also based on *VCDiff*. The *zdelta* algorithm uses the *zlib* compression library instead of the LZ77 compression. In its `COPY` instructions it uses multiple pointers to specify the location of the copy [18]. *edelta* is a linear time, constant space differencing algorithm [4]. It modifies the Reichenberger greedy algorithm, which is proven optimal but runs in quadratic time and is therefore not suitable for real-world applications. This way *edelta* sacrifices compression for speed. It has not been thoroughly evaluated but is shown to perform very well on executable files with systematic changes between versions. However, for image data the gain is not present, as seen in Chapter 5.

3.2 *bsdiff*

One of the most widely used tool is *bsdiff*, described by Colin Percival [11]. It has been developed specifically for executable files but produces small patches for any binary files in comparison to other tools. *bsdiff* is quite memory hungry, requiring $max(17*n, 9*n+m)+O(1)$ bytes of memory for producing a patch and $n + m + O(1)$ for recovering the original file, where m and n are the sizes of the old and new files, respectively. The runtimes for *bsdiff* and *bspatch* are in $O((n + m)\log n)$ and $O(n + m)$, respectively. A more sophisticated algorithm producing 4 times smaller patches on average is described in [12] but not yet implemented.

bsdiff's main trait is that it computes approximate matches, extending the perfect ones in either direction. The constructed patches consist of control part with instructions to be executed in order to recreate the target file, the bytewise differences between the approximate matches, and extra bytes from the target file that are not part of a match. The instruction set contains two types of instructions: `ADD` and `INSERT`. `ADD` is similar to *xdelta*'s `COPY` - it specifies a number of bytes to be copied from the old file to the new one. The `INSERT` instruction indicates that a section from the unmatched bytes is to be added to the target. Additionally, the instruction and the matches parts of the patch file are further compressed using the *bzip2* compression library.

The original algorithm relies on common changes observed in executable code, consistently producing smaller patches for executables than other tools. *bsdiff* outperforms the *VCDiff*-based tools in terms of patch size and is preferable for small executable files or when runtime is not an issue. However, with increasing the file size, the non-linear runtime increases very quickly. More detailed test results are presented in Chapter 5.

3.3 *rdiff*

rdiff, part of the *librsync* package, employs a very different technique. It does not require access to both of the files when the delta (difference) file is computed. Instead, only the new file and a *signature* file containing checksums for blocks of the old file are needed. It computes checksums for blocks in the new file and matches them against the old ones to detect the changes.

rdiff differencing is applied in two steps: first, a signature if the original file is generated. Then a patch is produced by this signature and the newer version of the file. Applying the patch to the original file yields the new file without data loss.

Being a general purpose differencing tool, *rdiff* does not use any file-specific knowledge and is therefore outperformed by *bsdiff* and *xdelta* for executable files. The patches in those cases are usually bigger and slower to produce. *rdiff* signatures and deltas are binary files in a format specific to *rdiff*. Signatures consist of a header, followed by a list of checksums for successive fixed-size blocks. Deltas consist of a header followed by an instruction stream, which when executed produces the output file. There are instructions to insert new data specified in the patch, or to copy data from the basis file.

Because block checksums are used to find identical sections, *rdiff* cannot find common sections smaller than one block, and it may not exactly identify common sections near changed ones. Changes that affect every block of the file are likely to cause no blocks to match at all [19].

Chapter 4

Approach

This chapter is concerned with the generation of patches that, as previously discussed, can be used for transfer or storing instead of the new file. We describe four approaches and their trade-offs. The evaluation of the proposed methods is presented in Chapter 5.

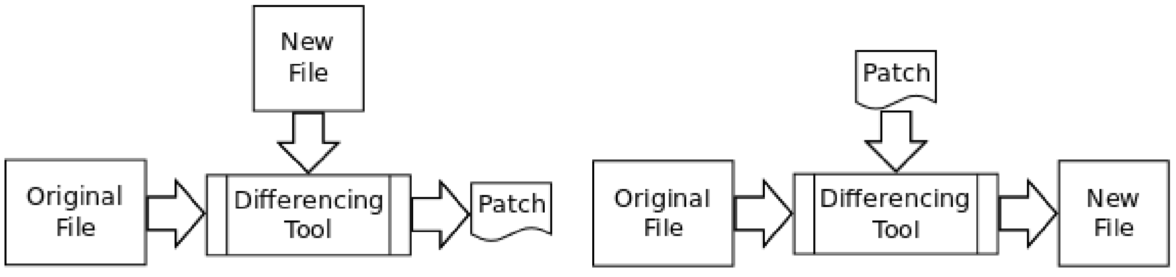
Regardless of the approach, the product of the differencing process is always a patch file. When facing numerous versioned files the method of creating the patches is not trivial. Different patching techniques and their trade-offs are also discussed here.

4.1 Differencing Tools with Media Files

Naturally, we first test the performance of current binary differencing tools directly on the versioned media files. This is a general, format and tool-independent solution, that allows for deployment in any scale and domain.

The approach is simple: provided two or more versioned files, a given differencing tool is run on them to generate the patch. This will give insight on the performance of off-the-shelf tools on compressed image data and the ability to detect differences despite the compression.

The new file is easily reconstructed without data loss by applying the patch to the original file. The pipelines for patch generation and application are shown in Figure 4.1.



(a) Patch generation **(b)** Patch application

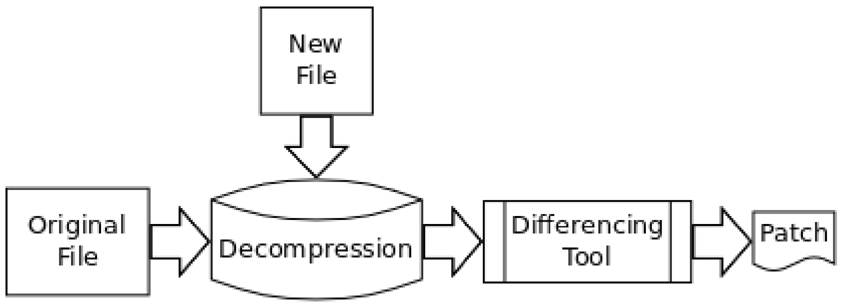
Figure 4.1: Using binary differencing tools directly on files.

4.2 Differencing Tools with Decompressed Media Files

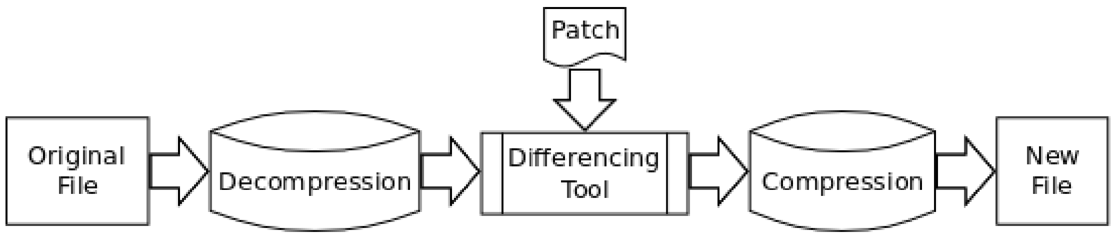
As discussed earlier, the compression used in media files can obstruct the difference computation, since small pixel modifications can cause changes to propagate throughout the compressed file.

In an attempt to improve the performance of the off-the-shelf binary differencing tools, a pre-processing step is added to the generation of the patch. Instead of computing the difference between two files directly, they are first decompressed. Then a tool is run on the decoded pixel data and therefore the patch represents the differences between the pixel values and image metadata in whatever representation is defined by the image format.

Before applying a patch, the orignal file needs to be decompressed again. Then the patching tool performs the recovery of the new file, again in decompressed form. A post-processing compression step, taking into account any metadata like compression method, level, or other image characteristics kept in the patch file, fully reconstructs the target file. The pipelines for patch generation and application with the additional steps are shown on Figure 4.2.



(a) Patch generation with decompression step



(b) Patch application with decompression and compression steps

Figure 4.2: Using differencing tools with pre- and post-processing of image files.

This approach eliminates the performance obstacle created by the compression. However, this does not immediately mean better patches. While the size of the pixel differences are often significantly smaller than the original decompressed image, it may still be the case there is no gain in terms of patch size. Highly compressed images can result in patches with size comparable to the original size, yet smaller than the ones produced after decompression. Furthermore, this method is obviously outperformed in terms of runtime by the direct one because of the additional decompression/compression steps.

Another shortcoming of this method is that it needs knowledge of the image format and the compression standard. Although cross-format differencing may be possible (as long as the pixel data is represented in the same way in both of the differenced images), for each type of image and all possible compression methods that may be used to encode its data, a separate implementation of the additional steps is required.

Furthermore, repeated decompression can cause data loss. As previously stated, JPEG-encoded images suffer from generation loss: every decoding and encoding of the image results in loss of data, which accumulates with every iteration. This loss rates are discussed alongside the runtime and patch size in Chapter 5 to determine the potential gains of employing this method.

4.3 Video Compression

The binary differencing tools perform well on executable files because the knowledge of the internal structure of such files and the common changes made to them are embedded into the tools' design. Created specifically for compiled source code, their performance is much better on such files than on any other binary data, including image files. If the knowledge about the target media files is taken into account during the development of a differencing tool, its performance will be tuned for the specific types and, eventually, this may lead to better results.

Video compression is based on such knowledge. It makes use of the facts that neighboring pixels have close intensities (*spatial redundancy*) and that almost each frame is similar to its predecessors and successors (*temporal redundancy*). Video compression typically encodes each (*inter*) frame in terms of the difference between itself and the previous frame. If this difference is greater than a predefined threshold (or there is no predecessor), the current (*intra*) frame is encoded independently.

The computed difference consists of the coordinates and the value change from the previous frame for each different pixel. The differencing can be performed on blocks of pixels instead of the whole frames, improving the size of the compressed stream. Motion compensation is used to encode small translational movements of such blocks, but fails when an object is rotated or scaled, or when great changes in illumination are present.

To be effective, this kind of compression is usually lossy. Each differencing introduces distortions and successive difference-encoding of frames leads to accumulation of the errors. Total pixel intensity change in a block or detected motion vector below certain thresholds cause the encoder to assume that the blocks are identical and the difference is lost. Furthermore, motion compensation relies on discrete motion vectors and is therefore imperfect. Various motion vector correction and search algorithms have been developed to compensate for the introduced distortions, to detect rotation, scaling and change in illumination of blocks, and to speed up the encoding process [15].

Although compressing video can be quite time consuming, the target data sets consist of only a few images each and encoding should take reasonable time. Even with hundreds of versions for a single file, the resulting video would be only a few seconds of length (at standard frame rates). Also, no audio tracks, subtitles and any other information that may be contained in a video files is present, as we are interested only in the encoded frame difference.

Video decoders are much faster than the encoders because they must play the frames at the desired rate. This means that the application of the encoded frame difference (patch file) to an old frame (original image file) should be less time consuming than its generation.

Custom tools for creation of files containing only the difference between frames and for recovering the new frames are used and tested in Chapter 5 alongside the other proposed methods. These tools are directly derived from the free *ffmpeg* and *libavcodec* solutions.

4.4 Signatures and Fingerprints

Another possible solution is similar to the one used by *rdiff*. Provided that a large number of files is already available on the server, only a signature (or fingerprint) can be transferred instead of the whole file. The server then can detect matching data blocks and request only the ones that were not found to be sent by the client.

However, this requires the rolling checksum to be computed for all server files candidate for a match. While this can be feasible for small local changes, the compressed image files containing large differences may well result in no matches.

A data signature, resilient to modifications is *digital fingerprint* [13]. It is typically used in problems of needle-in-a-haystack type, determining whether a small data entity is contained in a large data base. The current problem, however, is not of the same type: knowing that a new file has similarities to a small set of available files (the old versions on the server), a way to detect *how* the new data differs and to recover it is sought.

Back to the *rdiff* signatures, they are computed for the old file and used along with the new one to create a patch. In our case a signature of the new file and multiple old ones should be used for the recovery. Simply trying to find matches of the new file blocks is less effective than using a binary differencing tool because change detection is constrained by the size of the blocks and failure to find a match requires the whole data block to be transferred from the client. Also, this approach does not make use of the availability of the version of the file prior to modification.

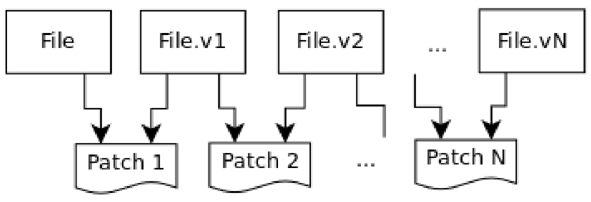
The server eventually only has to store one (or a few) full version of the original file along with a set of patches. Efficiently identifying identical sectors in different files is shown to be very rare, especially with compressed, high entropy data such as image data [21]. Therefore this approach will probably yield insignificant results.

Considering the shortcomings of this approach, the usage of the original *rdiff* algorithm is most appropriate. It will be evaluated in Chapter 5 alongside the other differencing tools.

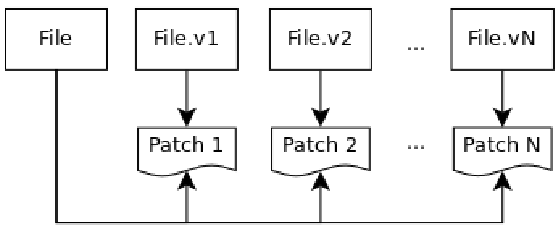
4.5 Patch Generation

4.5.1 Sequential Patching

Perhaps the most intuitive way to generate multiple patches is to always use the immediately preceding version as input for the differencing (Figure 4.3a). Considering that changes accumulate over multiple versions, the resulting files become less and less similar to the original



(a) Sequential patch generation



(b) Root patch generation

Figure 4.3: Main patch generation techniques.

and this *sequential* approach guarantees that the patches are as compact as possible, produced using the file that is most similar to the new one.

The original file is merely another version (the zeroth one) and does not receive any special treatment apart from that it must always be accessible in order to recover any other file. It is kept in its entirety on the server and is used as a starting point for the recovery of any latter version.

The retrieval of any given version requires multiple application of patches, starting with the original unmodified file and recovering all versions before the desired one. This essentially hinders the data transfer from the server to the client. Upon request of version N, first N patches need to be applied on the server and then the last recovered file sent to the client. It is immediately obvious that the time is actually more than the required when no differencing was done.

When a user modifies a file, the client machine performs the differencing and sends the patch to the server. No further action is needed, as the patch is generated by the last version.

4.5.2 Root Patching

A solution to the multiple patch applications is given by the *root* patching approach. It always uses the same original file and each version as input for the differencing, as shown in Figure 4.3b. This way any given version is recoverable after only a single patch application to the original file. This approach is also beneficial when only small changes are present in each version, or when many version files are produced after directly modifying the original.

However, this introduces another problem. A client can send only a patch created with the previous version as input. When the server receives such patch, it needs to recover the file and run the differencing again with the original file instead. The trade-off offered is faster access time for the client at the cost of computationally intensive and slower uploading to the server. The client, of course, does not experience the slow-down during storing the new version.

4.5.3 Reverse Patching

Both *root* and *sequential* patching can be altered by reversing the order of files (and patches, respectively). Provided the newest version of a file is the most often accessed one, it may be beneficial to keep it and create all patches from it instead of the original unmodified file. Considering that users usually add data to images, rather than remove, this reversal may also result in smaller patches (as the target files will be smaller). Also, if a differencing tool causes data loss, it will not be introduced until the version is already outdated.

A *reverse sequential* approach thus would require one additional operation. Upon receiving of a new version encoded as the made changes to the previous version, the server needs to recover it and then generate a patch for the old version. The first step can be eliminated if the whole file is received; however, this only addresses the problem with storage requirements and does not reduce the data traffic.

A *reverse root* patching would have the same problem when a new version is received. In addition it would require every single patch to be regenerated on the server because the root file has changed.

4.5.4 Comparison

In order to compare these three approaches to the default one where no differencing is performed, let us take a look at the delays the server and its clients experience during the transfer of a file in either direction and the effect on the used disk space on the server.

The download time, as seen from both the client and the server, starts after the server has received a request for a file transfer and ends when the file is sent. For the client the upload time can be different from the one experienced from the server. Any operations done to the file prior to its upload are not seen by the server, and respectively, the client is not concerned by any further processing on the server after it has received the file. Tables 4.1 and 4.2 therefore present the requirements from the viewpoints of both participants in the transfer.

Method	Download time	Upload time
None	S/D	S/U
Sequential	$NT_{\Pi}^{Seq} + S/D$	$T_{\Delta}^{Seq} + c^{Seq}S/U$
Root	$T_{\Pi}^{Root} + S/D$	$T_{\Delta}^{Seq} + c^{Seq}S/U$
Reverse Sequential	S/D	$T_{\Delta}^{Seq} + c^{Seq}S/U$
Reverse Root	S/D	$T_{\Delta}^{Seq} + c^{Seq}S/U$

Table 4.1: Patching methods performance from the viewpoint of the client.

D and U denote the client's download and upload speeds respectively and S - the expected file size with the assumption that the modified version has approximately the same size as the original one. T_{Δ} and T_{Π} are the respective method's expected times that the differencing and patching take to complete. c is the achieved compression rate, such that a patch is $1/c$ times smaller than the version file it substitutes. All times are for transfer of the N^{th} version of a file.

The *root* methods are preferred when requests for file versions other than the last one are often made. However, this is not the case in the publishing industry where users generally modify the last changed file and rarely need older versions. The *sequential* method outperforms the rest

Method	Download time	Upload time	Storage requirement
None	S/D	S/U	NS
Sequential	$NT_{\Pi}^{Seq} + S/D$	$c^{Seq}S/U$	$S + (N-1)c^{Seq}S$
Root	$T_{\Pi}^{Root} + S/D$	$c^{Seq}S/U + T_{\Pi}^{Seq} + T_{\Delta}^{Root}$	$S + (N-1)c^{Root}S$
Reverse Sequential	S/D	$c^{Seq}S/U + T_{\Pi}^{Seq} + T_{\Delta}^{RevSeq}$	$S + (N-1)c^{RevSeq}S$
Reverse Root	S/D	$c^{Seq}S/U + T_{\Pi}^{Seq} + NT_{\Delta}^{RevRoot}$	$S + (N-1)c^{RevRoot}S$

Table 4.2: Patching methods performance from the viewpoint of the server.

in terms of upload time and the *reverse* ones do not cause slower the download as no patching is done prior to file transfer. However, the *reverse root* method will not be evaluated further because of its requirement for N differencing runs upon receiving of a file. We will show that the process of creating a patch is the most computationally intensive and takes longer compared to the patch application.

The two *sequential* methods offer a trade-off in terms of download/upload times. Their performance also depends on the requested version, with the *reverse* one being better for newer versions.

The saved disk space on the server depends only on the achieved compression and the expected number of versions in the dataset, as all methods require storing one file in its original form and all its versions are represented by only a patch. As already discussed, we investigate the *root* approach and the two *sequential* ones to determine this eventual gain.

In order to compare the methods, *gain coefficients* (Table 4.3) are introduced. They illustrate how well a given approach performs in comparison to the case when no differencing is done. In all cases the smaller means better and the coefficients are computed as the ratio of the performance of the current approach and the default one, preserving the consistency. The upload gain coefficient from the viewpoint of the client is always $c^{Seq} + UT_{\Delta}^{Seq}/S$ because the client always perfroms differencing of two consequtive versions (*sequential* differencing).

Method	Download	Upload (server)	Storage
None	1	1	1
Sequential	$1 + DNT_{\Pi}^{Seq}/S$	c^{Seq}	$((N-1)c^{Seq} + 1)/N$
Root	$1 + DT_{\Pi}^{Root}/S$	$c^{Seq} + U(T_{\Delta}^{Root} + T_{\Pi}^{Seq})/S$	$((N-1)c^{Root} + 1)/N$
Reverse Sequential	1	$c^{Seq} + U(T_{\Delta}^{RevSeq} + T_{\Pi}^{Seq})/S$	$((N-1)c^{RevSeq} + 1)/N$
Reverse Root	1	$c^{Seq} + U(NT_{\Delta}^{RevRoot} + T_{\Pi}^{Seq})/S$	$((N-1)c^{RevRoot} + 1)/N$

Table 4.3: Gain coefficients for the different patching methods. The upload gain coefficient from the viewpoint of the client in the case of differencing is always $c^{Seq} + UT_{\Delta}^{Seq}/S$. Smaller means better.

Chapter 5

Evaluation

5.1 Test Environment

To test the performance of the described methods, file stores of two independent publishing companies are used, which will be referred to as *Dataset A* and *Dataset B* for consistency. The data is an exact copy of the real-world working environment in these companies.

To be able to thoroughly evaluate the tools, a deep understanding of the test environment and the sample files is needed. A summary of the number of files and the file sizes is available in Appendix A. Regardless of the choice of original file and patching method, the number of version files and patches remains the same.

From Figure 5.1 it is clear that InDesign and Photoshop documents are the most critical file formats as far as saving storage is concerned. Appendix A confirms that the file size plays bigger role than the number of the files. There is also evaluation of the expected file size for each format in both datasets.

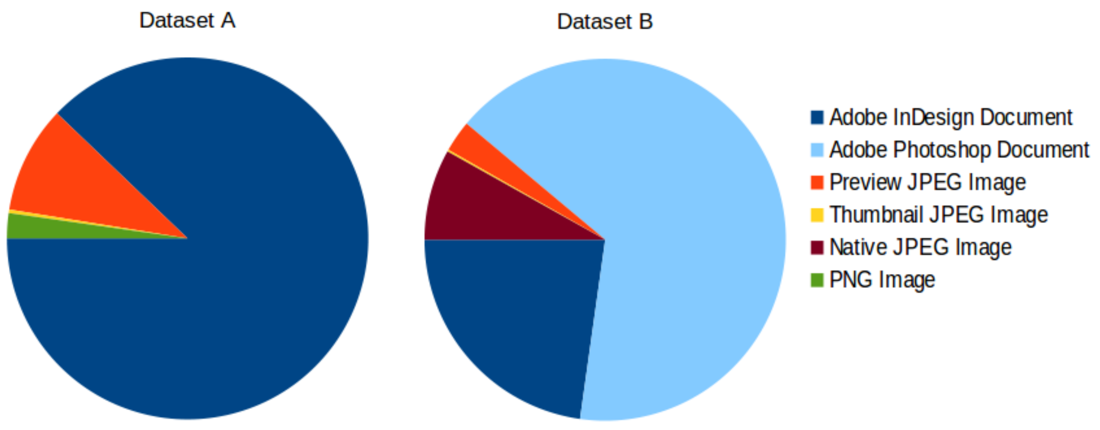


Figure 5.1: Storage requirements for all file types in both datasets. Dataset A occupies 13 GB of disk space in total and Dataset B occupies 349 GB.

We compare the differencing methods in terms of the time needed to produce and apply a patch, the achieved compression, and the potential data loss. All tests are run on a 2.66 GHz Intel Core 2 Duo processor. The memory size (4 GB) is much larger than any of the file sizes. Memory consumption is not subject to this investigation, considering that modern machines can handle differencing of this scale without any issues.

5.2 Binary Differencing Tools

This section presents the tests of off-the-shelf binary differencing tools applied directly to the target files. First, to get an overview of the performance of the tools, tests only on the smaller Dataset A are performed. The *sequential* method is used to produce all patches. Next, in more detail, further investigation of the conditions that were found to have the best impact on the results is provided.

5.2.1 Patch generation time

bsdiff is severely outperformed by the other tools for big file sizes (Table 5.1). Its non-linear runtime will grow much more than a few seconds when faced with larger files like the Adobe Photoshop documents. Considering that the differencing will take place on a client machine, before transferring the patch to the server, runtime of the magnitude of minutes can prove unacceptable. *xdelta*, *edelta* and *rdiff*, however, perform very well in terms of time needed to produce a patch for files of such size. While for small files *xdelta* is outperformed by the other tools, the difference is a matter of a few milliseconds and hardly has any impact in practice.

File Type	*bsdiff*	*xdelta*	*edelta*	*rdiff*
Adobe InDesign Document	2358.1 ms	237.7 ms	623.1 ms	168.7 ms
PNG Image	337.8 ms	139.3 ms	117.6 ms	51.4 ms
Preview JPEG Image	373.4 ms	136.6 ms	96.7 ms	42.8 ms
Thumbnail JPEG Image	17.8 ms	66.2 ms	16.1 ms	20.8 ms

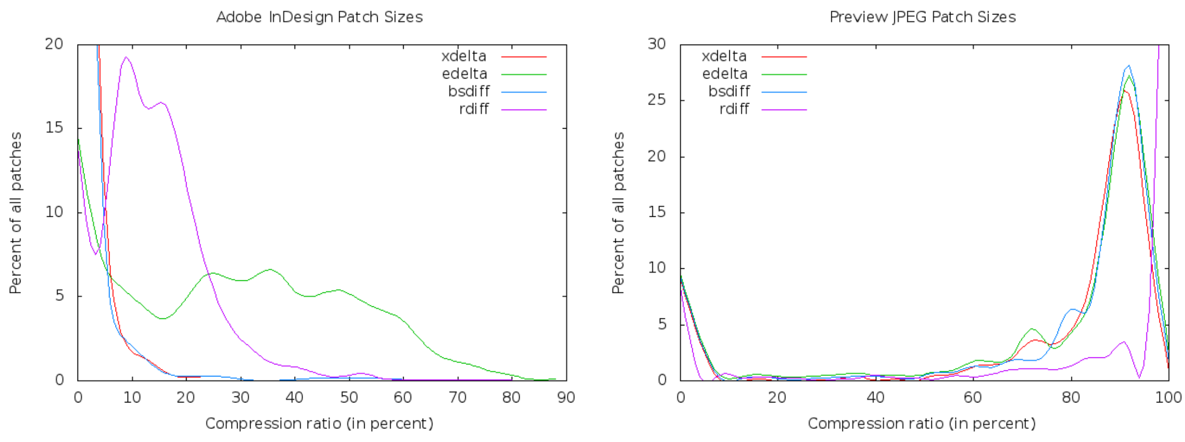
Table 5.1: Expected time to produce a patch on files from Dataset A.

5.2.2 Compression

The InDesign files exhibit very compact patches, as seen on Table 5.2. *bsdiff* and *xdelta* show very similar rates, outperforming the other tools. For the Adobe InDesign files the differencing results in 25-30 times smaller files. This does not leave much room for improvement, especially considering that the InDesign files are closed format and we can only speculate what their

File Type	*bsdiff*	*xdelta*	*edelta*	*rdiff*
Adobe InDesign Document	3.6%	3.7%	27.8%	12.8%
PNG Image	98.3%	98.2%	98.1%	98.2%
Preview JPEG Image	87.1%	86.2%	87.0%	96.1%
Thumbnail JPEG Image	92.7%	87.6%	89.3%	96.2%

Table 5.2: Expected relative patch size in Dataset A. It is computed as the ratio between the sizes of the patch and the new file used to produce that patch. Smaller means better.



(a) Adobe InDesign files **(b)** Preview JPEG Images

Figure 5.2: Patch size distributions for all four tools.

structure is and how raw data is stored, though the results suggest that it is uncompressed. The results also imply that the Adobe documents are very similar.

The simpler algorithm of *rdiff* results in an increase in the patch size. While the 12.8% ratio can be satisfactory in some cases, *rdiff* is well outperformed by both *bsdiff* and *xdelta*. *edelta* is performs the worst from all tools, as its patch sizes are not as consistent (Figure 5.2a).

For all types of image files and for all tools the achieved compression is virtually the same and is very poor. Considering that the JPEG files correspond to the InDesign files, it is safe to conclude that even small changes (present in the InDesign files as seen by their patch sizes) propagate throughout the whole compressed image file. Another solution is required for this type of files.

Figure 5.2 shows the distributions of the size of the produced patches by the four investigated tools on Adobe InDesign and preview JPEG files from Dataset A. From 5.2a it is clear that most files are very similar and *bsdiff* and *xdelta* produce very compact patches. *rdiff*'s patches are larger, and *edelta* shows very inconsistent performance. 5.2b shows the poor performance on compressed image files (preview JPEG in this case). The jump of the rate of very well compressed files is explained by the fact that a few version files do not encode any changes and are instead exact copies of each other. When faced with modified files the tools produce patches with size almost the same as the files themselves, as is the case for the majority of these image files. The patch size distributions of the other compressed image file types are very similar to the one on Figure 5.2b.

In conclusion, *xdelta* shows the best performance on Adobe InDesign documents. Although *bsdiff* achieves slightly better patch size, its run-time for large files is much slower. *edelta* and *rdiff* are faster than *xdelta*, but *edelta* shows inconsistency in the size of its patches. The performance on compressed images is far from satisfactory and these tools will not be investigated further on such files.

5.2.3 Patch application time

File Type	*bsdiff*	*xdelta*	*edelta*	*rdiff*
Adobe InDesign Document	72.7 ms	36.0 ms	115.5 ms	61.8 ms
PNG Image	33.3 ms	15.0 ms	16.9 ms	13.9 ms
Preview JPEG Image	41.5 ms	15.2 ms	20.8 ms	14.6 ms
Thumbnail JPEG Image	13.0 ms	13.1 ms	12.8 ms	12.6 ms

Table 5.3: Expected time to apply an existing patch to recover the new version.

In contrast to differencing, patching requires much less time (Table 5.3. It is done by the differencing tool by simply executing the instructions encoded in the patch file. Runtimes of this order of magnitude are very satisfactory.

5.2.4 Detailed Results on Adobe Document Files

The performance of the binary differencing tools on compressed image files is extremely poor. The produced patches are almost as big as the target files. For this reason only Adobe document files will be considered for further investigation of the off-the-shelf binary differencing tools.

In both evaluated datasets the Adobe InDesign and Photoshop documents together take 85% of all disk space. This makes finding an optimal solution for this type of files more beneficial than one for image files and is in the main focus.

	bsdiff	*xdelta*	*edelta*	*rdiff*
T_Δ, s	2.358	0.238	0.623	0.169
T_Π, s	0.073	0.036	0.116	0.062
c	0.036	0.037	0.278	0.128

Table 5.4: Summary of the performance of OTS tools on InDesign documents from Dataset A. Smaller means better.

Having gathered the data in the previous sections (Table 5.4), now we can evaluate the performance of the tools in accordance to Table 4.3. The expected file size is taken from Table A.3. We consider client download and upload speed of 10 *Mbit/s* and 1 *Mbit/s*, typical for the standard packages of most Internet providers. Further discussion of the speeds at which differencing is still beneficial follows.

Gain	*bsdiff*	*xdelta*	*edelta*	*rdiff*
Download	1.027	1.013	1.043	1.023
Upload (client)	0.123	0.046	0.301	0.134
Upload (server)	0.036	0.037	0.278	0.128

Table 5.5: Gain coefficients of OTS tools on InDesign documents from Dataset A. Number of versions $N = 1$ is used. Smaller means better.

The gain coefficient for storage is not calculated the same way as suggested in Table 4.3, as it depends on the expected number of versions per file and is therefore much more dataset-specific than the other coefficients. However, it converges to c for large N and is equal to 1 when no versions are present. Of course, the differencing is performed only when at least one version is created, and in this case the coefficient is calculated by $(c + 1)/2$, which is proportional to c. It will decrease when more versions are introduced to the dataset, thus saving more space (considering that $0 \leq c \leq 1$), while remaining proportional to c. Since for now we are only interested in comparing the performance of the tools, just taking the compression rate is a good enough indicator of the saved storage space. This results in identical to the server upload gain coefficients.

bsdiff is marginally better than *xdelta* as far as compression is concerned. *rdiff* and especially *edelta* generate much larger patches. The upload gains from the viewpoints of the server are exactly the same as the compression rates and the saved disk space is proportional to it. For example, an expected number of versions per file of 3.7 (which is the case in Dataset A) and differencing with *xdelta* result in 70% less required disk space.

As expected, *bsdiff*'s slow differencing time results in 3 times slower upload from the viewpoint of the client when compared to the server's experienced time. Apart from that, the added time due to the differencing is smaller compared to the transfer of the produced patch and lead to up to 21 times faster transfer than when no differencing is done. Even the worst performing tool *edelta* reaches a speed-up with a factor of 3. *xdelta*, showing the best results out of the four tools, achieves a 21x speedup when transferring a file to the server. The second best tool, *bsdiff*, is only 9 times faster. However, its coefficient may dramatically increase with increasing the file size (Adobe Photoshop files) because of the non-linear runtime of this tool. *edelta* is severely outperformed by all other tools. The fastest tool *rdiff* loses its lead due to the bigger patches it produces.

The download time when using differencing is, of course, slower than when the whole new file is sent to the server. However, the loss is only 4.3% with the worst performing tool and the benefit of the much faster upload is overwhelming.

It is worth noting that increasing the download speed of the client essentially makes the performance worse. For example, a 10 times faster (100 *Mbit/s* for download and 10 *Mbit/s* for upload) connection would result in 15% slower access to the file, in which case the trade-off becomes more obvious. However, upload would still be more than 10 times faster. In fact, the differencing loses its gain only with upload speed of more than 120 Mbit/s. Even then, the disk space benefits are obvious.

5.2.5 Varying the patching method

So far only the *sequential* patching method and Dataset A have been evaluated, as it is irreplaceable part of the differencing on the client side. Now *root* and *reverse sequential* patching are performed on both datasets in an attempt to evaluate the possible gains of these alternative approaches. The *reverse sequential* method simply reverses the order of the patching while the *root* one suggests always taking the original file as input for the differencing instead of the previous version. *xdelta*, found to be the best performing tool on Adobe document files, is used in all of the following tests (Table 5.6).

File Type	Method	Delta time	Compression	Patch time
Adobe InDesign Document Dataset A	*Root*	226.9 ms	1.7%	40.5 ms
	Sequential	237.7 ms	3.7%	36.0 ms
	Rev. Seq.	247.0 ms	3.7%	39.9 ms
Adobe InDesign Document Dataset B	*Root*	292.8 ms	10.4%	38.9 ms
	Sequential	242.7 ms	3.4%	33.9 ms
	Rev. Seq.	242.2 ms	2.3%	30.2 ms
Adobe Photoshop Document	*Root*	2718.2 ms	47.1%	223.6 ms
	Sequential	2361.5 ms	38.2%	229.4 ms
	Rev. Seq.	2843.7 ms	37.4%	232.1 ms

Table 5.6: Comparison of the performance of patching methods with *xdelta* and Adobe InDesign and Photoshop documents from both datasets. The *Delta* column indicates the time needed to generate the patch (performing delta) and the *Patch* column - the time needed to apply the patch (performing patching). The compression is again the ratio of size of the generated patch to the size of the file used as input. Smaller means better.

Conterintuitively, the sequential methods produce slightly bigger patches on average for Dataset A. This is due to the fact that many versions essentially encode the undone changes to the preceding version, e.g. *file.vN* is modified to produce *file.vN+1* and then reverted resulting in *file.vN+2*, which becomes much more similar to *file.vN*. This causes the differences between two consecutive versions to be greater on average than the differences between a given version and the original. On Figure 5.3 the exported thumbnails for a typical InDesign file are shown, illustrating the modifications. Still, the difference is hardly noticeable, as seen on Figure 5.4a. This is not the case for Dataset B though. The behavior is as expected and the *root* method is

(a) Version 1

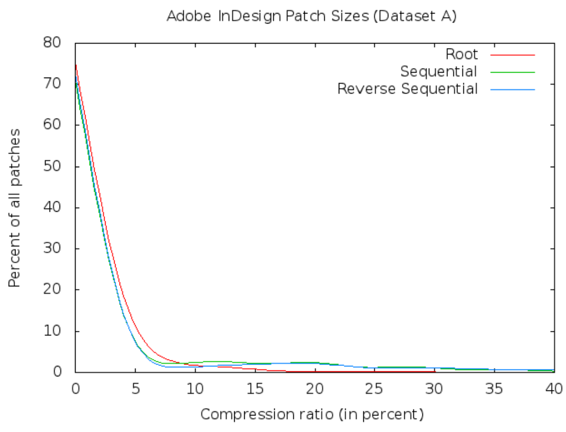
(b) Version 2

(c) Version 3

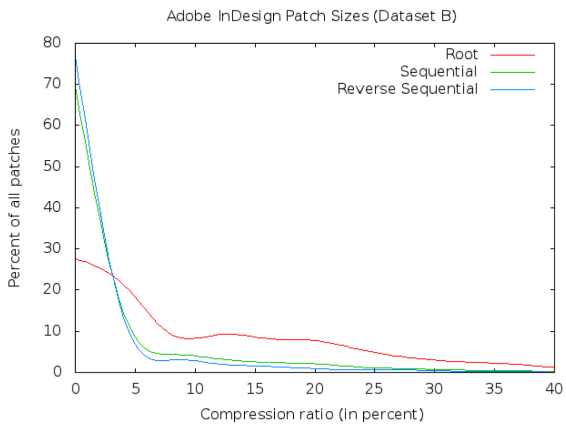
(d) Version 4

(e) Version 5

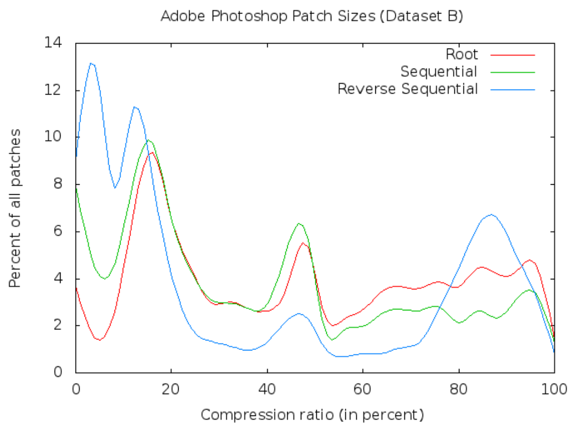
Figure 5.3: Illustration of reverted changes over several file versions.



(a)



(b)



(c)

Figure 5.4: Comparison of different patching methods on Adobe files in both datasets.

outperformed by the *sequential* ones in terms of compression (Figure 5.4b).

The much larger average size of the Photoshop files causes longer differencing and patching times but they remain linear with respect to the input size. The compression in this case is much worse too, but also inconsistent - the distributions of the patch sizes is shown on Figure 5.4c. A closer look at those files reveals that in the vast majority of them the data is compressed using PackBits, a Run-Length Encoding of data. This type of compression is simple and results in better patches than JPEG and PNG, but still results in a drop in performance. The few files having their image data in raw format exhibit very good compression level. None of the encountered Photoshop files are Zip-compressed.

File Type	Method	Download	Upload (server)	Upload (client)
Adobe InDesign Document Dataset A	*Root*	1.015	0.047	0.046
	Sequential	1.013	0.037	
	Rev. Seq.	1.000	0.047	
Adobe InDesign Document Dataset B	*Root*	1.014	0.048	0.044
	Sequential	1.012	0.034	
	Rev. Seq.	1.000	0.046	
Adobe Photoshop Document	*Root*	1.019	0.406	0.402
	Sequential	1.019	0.382	
	Rev. Seq.	1.000	0.407	

Table 5.7: Gain coefficient comparison of the performance of patching methods with *xdelta* and Adobe InDesign and Photoshop documents from both datasets.

Table 5.7 shows the gain coefficients of the different patching methods implemented with *xdelta* in order to compare them.

Interestingly, the InDesign documents from the two datasets show very similar results despite the difference in the performance of the methods. The *sequential* method outperforms the other two in both cases. Clearly, the benefits of each of the methods does not largely depend on the type of modification typically done to the files and the specifics of the dataset.

The Photoshop documents exhibit much worse gain. Still, the benefit of using file differencing is obvious. When faced with large files with sizes of the order of tens of Megabytes, the upload time can be several minutes, in which case a 60% speed-up is worth considering. It is noticeable that when the compression is not as good the time needed to produce a patch is much smaller than the time needed to send the patch. This is the reason *bsdiff* was also evaluated as its smaller patches could make up for the longer runtime. It indeed showed better compression rates than *xdelta*, although still inconsistently distributed. The benefit is of up to 22% smaller patch size, but it is surpassed by over 20 times slower differencing and around 10 times slower patching. This results in much worse upload gain - more than twice poorer than *xdelta*'s, thus confirming the initial hypothesis that *bsdiff*'s runtimes prove a huge drawback of the tool for large files.

The download time can remain unaffected by using the *reverse sequential* method. However, if processing on the server side is undesirable, the *sequential* approach should be employed. The downside of this is the multiple patch applications. The download time would therefore increase with each new version coming to the server. It can, however, be bounded by saving a recovered file after every 4 or 5 versions for example. This way the starting point for the sequential patching guarantees only a few patch applications and the download should never bewith more than 10% slower. The optimal setup depends on the dataset and its expected number of versions per file.

The *root* patching method will not be evaluated further. Its advantage is the need of only one patch application, but this is not as beneficial due to the very little time needed for performing it. Furthermore, *reverse sequential* offers the same and it achieves better compression. On the other hand, they both require differencing to be done on the server for each upload. Assuming that no additional processing power should be used on the server side (because there are many clients and version files come at a high rate, for example), the *sequential* approach is used in all following tests. Even when it is outperformed by any other method, the lead is not substantial and does not cause drastic changes in performance in practice.

In the end, the performance of all patching methods is similar, although related to the trends in the dataset, which in turn are caused by the working environment and common practice in the company, or the way files are modified. This is why a different patching method may be best suited for the particular company, setup and dataset, despite our findings that the *sequential* performs the best.

5.3 Binary Differencing Tools with Decompressed Image Files

The direct application of binary differencing tools on compressed image files is not beneficial in any way. In an attempt to improve the compression rates, we employ the approach described in Section 4.2. Prior to differencing, the image data is decompressed using a standard library dealing with the specific format, and then the raw pixel data is used as input for the differencing tool. After applying a patch to an again decompressed original file, the resulting recovered file is in its raw form. Then a compression step follows, yielding the complete reconstructed version image.

The PNG files can be decompressed and compressed multiple times without any data loss. The JPEG format, however, is lossy and every decompression corrupts the image quality. Unfortunately, the loss is impossible to be quantitatively evaluated, as it is not reflected in the image size. In most cases it cannot be visually perceived, which can be considered satisfactory, especially for thumbnail and preview images. Furthermore, upon opening for modification by any image software the file is decompressed and therefore the loss is already present. Storing the images in JPEG format already suggests that data loss of this type is acceptable.

The following set of tests is performed with *xdelta* and *sequential* patching method. The runtimes include also the decompression and/or compression times performed before or after the actual differencing.

As seen on Figure 5.5, the patch files in many cases are actually bigger than the original image. This is due to the fact that after decompression, the image data explodes in size and it

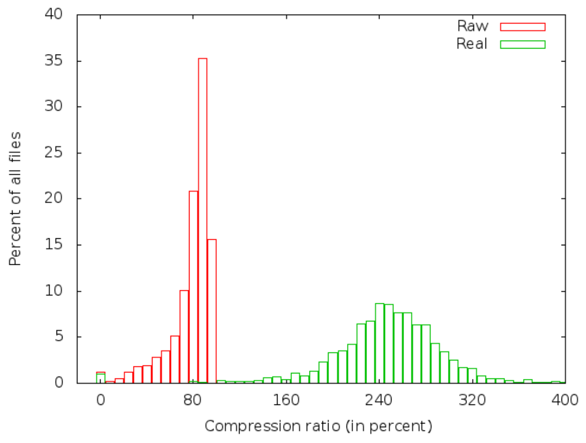


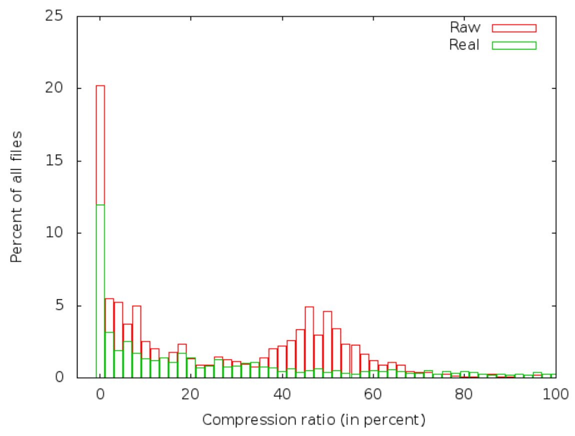
Figure 5.5: Patch sizes of native JPEG image files from Dataset B. *Raw* entries indicate the ratio of the patch size to the decompressed image and cannot be more than 100%. *Real* entries represent the ratio of the patch size to the original file.

can happen that even after it is reduced by the differencing, it is still several times as big as the input image. Patches that are larger than the input files are considered invalid and are ignored. Instead of them, the actual file should be sent to the server, which yields compression rate of 100%.

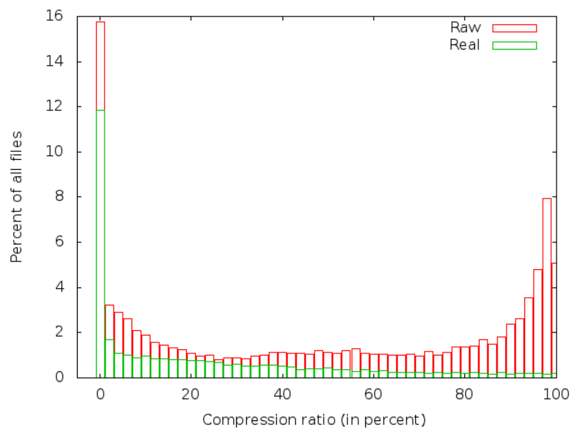
File Type	Delta time	Raw compression	Compression	Patch time	Efficiency
Thumbnail JPEG A	16.4 ms	26.5%	20.6%	14.1 ms	46.2%
Thumbnail JPEG B	15.8 ms	50.2%	19.3%	12.9 ms	33.5%
Preview JPEG A	339.7 ms	13.7%	21.3%	127.0 ms	46.2%
Preview JPEG B	275.4 ms	44.6%	22.4%	85.3 ms	15.5%
PNG	266.9 ms	27.6%	1.0%	340.9 ms	47.5%

Table 5.8: Performance of the differencing on image files. Compression is considered beneficial if the patch size is less than 80% of the version file used as input. The efficiency column indicates how many of the files show such compression. The rest will be considered with compression of 100% and their patches will be ignored. Instead, the whole files should be sent to the server.

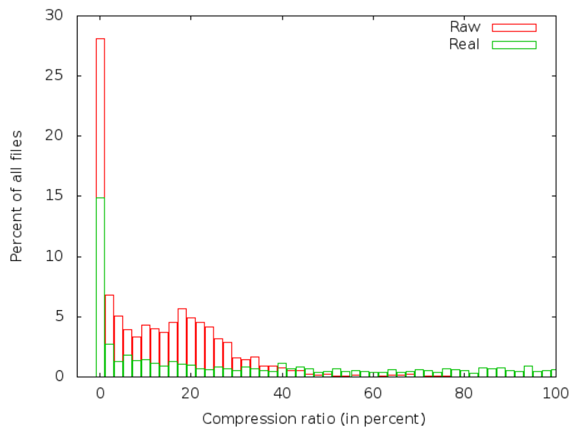
Figure 5.6 shows the achieved compression rates for all types of image files. However, it is obvious that this way of differencing has an effect - the patch to (decompressed) version file ratio drops significantly when compared to the direct tool application. Of course, the differencing should be terminated when the patch size reaches a predefined threshold, for example 80% of the original size. The rest are discarded because the patches are bigger than the actual files and their compression is effectively 100%. Still, many of the patches exhibit very good compression, often below 1%. These results are taken into account in Table 5.8. The last column in it indicates the number of files, for which the differencing is actually beneficial. The rest of the files should be



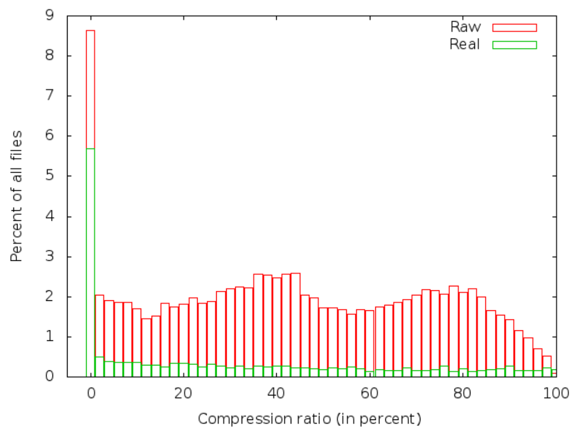
(a) Thumbnail JPEG Dataset A



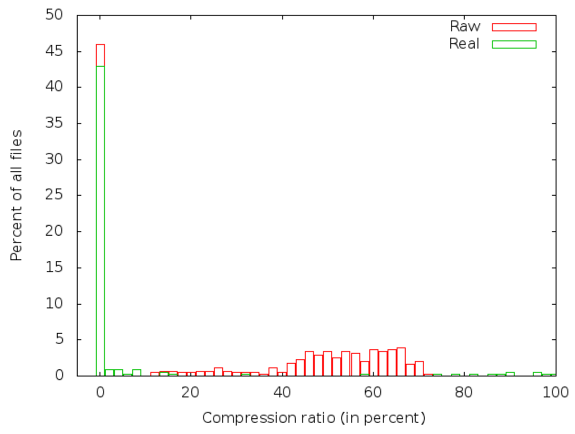
(b) Thumbnail JPEG Dataset B



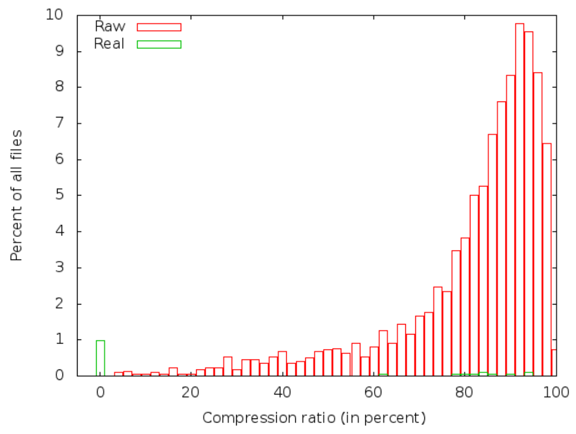
(c) Preview JPEG Dataset A



(d) Preview JPEG Dataset B



(e) PNG



(f) Native JPEG

Figure 5.6: Raw and real compression rates. All histograms are trimmed at 100%.

sent in their entirety to the server.

The choice of the threshold may appear somewhat arbitrary, and any threshold up to 100% can be just as good. The reason is that the compression rate is unknown until after the differencing is done and anything smaller than the actual file is beneficial, however marginally. Therefore patches are generated for all files and this is reflected accordingly in the differencing time. They are applied before download only in the cases where the size is below the threshold. This provides ground for reducing the threshold in order to improve the download gain at the cost of minimal impairment of the upload and compression gains. Regardless, Figure 5.6 shows that the amount of such large patch sizes is very low and any threshold between 50% and 100% would yield practically the same results, justifying the choice of 80% as a sensible one.

The raw compression rates are very unevenly distributed though always with a peak at 1%, as seen on Figure 5.6. In some cases the patch is up to 30 times bigger than the original file. The real compression rate is estimated sticking to the 80% threshold.

bsdiff was again evaluated, considering its slightly better patch sizes and fast runtime for small files. However, even the smallest JPEG files' (thumbnails) size increases tens of times when decompressed. Then the gain in time is hardly noticeable. While the raw compression is about 16% better, after the trimming of the results to the desired threshold of 80%, the real compression rate is virtually the same as *xdelta*'s. The performance deteriorates with increasing the file size and again *xdelta* is confirmed as the best performing tool.

A significant difference between the two datasets is observed. The peaks at 100% for Dataset B are due to the fact that the images are very different at pixel level. Very common modification technique is changing the lighting throughout the image, which effectively results in changes in all pixel values. Local changes, on the other hand, can be differenced exceptionally well, as suggested by the peak of files compressed to less than 1% of the entire file.

File Type	Download	Upload (client)	Upload (server)
Thumbnail JPEG A	1.740	1.086	0.633
Thumbnail JPEG B	2.351	1.223	0.730
Preview JPEG A	1.191	0.747	0.636
Preview JPEG B	1.267	1.426	0.880
PNG	1.764	0.656	0.530

Table 5.9: Gain coefficients for the image files. For the download coefficient number of patch applications best case $N = 1$ is used. The efficiency for each file type is taken into consideration, altering the upload times with $c = 100\%$ and the download with coefficient of 1.000 for the files whose patches are above the threshold of 80%.

The gain coefficients (Table 5.9) for download time are very poor. Because the application of a single patch requires a decompression of the original image and then compressing the recovered one, its runtime becomes considerable. This results in more than 70% slower download in some cases. This may remain unnoticed by the clients, provided that the download times for files of such size is much less than a second. However, the gain for the upload is also unclear. The majority of the files are still sent in their original form because of the poor performance of the differencing on them. Still, the time needed for their processing must be considered.

There is a gain for the PNG images. Provided that this approach is still lossless, it may prove useful with this kind of files, exactly because no data is lost and the more than 90% faster upload in nearly half of the cases sums up to 35%-47% faster transfers on average. The slower download may not prove crucial as the time still remains a fraction of a second for files of this size and download speed as discussed.

The client actually experiences a drop in the performance in both uploading and downloading JPEG files in addition to the introduced data loss. This makes the method practically unusable for this setup. It could be beneficial to only perform the differencing on the server side with the sole purpose of reducing the disk space requirements for JPEG files but this would still slow down the access time (if using *sequential* patching) to a given version of a file and introduce data loss. Given the small size of the files, this approach will be not considered advantageous in any way. It is noticeable though, that the upload time improves with increasing of the file size and the efficiency. This could mean that the approach is suitable for large JPEG files.

On the contrary, the performance on native JPEG files is exceptionally bad. Even the compression rate for the raw differencing peaks at 90%, as shown on Figure 5.5 and Figure 5.6f. The resulting real compression is around 250%, and only 1% of all files produce patches with size less than 80% of the original file. This is why the gain coefficients for this kind of images are not calculated. This approach has no advantage for such files whatsoever.

With decompression and compression of JPEG files, data loss is inevitable. However, it is practically imperceivable and the recovered images are visually identical to the corresponding images used for the differencing. Furthermore, the image modification itself introduces the same loss, as the editing software also needs to decompress an image and then compress it again with the changes made. By itself, this fact means that this kind of quality corruption is widely accepted. It is true that using differencing the loss is doubled because for each modification the decompression/compression cycle is done twice, but while image quality may be very important for high resolution images, for thumbnails and previews it is of next to none priority.

The overall benefit of this method is questionable. While some of the files indeed show great improvement in the performance of the differencing on them, the overall result is only marginally better. Undoubtedly, the achieved relative patch sizes are much better, but the size of the decompressed data still makes them impractical. The runtime of the tool is not as satisfactory anymore, partially because of the added processing steps, and partially because the small file sizes allow for very fast direct transfer to and from the server. The saved disk space is negligibly small, also because the size of these files is much less compared to the Adobe documents.

5.4 Video Frame Differencing on Image Files

With the limited success of performing file differencing on the decompressed image data, the need of a fundamentally different approach is obvious. The binary differencing tools are often tuned for executable files taking into account the nature of the binary data and the specifics of these files. Although the binary data is practically meaningless when it comes to image files because of the used compression, we try to employ the same idea.

Video compression uses knowledge of the nature of the images that represent each frame of a video stream. It is effectively a form of differencing, though not of binary data, but rather on a much higher level. In this section we describe the test results of compressing two version files into frames of a video. The second frame, being encoded as its difference from the first one, serves as a patch file. The recovery includes another encoding of the original file and then extracting the second image using the patch.

However, this approach comes with its limitations. Video encoding is usually lossy. The lossless variants simply put images one after the other without any encoding step, thus resulting in no reduction of the file size. Lossy codecs can achieve excellent compression but often at the price of greatly corrupting the quality of the images. The compromise is to use a lossy encoding with the best possible quality. The result is therefore virtually lossless, with next no none visible difference in quality. Apart from the data loss for any type of images, this method is useless when the two files are with different resolutions. This makes it not applicable when the modifications include image resizing or cropping.

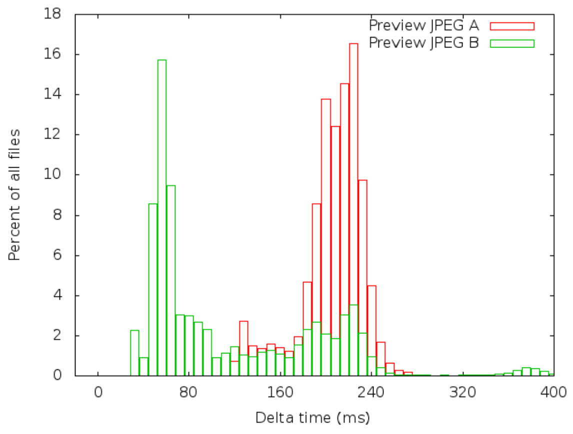
File Type	Delta time	Compression	Patch time	Efficiency
Thumbnail JPEG A	37.9 ms	18.4%	55.0 ms	93.9%
Thumbnail JPEG B	37.4 ms	25.8%	55.3 ms	66.3%
Preview JPEG A	209.8 ms	21.8%	331.6 ms	86.7%
Preview JPEG B	73.5 ms	18.4%	110.4 ms	73.8%
Native JPEG	722.0 ms	13.2%	897.6 ms	42.6%
PNG	70.9 ms	0.5%	97.4 ms	46.7%

Table 5.10: Performance of the video differencing on image files. The efficiency column indicates how many of the pairs of files have the same resolution and the video encoding is possible *and* achieve compression below the threshold of 80%. The rest will be considered with compression of 100% and the whole files should be sent to the server. Smaller means better.

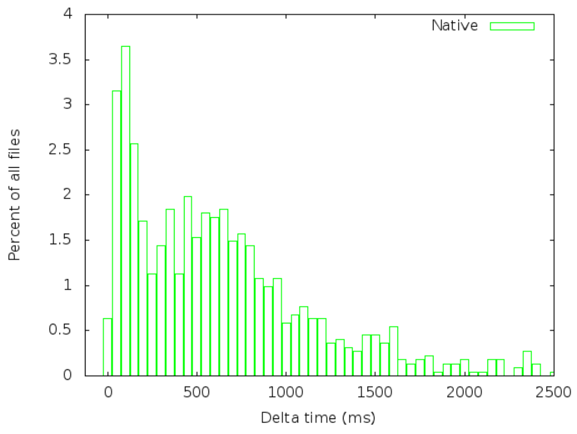
Table 5.10 presents a summary of the performance of this method. It is worth mentioning that the runtimes are much more consistent compared to the other methods and remain very similar, regardless of the difference between the files, and are directly proportional to the file sizes. Figure 5.7 clearly supports this claim. A comparison to Figure A.1c and A.1e suggests that after the lower bound of approximately 35 ms the runtime is linear with respect to file size. This also explains the fact that practically all (very small sized) thumbnail images show runtime between 1 and 4 ms above the bound.

The compression rates are very similar to the ones for differencing of decompressed images but this time the efficiency is much better. Less than 10% of all patches are above the threshold. Still, the efficiency is greatly reduced by the fact that many versions have different resolutions and are therefore not eligible for the video encoding. As seen on Figure 5.8, the compression rate of the rest of the files is impressive and the biggest drawback of the method so far is that it is not applicable to arbitrary files.

The patch application time behaves very similarly to the generation time, only its lower

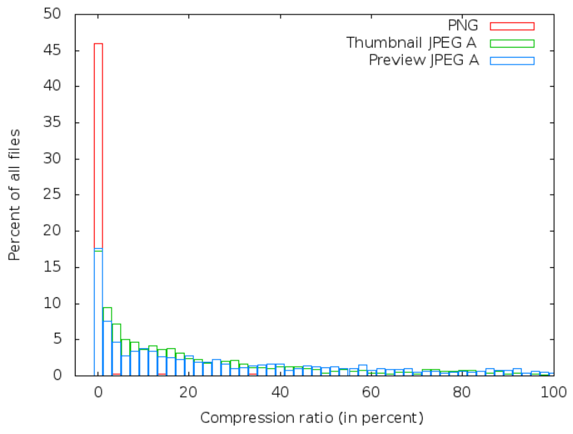


(a) Preview JPEG

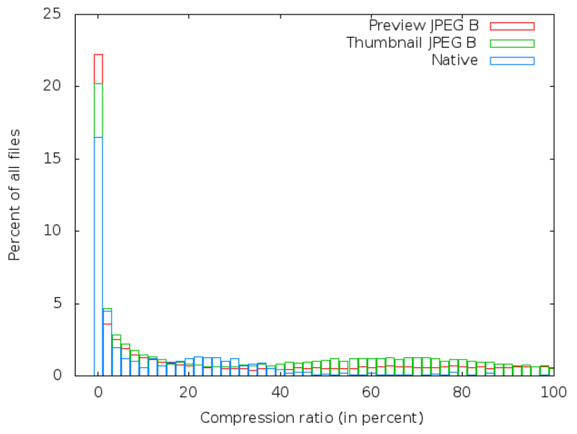


(b) Native JPEG

Figure 5.7: Patch generation (delta) times for preview and native JPEG files using video differencing.



(a) Dataset A



(b) Dataset B

Figure 5.8: Compression rates for both datasets using the video differencing approach.

bound is shifted to 54 ms. It actually takes longer, unlike the differencing tools where the application is much faster than the generation. In order to recover an image, the original file (the first frame) is encoded again, the patch (the second frame) is appended to it and then the extraction takes place. In contrast to the patch generation where only encoding of the two files is done, this requires two separate executions of the video processing tool.

The download slow-down for small files can again remain unnoticed because of their total transfer time of a few milliseconds. Also, if *reverse sequential* patching is used instead (assuming the performance is similar to the one of the *sequential* patching, as seen in the previous sections) the download gain becomes 1 for the cost of damaging the upload gain and more processing on the server. In this case though, the upload coefficients become close to and even larger than 1, meaning the upload will be actually slower. Assuming that download speed is often 10 times

File Type	Download	Upload (client)	Upload (server)
Thumbnail JPEG A	6.869	0.665	0.234
Thumbnail JPEG B	12.457	1.677	0.508
Preview JPEG A	1.936	0.390	0.322
Preview JPEG B	2.643	0.643	0.398
Native JPEG	1.368	0.716	0.630
PNG	1.215	0.562	0.535

Table 5.11

faster than upload, the total time is still less with the current, *sequential* approach.

Overall the differencing compression is much better than any of the other methods. For very small files the runtime becomes comparable to the time required to transfer the whole file to the server and the approach can hardly be considered beneficial. Still, with increasing the file size, the advantages become more obvious and the compression for the biggest investigated native JPEG images is excellent compared to the one showed by other methods and file types. However, the efficiency is greatly reduced by versions with different resolution, which in turn makes the upload and compression gains much worse. Even the excellent compression rate of PNG images is shadowed by the fact that less than half of the files are eligible for such differencing. Provided that the typical image editing does not include resizing or cropping, the benefits can become substantial.

Although the gains in data transfer time are questionable, there is definitely advantage when it comes to saved disk space - between 37% and 76%, depending on the file type. In case the amount of this kind of files is very large, the video encoding differencing approach can prove useful. However, all image files suffer from another side effect from this approach - data loss. Video encoding introduces more quality corruption than differencing of decompressed image data, which can be undesirable for high quality and/or PNG images. The loss, however minimal, is often visually perceivable. It can be reduced or increased trading off the compression rate by using different video codecs and varying their settings. This way of controlling it can yield satisfactory results depending on the purpose of the images.

Again the benefits of this method are not obvious. While the achieved compression surpasses all other investigated methods, its efficiency is still reduced by the illegibility of many images and the relative slow runtimes. Also, quality corruption is more severe and introduced regardless of the image format (even for the lossless ones). This approach can still be beneficial in many cases, especially when large number of big images are managed, and their quality is not of great priority.

Chapter 6

Conclusion

6.1 Summary

The variety of media files makes it very difficult to present a single solution performing well for all types and sizes. Even a dataset consisting of files of the same type and with the same purpose (preview, thumbnail, etc.) exhibits great variance in file size and performance of the tools. In this work four binary differencing tools were investigated, along with four different patching methods. In addition to the direct differencing for image files, two other methods were evaluated - with the addition of pre- and post-processing steps dealing with the compression of the images, and video encoding approach.

For Adobe Documents (and most likely other image editing software files) the binary differencing tool *xdelta* shows the best run-time for the compression it achieves. *rdiff* is faster, but its patches are bigger. *bsdiff* shows better compression, but its non-linear runtime makes it unusable for files of this size. The benefits of *xdelta* are considerable - decrease of the file size with more than 96% for Adobe InDesign files, causing up to 30 times faster file transfer. Adobe Photoshop documents also show substantial gain despite their much larger size.

The two sequential patching methods show very similar performance. The trade-off between them is in the version recovery time, with one being faster for new versions, and the other - for old ones. Also, the *reverse* method requires processing to be done on the server. While the *root* method can be more beneficial in some cases depending on the dataset, it also requires differencing to take place on the server and is still expected to produce larger patches.

The direct application of binary differencing tools shows very poor performance on image files such as JPEG and PNG images. This is due to their compression, causing even small changes to propagate throughout the whole file's binary representation. Decompression prior to generating a patch can yield better results in some cases, but due to the very large size of the raw pixel data, more often than not this method is also not beneficial, especially for JPEG compressed images. Data loss introduced with using this method is none for lossless images and next to none for lossy ones.

Employing video compression approach yields much better compression rates but its slower runtime makes it practically unusable for small files. Bigger files show better gains but their quality deteriorates.

Essentially all of the investigated approaches can be beneficial to some extent, especially regarding saved disk space. Even the worst performing tools lead to reduction of the storage requirements. With the constantly increasing size of image files and image modification software documents and growth of file stores, this can lead to saving substantial disk space. In many cases the file transfer time is also reduced and while for small files this does not make any noticeable difference especially with the improvement of the connection speed, large files are processed considerably faster. Again, increasing the file size makes the benefits even better.

6.2 Limitations

Still, most of the results are very dataset-specific. Not only do they depend on the expected number of versions per file, size and network speed, but also on the type of changes typically made to the files and their severity. A detailed knowledge about the specifics of the file store at hand is needed to fully exploit the benefits of file differencing and find the best suitable approach, tool and patching method.

For example, if changes are often undone, the *root* patching will perform better than the *sequential* ones. If the last version of a file is usually requested, the *reverse* methods will prove more suitable in case expensive processing can be done on the server. The *sequential* patching requires multiple patch applications to be performed on the server prior to transferring a requested version, therefore increasing the download time.

Decompression of images causes their size to increase multiple times. Many of the images are extremely well compressed at more than 98%, but this holds only when small local changes are made and the produced patches become even larger than the files themselves when common modifications that affect most or all pixels are made (such as changes in lighting).

The video encoding approach introduces considerable quality corruption to the recovered images, which can be controlled but not eliminated. Furthermore, this approach is only usable for files with the same resolution and not when the changes made include cropping or resizing of an image, which is very common in practice.

This indicates that the type of changes that are typically made to the images has a great impact on the performance of the differencing. The size and purpose of the files are also important factors. Thumbnail and preview images may allow for loss of quality but their small size often causes the differencing to still be inefficient and simply transferring the whole files to be faster. Large images can be well compressed and transferred fast if data loss is acceptable.

6.3 Future Work

Big files like Photoshop documents allow for improvement of the performance of differencing tools on them, especially because of their open format. Further format-specific investigation can improve the compression rates of *xdelta*, essentially greatly reducing the required storage space and transfer time. When their data is (RLE) compressed, decompressing it prior to differencing may well lead to reduction of the patch size.

Combining different approaches or patching methods can improve the results by individually selecting the method that performs the best for each image. This requires knowledge of the made changes that eventually helps in the choice of the most suitable setup. As already discussed, the performance of the *sequential*, *root*, and *reverse* methods depends on the type of changes and requests. Similarly, decompressed files are best differenced when local changes are made, and video encoding works best with images of the same resolutions. Furthermore, the *sequential* patching methods can be altered by saving the entire file every few versions, so the number of patch applications upon request is bounded. Further investigation of the conditions that produce the best results is needed.

It may be possible that the video differencing runtime can be reduced by using a less generic approach and tuning the video encoding tool for specific image types, making some of its processing obsolete. Also, handling of images with different resolutions can be implemented when the nature of the changes is known (cropping, resizing, composition). The two images can be scaled to the same size accordingly and the differencing is performed with information about the original size and the scaling stored in the patch. After patching, the image and its dimensions can be restored. This can immensely improve the efficiency of the approach and make it useful for all types of image files.

Appendix A

Test Data

Dataset A consists of 3,221 Adobe InDesign files, of which 683 original and 2,538 versions of them. Each original file has up to 5 versions, and in addition, each of the InDesign documents has a corresponding exported preview JPEG image with resolution 819x1190 pixels of its contents (a magazine page abundant of text and images), as well as a thumbnail JPEG image with resolution 100x145 pixels. Additional 870 PNG files (431 original ones) in various resolutions are available.

Dataset B consists of 24,070 InDesign files - 4,604 original and the rest version files. Again JPEG preview and thumbnail files (57,945 files of each type) are present. From them 18478 are original while the rest is the modified files. In contrast to Dataset A, these files have various resolutions. In addition, 17,165 high-quality native JPEG files are present (8,243 originals and 8,922 versions). Also available are 10628 Adobe Photoshop files, of which 3475 original and 7153 version files. Each original file in this dataset has up to 11 versions.

File Type	Original		Version		Total	
	A	B	A	B	A	B
Adobe InDesign Document	683	4604	2538	19466	3221	24070
Adobe Photoshop Document	-	3475	-	7153	-	10628
Native JPEG Image	-	8243	-	8921	-	17165
Preview JPEG Image	683	18478	2538	39467	3221	57945
Thumbnail JPEG Image	683	18478	2538	39467	3221	57945
PNG Image	431	-	439	-	870	-

Table A.1: The number of each type of files in both datasets.

The total number of files and the number of version files are shown separately on Table A.1 because for each patching sequence one file is kept (either the original or the last version). Still, the number of version files remains the same, regardless of the choice of an original file serving as a starting point for the differencing. For the average size calculation on Table A.2, the files are not distinguished because the left-out files differ in the normal and the *reverse* patch generation

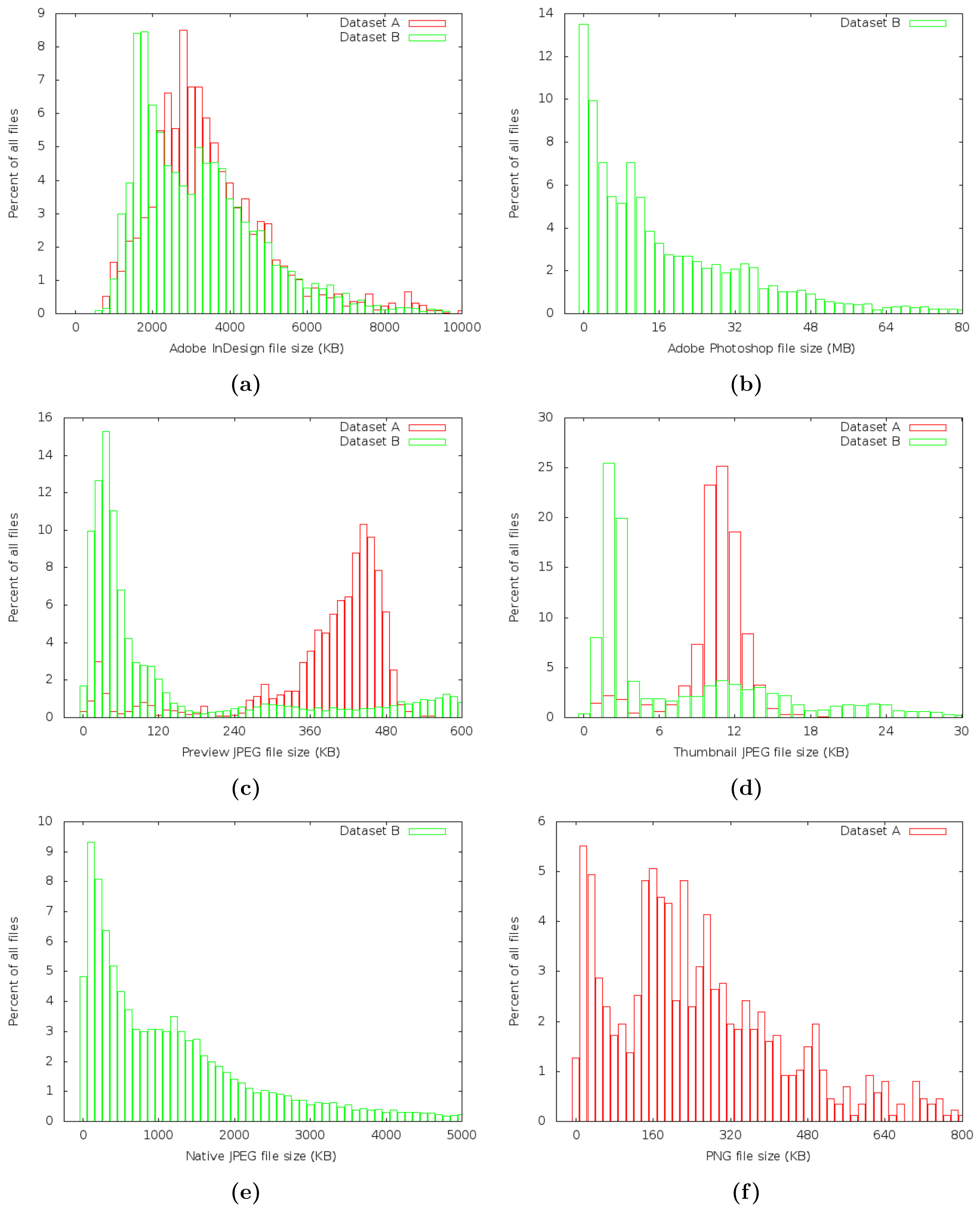


Figure A.1: File size distributions for all investigated formats.

approaches but the results are virtually the same.

Figure A.1 shows the distributions of the size of all formats. From these distributions, the expected size of the files of each type can be evaluated. The results are shown on Table A.3. "Expected" file size essentially means that a file from a given type has size in the range $\mu \pm \sigma$ with confidence level of 90%. μ is close to the most often encountered size (mode) in the dataset.

File Type	Minimum		Maximum		Average	
	A	B	A	B	A	B
Adobe InDesign Document	877 KB	68 KB	19.2 MB	19.8 MB	3.7 MB	3.4 MB
Adobe Photoshop Document	-	48 KB	-	282 MB	-	21.3 MB
Native JPEG Image	-	20 KB	-	48.6 MB	-	1.6 MB
Preview JPEG Image	12 KB	4 KB	548 KB	1.3 MB	384 KB	163 KB
Thumbnail JPEG Image	3 KB	1 KB	19 KB	36 KB	11 KB	8 KB
PNG Image	10 KB	-	5.8 MB	-	334 KB	-

Table A.2: The range and average size of each type of files in both datasets.

File Type	μ	σ
Adobe InDesign Document Dataset A	3.4 MB	2.3 MB
Adobe InDesign Document Dataset B	3.0 MB	1.6 MB
Adobe Photoshop Document	15.1 MB	20.0 MB
Native JPEG Image	1.3 MB	1.6 MB
Preview JPEG Image Dataset A	384 KB	132 KB
Preview JPEG Image Dataset B	62 KB	68 KB
Thumbnail JPEG Image Dataset A	11 KB	4 KB
Thumbnail JPEG Image Dataset B	4 KB	3 KB
PNG Image	265 KB	270 KB

Table A.3: Expected file size for different file types.

Bibliography

[1] Adobe Systems Incorporated. Adobe photoshop file formats specification. Technical report, 2012.

[2] B. Baker, U. Manber, and R. Muth. Compressing differences of executable code. In *ACM-SIGPLAN Workshop on Compiler Support for System Software (WCSS)*, pages 1–10, 1999.

[3] A. Black and C. Burris Jr. A compact representation for file versions: A preliminary report. In *Data Engineering, 1989. Proceedings. Fifth International Conference on*, pages 321–329. IEEE, 1989.

[4] R. Burns and D. Long. A linear time, constant space differencing algorithm. In *Performance, Computing, and Communications Conference, 1997. IPCCC 1997., IEEE International*, pages 429–436. IEEE, 1997.

[5] L. Deutsch. DEFLATE compressed data format specification version 1.3. *RFC1951*, 1996.

[6] D. Duce and T. Boutell. Portable network graphics (PNG) specification. *Information technology ISO/IEC*, 15948:2003, 2003.

[7] A. Hemel, K. Kalleberg, R. Vermaas, and E. Dolstra. Finding software license violations through binary code clone detection. *van Deursen et al.*, pages 63–72, 2011.

[8] D. Korn and K. Vo. Engineering a differencing and compression data format. In *Proceedings of the Usenix Annual Technical Conference*, pages 219–228, 2002.

[9] D. Korn and K. Vo. The VCDIFF generic differencing and compression data format. *Work in Progress*, 2002.

[10] H. Makholm. Specification of the XCF file format. Technical report, 2006.

[11] C. Percival. Naïve differences of executable code. *http://www.daemonology.net/bsdiff*, 2003.

[12] C. Percival. *Matching with mismatches and assorted applications.* PhD thesis, University of Oxford, 2006.

[13] V. Roussev. Hashing and data fingerprinting in digital forensics. *Security & Privacy, IEEE*, 7(2):49–55, 2009.

[14] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 117–128. ACM, 2009.

[15] D. Salomon. *Data compression: the complete reference*. Springer-Verlag New York Incorporated, 2004.

[16] N. Samteladze and K. Christensen. DELTA: delta encoding for less traffic for apps. In *Local Computer Networks (LCN), 2012 IEEE 37th Conference on*, pages 212–215. IEEE, 2012.

[17] The Chromium Projects. Software updates: Courgette, 2012.

[18] D. Trendafilov, N. Memon, T. Suel, et al. zdelta: An efficient delta compression tool. 2002.

[19] A. Tridgell and P. Mackerras. The rsync algorithm, 1996.

[20] A. van Hoff and J. Payne. Generic diff format specification. Technical report, Technical Report NOTE-GDIFF, World Wide Web Consortium, 1997.

[21] J. Young, K. Foster, S. Garfinkel, and K. Fairbanks. Distinct sector hashes for target file detection. *Computer*, pages 28–35, 2012.