Entwicklung eines Container-Verfahrens für digitale Bildwasserzeichen

Design of a Watermark Container for digital Image Watermarking

Master-Thesis von Jonathan Römer

13.12.2011

TECHNISCHE UNIVERSITÄT DARMSTADT

Fraunhofer SIT

CASED

Entwicklung eines Container-Verfahrens für digitale Bildwasserzeichen
Design of a Watermark Container for digital Image Watermarking

Vorgelegte Master-Thesis von Jonathan Römer

1. Gutachten: Prof. Dr. Michael Waidner
2. Gutachten: Dr. Huajian Liu
3. Gutachten: Dr. Martin Steinebach

Tag der Einreichung:

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 13.12.2011

(Jonathan Römer)

Zusammenfassung

Digitale Wasserzeichen ermöglichen das versteckte Einbetten von Informationen in digitale Werke. Die robuste und gleichzeitig transparente Einbettung von Wasserzeichen ist allerdings sehr komplex und benötigt daher oft lange Rechenzeiten. Der Wasserzeichen-Container liefert eine Lösung für dieses Problem. Das Konzept des Containers ist es, den Markierungsvorgang in zwei Phasen aufzuteilen: In der ersten Phase, dem Containering, wird einmalig ein Wasserzeichen-Container erzeugt, der bereits alle Permutationen von markierbaren Versionen eines Originalwerks enthält. In der zweiten Phase, dem Shuffling, können dann sehr schnell beliebige, individuell markierte Versionen des Werkes aus der zuvor erzeugten Containerdatei generiert werden. So kann die Einbettungsdauer durch die Anwendung des Container-Verfahrens enorm beschleunigt werden. Derartige Container-Verfahren werden bereits für Video- und Audiowasserzeichen eingesetzt. In dieser Arbeit wird ein Container-Verfahren für Bildwasserzeichen vorgestellt. Die Grundlage dafür bildet das *ImageMark* Bildwasserzeichen, welches die Einbettung einer Wasserzeichennachricht mittels der Manipulation von Fourierkoeffizienten des Helligkeitsanteils eines Bildes vornimmt.

Schlüsselwörter:
Digitale Bildwasserzeichen, Wasserzeichen Container, Effiziente Wasserzeicheneinbettung

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Die private Nutzung von Computern und des Internets steigt stetig an. Im ersten Quartal des Jahres 2010 besaßen 78% aller Personen in Deutschland ab einem Alter von zehn Jahren einen Computer und 75% einen Internetanschluss zur privaten Nutzung[1]. Das Verwenden digitaler Medien wird auf Grund dieser Entwicklung immer beliebter und verdrängt mehr und mehr die Verwendung analoger Medien. Durch ständig größer und günstiger werdende Bandbreiten und Speicherkapazitäten wird zudem das Austauschen von sehr großen Datenmengen und somit sowohl das legale als auch das illegale Austauschen von digitalen Medien erleichtert. Kopien von digitalen Medien, seien es Bilder, Texte, Video- oder Audiodokumente, können ohne Qualitätsverlust in großen Mengen hergestellt und verbreitet werden. Abgesehen von Strom- und Speicherkosten ist dabei kein weiterer finanzieller Aufwand notwendig. Analoge Kopien von vergleichbarer Qualität können dagegen nur in begrenzten Mengen und mit einem entsprechenden Kostenaufwand angefertigt werden (vgl. [4]).

Mit der zunehmenden Verwendung digitaler Medien wird seit einigen Jahren an der Entwicklung von digitalen Wasserzeichen geforscht. Sowohl für Audio-, als auch für Bild- und Videodaten stehen bereits einige Verfahren zur Verfügung (vgl. [2], [13]). Das digitale Wasserzeichen ist ein Mechanismus aus der Steganographie und kann zur versteckten Einbettung von Zusatzinformationen in ein digitales Werk verwendet werden. Die eingebetteten Informationen können aus einem markierten Werk meist mittels eines privaten Schlüssels, der bei der Einbettung verwendet wurde, wieder ausgelesen werden. Die wichtigsten Eigenschaften eines digitalen Wasserzeichens sind folgende (vgl. [4], [14]):

- Robustheit
- Nicht-Detektierbarkeit
- Nicht-Wahrnehmbarkeit
- Sicherheit
- Komplexität
- Kapazität

Je nach Art und Anwendung eines Verfahrens liegen die Prioritäten auf einigen dieser Eigenschaften. Es ist nicht möglich alle Eigenschaften gleichzeitig zu optimieren. Will man beispielsweise eine hohe Kapazität erreichen, so wird dies zwangsläufig die Nicht-Wahrnehmbarkeit

[1] Quelle: www.destatis.de, Zugriff: 09.12.2011

oder die Robustheit verringern. Die genannten Eigenschaften konkurrieren also miteinander. Laut Dittmann lassen sich digitale Wasserzeichen in *fragile* und *robuste Wasserzeichen* unterteilen (vgl. [4]). Fragile Wasserzeichen sind nicht robust gegen Veränderungen am Datenmaterial, sodass Veränderungen an diesem das Wasserzeichen beschädigen können und die eingebettete Information nach einer Veränderung nicht mehr ausgelesen werden kann. Ein robustes Wasserzeichen kann dagegen auch nach inhaltserhaltenden und inhaltsverändernden Manipulationen ausgelesen werden. Es kann nur entfernt werden, indem das Dokument selbst zerstört oder dessen Qualität sehr stark beeinträchtigt wird. Der Grad der Robustheit eines Wasserzeichenverfahrens wird an Hand der Angriffe bewertet, die ein Wasserzeichenverfahren übersteht. Wenn aus einem markierten Werk auch nach einem durchgeführten Angriff noch das eingebettete Wasserzeichen ausgelesen werden kann, so ist das Wasserzeichenverfahren robust gegen diesen Angriff. Zur Überprüfung der Robustheit von Bildwasserzeichen-Verfahren beinhalten diese Angriffe meist verlustbehaftete Kompression (JPEG), Weichzeichen-Filterung, Entfernen von Teilen des Bildes, Hinzufügen von Bildrauschen sowie affine Transformationen (Drehung, Skalierung, Translation) und beliebige Kombinationen aus all diesen Teilangriffen.

Die Anwendungsgebiete von digitalen Wasserzeichen sind sehr vielfältig und beinhalten Fingerprinting, Urheberrechts-Schutz, Kopierschutz, Integritätsschutz, Authentizitätsschutz und Broadcast Monitoring (vgl. [2], [14]).

Die Verwendung von digitalen Wasserzeichen hat im Vergleich zu kryptographischen Mechanismen einen klaren Vorteil. Mit Hilfe von kryptographischen Methoden können digitale Signaturen an das Datenmaterial angefügt werden, um beispielsweise die Informationen über den Urheber einer Datei an diese zu binden. Hashfunktionen und Prüfsummen können verwendet werden, um die Authentizität und die Integrität von Daten zu beweisen. Als Eingangssignal für kryptographische Methoden dient dabei immer die bitweise Darstellung einer Datei. Daher führen auch inhaltserhaltende Veränderungen an dieser oder eine Kompression des Datenmaterials zu einer Veränderung der Syntax des Datenmaterials ohne die Semantik der Datei zu verändern. Zudem ist es ohne großen Aufwand möglich eine an das Datenmaterial angehängte Signatur zu entfernen oder auszutauschen ohne das Datenmaterial zu beschädigen. Auch bei der Vewendung von kryptographischer Verschlüsselung liegt das digitale Medium spätestens beim Betrachten oder Hören durch den Benutzer in einer unverschlüsselten und somit ungeschützten und kopierbaren Form vor. Eine robuste Einbettung von Informationen in digitale Medien kann aus diesen Gründen mit Methoden aus der Kryptographie nicht gewährleistet werden (vgl. [1]).

Diese Arbeit beschäftigt sich mit robusten digitalen Bildwasserzeichen. Die robuste und gleichzeitig nicht wahrnehmbare Einbettung von Informationen in Bilddaten ist im Allgemeinen mit hoher Komplexität, also rechenaufwändigen Operationen, verbunden. In einem einfachen Anwendungsszenario, in dem ein Benutzer nur einzelne Bilder markiert, stellt es kein Problem dar, wenn der Markierungsprozess mehrere Sekunden dauert. Möchte man allerdings eine große

Anzahl von Bildern nacheinander oder gleichzeitig markieren, so ist die Geschwindigkeit der Wasserzeichenmarkierung ausschlaggebend für die Benutzbarkeit des Verfahrens.

In dieser Arbeit wird ein Verfahren beschrieben, das die Wasserzeichen-Markierung von digitalen Bildern beschleunigen kann.

In Kapitel 2 wird zunächst der aktuelle Stand der Forschung an Bildwasserzeichen untersucht. Im darauf folgenden Abschnitt wird die Motivation beschrieben, aus der diese Arbeit entstanden ist. In Kapitel 4 folgen die Beschreibungen des Wasserzeichen-Container Konzeptes und des ImageMark Bildwasserzeichens, welches die Grundlage dieser Arbeit ist. Darauf aufbauend werden in Kapitel 5 zwei neue Konzepte eines Container-Verfahrens für das ImageMark Bildwasserzeichen vorgestellt. Kapitel 6 enthält eine Beschreibung der technischen Umsetzung. Im folgenden Abschnitt werden die entwickelten Container-Verfahren evaluiert. Die letzten beiden Kapitel enthalten ein Fazit und einen Ausblick auf mögliche weiterführende Arbeiten in dem behandelten Themengebiet.

2 Aktueller Stand der Forschung an Bildwasserzeichen-Verfahren

In diesem Abschnitt wird der momentane Stand der Forschung auf dem Gebiet der digitalen Bildwasserzeichen dargestellt. Die existierenden Wasserzeichenverfahren lassen sich in robuste und fragile sowie in nicht blinde und blinde Verfahren gruppieren. Ein robust eingebettetes Wasserzeichen kann nicht entfernt werden ohne das markierte Bild zu zerstören oder zumindest dessen Qualität so stark zu verringern, dass es nicht mehr brauchbar ist. Fragile Wasserzeichen werden dagegen schon durch kleinste Veränderungen am Bildmaterial zerstört. Bei einem blinden Wasserzeichenverfahren wird bei der Detektion weder das Originalbild noch eine andere komplexe Datenstruktur benötigt, die Informationen über das Originalbild enthält. Nicht blinde Verfahren benötigen das Originalbild oder komplexe Datenstrukturen mit Informationen über das Originalbild bei der Detektion.

Des Weiteren unterscheiden sich die existierenden Verfahren in der Domäne, in der die eigentliche Wasserzeicheneinbettung vorgenommen wird. Bei einigen Verfahren wird das Wasserzeichen durch das Auftragen eines Wasserzeichen-Musters auf das gesamte Bild oder auf einzelne Blöcke des Bildes eingebettet. Dieses Vorgehen hat den Nachteil, dass die Nachrichten-Bits der eingebetteten Wasserzeichen-Nachricht an bestimmte Regionen des Bildes gebunden sind. Entfernt man Teile eines markierten Bildes, so gehen auch zwangsläufig Teile der eingebetteten Wasserzeichennachricht verloren. Die meisten Pixel-basierten Wasserzeichen sind des Weiteren nicht gegen Gauß-Filter robust, da diese die eingebetteten Muster zerstören (vgl. [14]).

Bei einer anderen Gruppe von Verfahren wird zunächst eine Transformation des Originalbildes in den Frequenzraum vorgenommen. In der Frequenzdomäne werden dann Manipulationen an den Bilddaten vorgenommen, um das Wasserzeichen einzubetten. Dazu kann auch hier beispielsweise ein Wasserzeichenmuster aufgetragen werden. Anschließend wird das markierte „Frequenzbild" wieder mittels der inversen Transformation in die Pixeldomäne übertragen. Die am Häufigsten verwendeten Frequenz-Transformationen sind dabei die diskrete Kosinus-, Fourier- und Wavelet-Transformaion (DCT, DFT, DWT). Der große Vorteil der Wasserzeicheneinbettung im Frequenzraum ist, dass sich die Wasserzeichenenergie eines einzelnen Nachrichten-Bits nicht nur auf eine bestimmte Region des Bildes beschränkt, sondern über das gesamte Bild verteilt. Diese Frequenzraum-Verfahren sind somit robust gegen das Entfernen und Manipulieren von Teilen des markierten Bildes, sowie gegen Translationen, da die angewandten Frequenzraum-Transformationen im Allgemeinen invariant gegen Translationen sind.

Ein häufig verwendeter Mechanismus bei robusten Bildwasserzeichen ist die Verwendung eines sogenannten *Templates* (vgl. [16]). Das Konzept des Templates ist es, ein möglichst einfaches Muster in das Bild einzubetten, welches auch nach affinen Transformationen (Rotationen, Trans-

lationen und Skalierungen) entdeckt werden kann. Das Template wird dann zusätzlich zu dem verwendeten Wasserzeichen bei der Einbettung in das zu markierende Bild eingebracht. Bei der Wasserzeichendetektion wird dann zunächst in dem Bild nach dem Template-Muster gesucht. Wird dieses gefunden, so können die an dem Bild vorgenommenen Transformationen an Hand des Unterschieds zwischen dem detektierten und dem erwarteten Template-Muster rückgängig gemacht werden. Erst dann wird an dem rekonstruierten Bild die eigentlich Wasserzeichendetektion vorgenommen.

In den beiden folgenden Unterkapiteln werden zwei Bildwasserzeichen beschrieben. Dabei wird sowohl auf die Funktionsweise als auch auf die Performanz eingegangen. Die beiden Wasserzeichenverfahren wurden an dieser Stelle gewählt, da es sich bei beiden Verfahren, wie auch bei dem ImageMark-Bildwasserzeichen, welches die Grundlage dieser Arbeit bildet, um blinde und robuste Verfahren handelt. Die Wasserzeicheneinbettung wird bei beiden Verfahren, wie auch bei ImageMark, in der Frequenzdomäne vorgenommen. Die Art der Einbettung im Frequenzraum unterscheidet sich allerdings deutlich vom Vorgehen bei ImageMark.

2.1 Robustes DWT/DFT Bildwasserzeichen (Kang et al.)

Kang et al. beschreiben in [12] ein blindes Bildwasserzeichen, das gegen JPEG-Kompression und affine Transformationen robust ist. Die Wasserzeicheneinbettung bei diesem Verfahren setzt sich aus zwei Teilen zusammen. Eine Synchronisationssequenz und die Wasserzeichennachricht werden in einem niedrigen Frequenzband der Wavelet-Domäne eingebettet. Um das Wasserzeichen gegen affine Transformationen zu schützen, wird zusätzlich ein Template nach dem in [16] beschriebenen Verfahren in der Fourierdomäne eingebracht. Das Template wird in die mittleren Frequenzen eingebettet, um Überlagerungen mit dem in der Wavelet-Domäne eingebetteten Wasserzeichen zu vermeiden. Das Wasserzeichenverfahren verfügt über eine Einbettungskapazität von 60 Bit für Graustufenbilder mit einer Auflösung von 512×512 und 8 Bit pro Pixel. Allerdings ist das Wasserzeichenverfahren nicht auf diese Größen festgelegt.

2.1.1 Einbettung und Detektion in der Wavelet Domäne

Die einzubettende Wasserzeichennachricht m der Länge $L = 60$ wird zunächst mittels der Fehlerkorrektur-Methode BCH auf m_c mit Länge $L_c = 72$ abgebildet. Dann werden die Bits m_{ci} von $\{0, 1\}$ auf $\{-1, 1\}$ gemappt und eine DSSS Kodierung (*direct-sequence spread spectrum*) vorgenommen. Für diese Kodierung wird ein bipolarer Spreizcode p ($p_i \in \{-1, 1\}$) der Länge $N_1 = 11$ verwendet, der aus dem privaten Schlüssel k über eine Pseudozufallsfunktion generiert wird. So erhält man eine Matrix W ($W_{ij} \in \{-1, 1\}$) mit den Dimensionen $N_1 \times L_c$, welche die kodierte Wasserzeichennachricht m enthält.

Durch die Anwendung einer vierstufigen DWT werden aus dem Eingangsbild $f(x,y)$ zwölf hochfrequente Bänder (LH_i, HL_i und HH_i, $i \in \{1,2,3,4\}$) und ein niederfrequentes Band (LL_4) generiert. Die hochfrequenten Bänder bleiben unverändert. Das Wasserzeichen wird lediglich in das niederfrequente Band LL_4 eingebracht.

Dann wird eine Matrix M generiert, die die selben Dimensionen wie LL_4 besitzt (32×32). Aus dem geheimen Schlüssel k wird über eine Pseudozufallsfunktion eine Synchronisationssequenz T der Länge $L_T = 63$ mit $T_i \in \{-1,1\}$ generiert. Diese Sequenz wird in Zeile 16 und Spalte 16 der Matrix M kopiert. $M(T)$ wird dann einem zweidimensionalen De-Interleaving (vgl. [5] und [20]) unterzogen, sodass man eine Matrix $M_D(T^D)$ erhält. Die Matrix W, die die kodierte Wasserzeichennachricht enthält, wird in die noch freien Positionen der Matrix kopiert. Dann wird diese Matrix $M_D(T^D, W)$ einem zweidimensionalen Interleaving unterzogen, sodass sich die Bits der Synchronisationssequenz wieder an ihrer ursprünglichen Position befinden, die Bits aus W aber verschachtelt vorliegen. $M(T, W^I)$ wird dann spaltenweise in einen eindimensionalen Array X kopiert. Die Koeffizienten des niederfrequenten Bandes LL_4 werden ebenfalls spaltenweise in einen eindimensionalen Array C kopiert. Der Elemente des markierten Array C' werden dann nach folgender Abbildungsregel aus C und X berechnet:

$$C'(i) = \begin{cases} C(i) - (C(i) mod \alpha) + \frac{3}{4}\alpha & \text{,falls } X(i) = +1 \text{ und } (C(i) mod \alpha) \geq \frac{1}{4}\alpha \\ (C(i) - \frac{1}{4}\alpha) - ((C(i) - \frac{1}{4}\alpha) mod \alpha) + \frac{3}{4}\alpha & \text{,falls } X(i) = +1 \text{ und } (C(i) mod \alpha) < \frac{1}{4}\alpha \\ C(i) - (C(i) mod \alpha) + \frac{1}{4}\alpha & \text{,falls } X(i) = -1 \text{ und } (C(i) mod \alpha) \leq \frac{3}{4}\alpha \\ (C(i) + \frac{1}{2}\alpha) - ((C(i) - \frac{1}{2}\alpha) mod \alpha) + \frac{1}{4}\alpha & \text{,falls } X(i) = -1 \text{ und } (C(i) mod \alpha) > \frac{3}{4}\alpha \\ C(i) & \text{,sonst} \end{cases}$$

Somit liegen die Differenzwerte aus $C(i)$ und $C'(i)$ in $\left[-\frac{1}{2}\alpha, \frac{1}{2}\alpha\right]$. Des Weiteren gilt:

$$C'(i) mod \alpha = \begin{cases} \frac{3}{4}\alpha & \text{,falls } X(i) = +1 \\ \frac{1}{4}\alpha & \text{,falls } X(i) = -1 \end{cases}$$

Das markierte niederfrequente Band LL_4' wird dann zusammen mit den anderen zwölf hochfrequenten Bändern einer inversen Wavelet-Transformation unterzogen. So erhält man ein markiertes Bild $f'(x,y)$, welches die Wasserzeichennachricht und die Synchronisationssequenz enthält. Abbildung 2.1 enthält eine Gesamtübersicht des Einbettungsverfahrens.

Bei der Detektion wird lediglich der private Schlüssel k benötigt, um daraus analog zur Einbettung den Spreizcode p_i, sowie die Synchronisationssequenz T zu generieren. Es wird zunächst

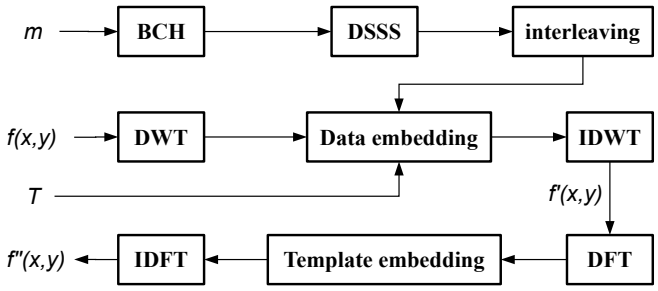

Abbildung 2.1.: Einbettung des Wasserzeichens und des Templates in der DWT- und DFT-Domäne (Quelle: [12])

das niederfrequente Band LL_4^* und daraus der eindimensionale Array C^* analog zum Vorgehen bei der Einbettung erzeugt. An Hand eines Koeffizienten $C^*(i)$ kann das eingebettete Bit $X^*(i)$ wie folgt berechnet werden:

$$X^*(i) = \begin{cases} +1 & \text{,falls} \quad C^*(i) mod \alpha > \alpha/2 \\ -1 & \text{,sonst} \end{cases}$$

Aus X^* kann dann die Synchronisationssequenz T^* und das kodierte Wasserzeichen W^* ausgelesen werden. Wenn T^* mit der erwarteten Synchronisationssequenz T übereinstimmt, so wird angenommen, dass W^* tatsächlich eingebettet wurde und somit gültig ist. Aus W^* kann mittels DSSS Dekodierung mit dem Spreizcode p und mittels BCH Dekodierung die detektierte Nachricht m^* berechnet werden.

Falls die Synchronisationssequenz nicht ausgelesen werden kann, muss zunächst eine Resynchronisierung durch Template-Detektion vorgenommen werden (vgl. Kapitel 2.1.2). Falls die Rekonstruktion des Bildes mittels Template-Detektion positiv ausfällt, wird an dem rekonstruierten Bild erneut die beschriebene Detektion vorgenommen. Kann das Template nicht gefunden werden, wird angenommen, dass das vorliegende Bild $g(x, y)$ unmarkiert ist.

2.1.2 Template-Einbettung und Detektion in der Fourierdomäne

Um gegen affine Transformationen robust zu sein, wird nach dem Verfahren von Pereira und Pun [16] nach der Wasserzeicheneinbettung in der Wavelet Domäne ein Template in der Fourierdomäne eingebettet. Dazu wird das markierte Bild $f'(x, y)$ zunächst durch das Hinzufügen von 0-Pixelwerten auf ein Bild mit den Dimensionen 1024×1024 gestreckt. Aus diesem Bild wird dann durch eine zweidimensionale Fouriertransformation ein Frequenzbild generiert. In dieses Frequenzbild werden 14 Template-Punkte eingebettet, die regelmäßig auf zwei Linien im oberen Teil des DFT Raumes verteilt werden (vgl. Abbildung 2.2). Um eine höhere Robustheit gegen JPEG-Kompression zu erhalten, wird dazu ein niedrigeres Frequenzband gewählt, als von Pereira

und Pun in [16] vorgeschlagen wird. Konkret bedeutet das, dass die Entfernung der Template-Punkte vom Mittelpunkt des Frequenzbildes insgesamt kleiner ist (f_{t1} und f_{t2} in Abbildung 2.2). Die Winkel der beiden Linien (θ_i) und die Radien der Punkte auf den Linien (r_{ij}) werden über den geheimen Schlüssels k mittels einer Pseudozufallsfunktion berechnet. Da Fourier-Frequenzbilder punktsymmetrisch im Mittelpunkt sind, werden die 14 Template-Punkte analog in der unteren Hälfte des Frequenzbildes eingebettet.

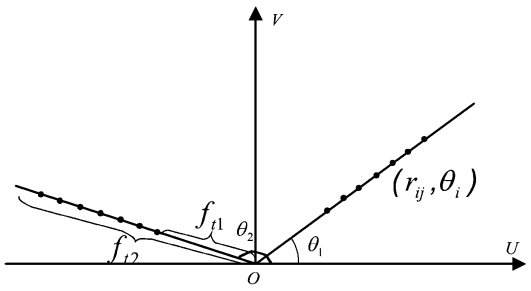

Abbildung 2.2.: Einbettung der Template-Punkte (Quelle: [16])

Die Einbettung der ausgewählten Template-Punkte wird durch die Verstärkung der Magnituden der Template-Punkte vorgenommen. Den neuen Wert eines Template-Punktes erhält man durch Addition des lokalen Mittelwertes aller Punkte in der Umgebung eines Template-Punktes auf die fünffache Standardabweichung der Punkte. In [16] wird nur die zweifache Standardabweichung verwendet. Das mit Template und Wasserzeichen markierte Bild im Pixelraum $f''(x, y)$ erhält man durch die Berechnung der inversen Fouriertransformation und Abschneiden der zu Beginn hinzugefügten Pixel.

Bei der Template-Detektion werden zunächst die 14 erwarteten Template-Punkte (θ_i, r_{ij}) aus dem geheimen Schlüssel k berechnet. Dann wird das zu untersuchende Bild $g(x, y)$ mit einem Bartlett-Fenster gefiltert. Durch Hinzufügen von 0-Pixelwerten wird das gefilterte Bild auf 1024×1024 Pixel gestreckt und eine zweidimensionale Fouriertransformation angewendet. Dann werden aus dem Frequenzbild die Positionen aller lokalen Maxima (p_{ui}, p_{vi}) extrahiert. Diese Maxima werden dann nach Winkelpositionen zum Mittelpunkt gruppiert. Die Maxima werden gruppenweise auf relative Übereinstimmung mit den erwarteten Radien r_{ij} um einen Skalierungsfaktor K überprüft.

Für alle Paare von Gruppen mit Übereinstimmungen wird daraufhin untersucht, ob deren Winkeldifferenz mit der Differenz aus θ_1 und θ_2 übereinstimmt. Aus den Paaren mit Übereinstimmungen kann dann eine Transformationsmatrix A berechnet werden, über welche die vorgenommenen affinen Transformationen rückgängig gemacht werden können. Da nur die obere Hälfte des Frequenzbildes untersucht wird, muss zusätzlich einmalig $180°$ auf die Template-Punkte einer Linie addiert werden und dann die beschriebene Suche nach lokalen Maxima und der Transformationsmatrix A nochmals durchgeführt werden.

Für jede gefundene Tranformationsmatrix A wird dann die mittlere quadratische Abweichung des Produktes der Transformationsmatrix A und der Koordinaten der lokalen Maxima zu den bekannten Template-Koordinaten berechnet. Die Transformationsmatrix mit der geringsten quadratischen Abweichung wird dann als Lösung gewählt, falls deren Abweichung kleiner als ein oberer Grenzwert T_d ist. Überschreitet die kleinste mittlere Abweichung diese Obergrenze, so wird angenommen, dass kein Template eingebettet wurde und somit auch kein Wasserzeichen vorhanden ist.

2.1.3 Resultate

Die in [12] beschriebenen Resultate beziehen sich auf 8-Bit Graustufen-Bilder mit einer Auflösung von 512×512. Der PSNR-Wert, der ein Indikator für das durch die Markierung hinzugefügte Rauschen ist, liegt stets über $42,5$ dB. Das eingebettete Wasserzeichen ist somit sehr transparent. Die höhere Einbettungsstärke und die Wahl eines niedrigeren Frequenzbandes bei der Template-Einbettung hat dabei nur einen geringen Effekt ($0,2$ dB) auf die Bildqualität des Wasserzeichenverfahrens, erhöht aber die Robustheit gegen affine Transformationen.

Das Wasserzeichen ist robust gegen JPEG-Kompression (bis zu einem Qualitätsfaktor von 10%), Gauß-Filterung, Schärfung, FMLR, Rotationen, Änderung des Seitenverhältnisses, Entfernung einzelner Spalten oder Zeilen des Bildes, allgemeine lineare Transformationen sowie Scherung. Ein eingebettetes Wasserzeichen kann ausgelesen werden, selbst wenn bis zu 65% der Bilddaten entfernt werden. Zu beachten ist allerdings, dass bei der Detektion stets die Auflösung des Originalbildes bekannt sein muss.

Die Einbettungsdauer beträgt für eine Implementierung des Verfahrens in C-Programmcode auf einem $1,7$ GHz Intel Pentium PC weniger als 4 Sekunden. Die Detektionsdauer beträgt zwischen 2 und 38 Sekunden.

2.2 Robustes DFT/LPM Bildwasserzeichen (Ridzoň und Levický)

Ridzoň und Levický beschreiben in [17] ein blindes, robustes Bildwasserzeichen, das die Diskrete Fouriertransformation (DFT) und Log-Polar Mapping (LPM) miteinander kombiniert. Bei der Einbettung wird aus einem geheimen Schlüssel ein Wasserzeichen-Bild generiert. Dieses Wasserzeichenbild wird dann einem inversen Log-Polar Mapping unterzogen und in die Magnitude der Fourier-Koeffizienten des Originalbildes eingebettet. Bei der Wasserzeichendetektion wird aus dem geheimen Schlüssel analog zur Einbettung das Wasserzeichenbild generiert und mit dem aus dem zu detektierdenden Bild extrahierten Wasserzeichenbild verglichen. An Hand der Korrelation der beiden Wasserzeichenbilder kann dann entschieden werden, ob das vorliegende Bild das ertwartete Wasserzeichen enthält. Bei diesem Verfahren kann somit keine Wasserzeichen-Nachricht eingebracht werden, es kann lediglich entschieden werden, ob ein vorliegendes Bild

mit dem geheimen Schlüssel markiert wurde. Im Folgenden wird der Algorithmus detailliert beschrieben.

2.2.1 Einbettung und Detektion in der DFT Domäne

Als Eingabe für die Einbettung dienen ein Graustufen-Bild I und der geheime Schlüssel K. Zunächst wird mittels *RIPEMD-160* [3] der Hash-Wert des geheimen Schlüssels K ermittelt. Aus diesem Hash wird dann ein Wasserzeichen-Bild W ($W(i,j) \in \{0,1\}$) generiert, dass die selben Dimensionen wie das Originalbild I besitzt. Durch Anwenden eines inversen Log-Polar Mappings (vgl. [14]) erhält man ein permutiertes Wasserzeichenbild W_{ILPM} mit dem selben Wertebereich und den selben Dimensionen.

Auf das Originalbild I wird eine zweidimensionale Fouriertransformation angewendet. Das permutierte Wasserzeichenbild W_{ILPM} wird in die Magnitude dieses Frequenzbildes eingebracht, die Phase bleibt unverändert. Abhängig von der Einbettungsstärke α werden die Koeffizienten des Frequenzbildes wie folgt verändert:

$$I^W_{DFT}(i,j) = \begin{cases} \frac{\alpha}{9} \sum_{i-1}^{i+1} \sum_{j-1}^{j+1} I_{DFT}(i,j) & \text{,falls} \quad W_{ILPM}(i,j) = 1 \\ \\ I_{DFT}(i,j) & \text{,falls} \quad W_{ILPM}(i,j) = 0 \end{cases}$$

Die Magnitude der zu markierenden Koeffizienten wird somit um das α-fache des lokalen Mittelwertes verstärkt. Um das markierte Bild im Pixelraum I^W zu erhalten, wird nach der Einbettung des Wasserzeichens in der Fourierdomäne eine inverse zweidimensionale Fouriertransformation angewendet. Abbildung 2.3 enthält eine Übersicht über den Einbettungsprozess.

Bei der Detektion des Wasserzeichens aus einem vorliegenden Bild wird lediglich der bei der Einbettung verwendete Schlüssel K benötigt. Analog zum Vorgehen bei der Wasserzeicheneinbettung, wird auf den geheimen Schlüssel K zunächst die Hash-Funktion *RIPEMD-160* angewendet und daraus das Wasserzeichenbild W generiert. Durch eine zweidimensionale Fouriertransformation erhält man aus dem zu testenden Graustufenbild I^T dessen Frequenzbild I^T_{DFT}. Aus der Magnitude dieses Frequenzbildes werden alle lokalen Maxima extrahiert und in das Wasserzeichenbild W^T_{ILPM} eingetragen. Durch Anwendung des Log-Polar Mappings erhält man aus W^T_{ILPM} das detektierte Wasserzeichenbild W^T. Auf W und W^T wird dann ein Korrelationstest durchgeführt, um entscheiden zu können, ob das vorliegende Bild tatsächlich mit dem geheimen Schlüssel K markiert wurde. Abbildung 2.3 enthält eine Übersicht des Detektionsprozesses.

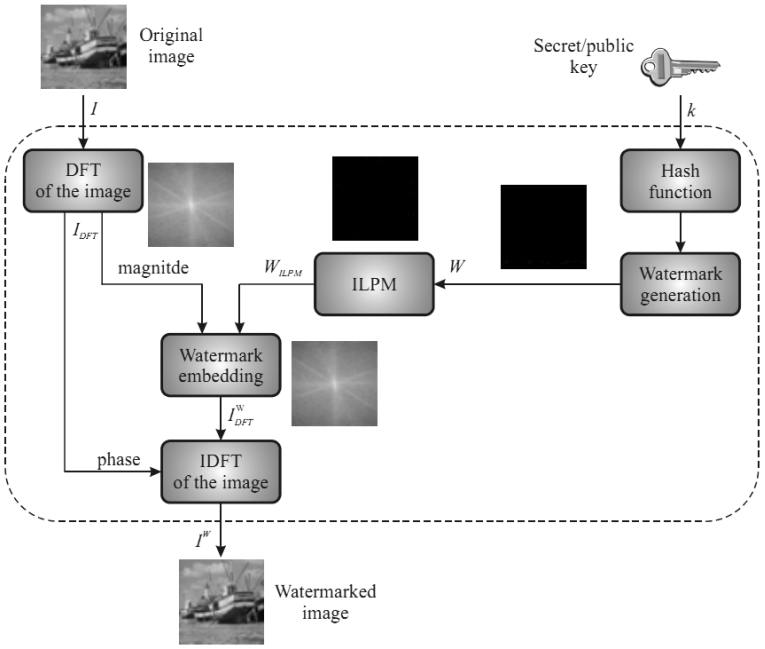

Abbildung 2.3.: Einbettung des Wasserzeichens in der DFT/LPM Domäne (Quelle: [17])

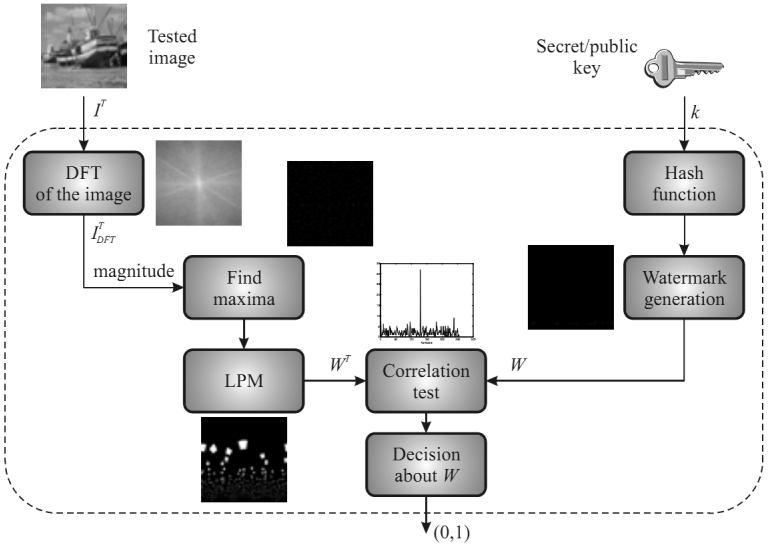

Abbildung 2.4.: Detektion des Wasserzeichens in der DFT/LPM Domäne (Quelle: [17])

2.2.2 Verwendung als Template

Wie in [17] und [18] beschrieben, kann dieser Algorithmus auch als Template in Kombination mit einem beliebigen anderen Bildwasserzeichenverfahren verwendet werden. Die Art des zusätzlich verwendeten Wasserzeichenverfahrens ist dabei frei wählbar.

Bei dieser Verwendung des Algorithmus wird zunächst mit dem zusätzlich gewählten Wasserzeichenverfahren ein Bildwasserzeichen W_1 in das Originalbild eingebettet. Danach wird der

Algorithmus von Ridzoň und Levický benutzt, um in das markierte Bild ein Wasserzeichen W_2 einzubringen, das aus einem Schlüssel K generiert wird. Der Schlüssel sollte dabei der selbe sein, der auch bei der Einbettung von W_1 verwendet wurde. Dieses zweite Wasserzeichen enthält dabei keine zusätzliche Wasserzeichennachricht, sondern kann lediglich dazu verwendet werden, Transformationen an einem markierten Bild zu identifizieren und rückgängig zu machen.

Bei der Detektion wird dann zunächst nach dem im vorigen Kapitel beschriebenen Verfahren eine Detektion des Wasserzeichens W_2 durchgeführt. Dabei kann zusätzlich zu der Information, ob das Template-Wasserzeichen vorhanden ist, ein Rotationswinkel und ein Skalierungsfaktor bestimmt werden, der die Transformationen bestimmt, die an dem markierten Bild vorgenommen wurden.

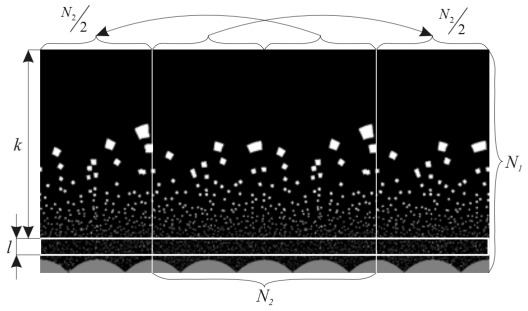

Abbildung 2.5.: Erweiterung des LPM-Wasserzeichenbildes zur Detektion von Rotationen (Quelle: [18])

Wie Abbildung 2.5 zeigt, wird für die Erkennung des Skalierungsfaktors dabei zunächst die linke Hälfte des Wasserzeichenbildes rechts angefügt und die rechte Hälfte des Bildes zusätzlich links angefügt. Auf das erweiterte Wasserzeichenbild $W^{T_{rot}}$ und das erwartete Wassereichenbild W wird dann folgende Korrelationsfunktion angewendet:

$$KT_{rot} = \sum_{i=k}^{k+l} \sum_{j=l}^{N_2} W(i,j) * W^{T_{rot}}(i, j+p) \text{ für } p = 0, 1, 2, .., N_2$$

An Hand der Verschiebung des höchsten Ausschlags in der Korrelationsfunktion KT_{rot} kann dann der Rotationswinkel bestimmt werden. Bei einer Verschiebung des höchsten Ausschlags nach links handelt es sich um eine Drehung gegen den Uhrzeigersinn, bei einer Verschiebung nach rechts dagegen um eine Drehung im Uhrzeigersinn (vgl. Abbildung 2.6).

Für die Detektion einer Skalierung muss keine Erweiterung des Wasserzeichenbildes vorgenommen werden. Hierfür wird folgende Korrelationsfunktion verwendet:

$$KT_{skal} = \sum_{i=k}^{k+l} \sum_{j=l}^{N_2} W(i,j) * W^{T}(i-q, j) \text{ für } q = k-1, k-2, ..., (N_1 - l)$$

Der Skalierungsfaktor lässt sich auch hier aus der Verschiebung des höchstens Ausschlags der

Korrelationsfunktion bestimmen. Eine Verschiebung nach links, bzw. rechts kann dabei auf eine Vergrößerung, bzw. Verkleinerung des Bildes zurückgeführt werden (vgl. Abbildung 2.7).

An Hand dieser Parameter kann dann eine Transformation des Bildes vorgenommen werden, bevor die Detektion des Wasserzeichens W_1 mit dem bei der Einbettung verwendeten Algorithmus vorgenommen wird. Die Detektion des Templates und somit auch die Detektion des Wasserzeichens schlägt fehl, wenn nicht bei beiden Korrelationsfunktionen KT_{skal} und KT_{rot} ein deutlicher Ausschlag gefunden werden kann.

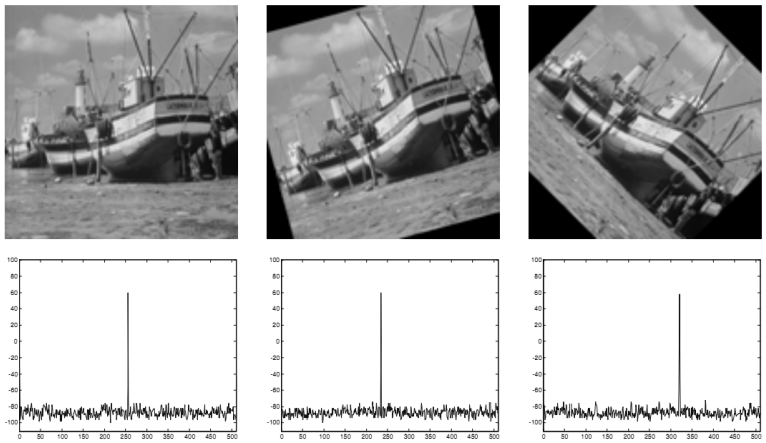

Abbildung 2.6.: Auswirkung von Rotationen auf die Korrelationsfunktion (Quelle: [18])

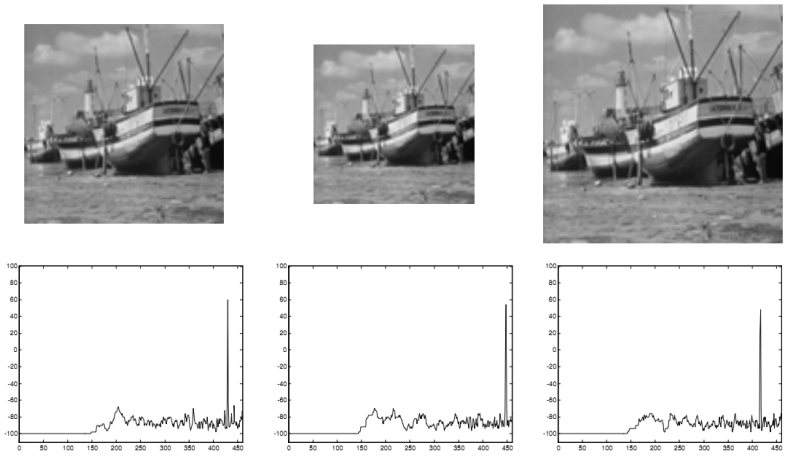

Abbildung 2.7.: Auswirkung von Skalierung auf die Korrelationsfunktion (Quelle: [18])

2.2.3 Resultate

In [17] werden einige Angriffe auf markierten Bilddateien ausgeführt und bewertet. Diese Angriffe beinhalten Rotation, Skalierung, JPEG-Kompression, Gauß-Filterung, Rauschen, Änderung

der Helligkeit, Entfernen von Bildteilen, sowie der Angriff „Stirmark $0,01$“[1]. Dabei zeigt sich dass der Algorithmus mit einer Einbettungsstärke von $\alpha_1 = 10$ gegen einige dieser Angriffe nicht robust ist. Abgesehen von dem Stirmark-Angriff, ist das Verfahren mit einer Einbettungsstärke von $\alpha_2 = 20$ und $\alpha_3 = 30$ dagegen robust gegen diese Angriffe.

[1] vgl. www.petitcolas.net/fabien/watermarking/stirmark/, Zugriff: 06.12.2011

3 Herausforderung

Die im Kapitel 2 vorgestellten Bildwasserzeichenverfahren besitzen eine hohe Robustheit und eine hohe Transparenz. Die gleichzeitige Erfüllung dieser beiden Kriterien kann nur durch hohe Komplexität erreicht werden. Diese hohe Komplexität geht im Allgemeinen mit hohem Rechenaufwand einher. Das in Kapitel 2.1 vorgestellte Bildwasserzeichen benötigt beispielsweise etwas weniger als vier Sekunden für die Einbettung einer Nachricht in ein Graustufen-Bild mit den Dimensionen 512×512 (1,7 GHz Pentium PC). Für größere Bilder ist bei diesem Verfahren dementsprechend mit einer erheblich längeren Einbettungsdauer zu rechnen. Beim Bildwasserzeichen ImageMark (vgl. Kapitel 4.1), welches die Grundlage dieser Arbeit ist, beträgt die Einbettungsdauer bei einem Bild mit den Dimensionen 2048×1536 über vier Sekunden (2 GHz Intel Core 2 Duo PC).

Durch eine Erweiterung kann das ImageMark Bildwasserzeichen außerdem dazu verwendet werden, PDF-Dateien oder Dokumente in verschiedenen E-Book Formaten zu markieren. Bei der Markierung eines solchen Dokumentes wird jedes in dem Dokument enthaltene Bild einzeln markiert. Die Markierungsdauer für ein solches Dokument beträgt somit, abhängig von der Anzahl der enthaltenen Bildelemente, ein Vielfaches der Zeit, die für die Markierung einer einzelnen Bilddatei benötigt wird.

Ein häufiger Anwendungsfall für digitale Wasserzeichen ist die individuelle Markierung von verkauften Werken in einem Online-Shop. Dabei wird die eindeutige Kennung eines Kunden oder die Rechnungsnummer in dessen erworbene Werke robust eingebettet, um der illegale Weiterverbreitung von diesen Werken durch den Käufer entgegen zu wirken. Wenn ein Benutzer ein gekauftes Werk im Internet verbreitet, so kann dies, sobald das markierte Werk wieder an den Rechteinhaber gelangt, auf den Käufer zurückgeführt werden. Dazu ist allerdings notwendig, dass der Verkäufer, bzw. Rechteinhaber aktiv nach markierten Werken sucht.

Entschließt sich ein Benutzer dazu, ein Werk bei einem Online-Shop zu kaufen, so muss sichergestellt werden, dass das individuell markierte Werk unmittelbar zum Download bereit steht. Lange Wartezeiten sind dabei absolut inakzeptabel, da sie dem Komfort und der Benutzerfreundlichkeit schaden.

Bei einer solchen Verwendung von Wasserzeichen sind auf der anderen Seite sowohl Robustheit, als auch Transparenz enorm wichtige Anforderungen. Zum Einen muss sichergestellt werden, dass die Qualität eines Werkes nicht unter der Wasserzeichenmarkierung leidet. Zum Anderen darf es nicht passieren, dass ein Benutzer das Wasserzeichen, welches seine Käuferkennung enthält, entfernen kann. Will man also die digitale Wasserzeichenmarkierung in einem solchen Szenario verwenden, so ist es notwendig einen Mechanismus zu finden, der die Einbettungsdau-

er drastisch verkürzt, ohne dass Robustheit und Transparenz des verwendeten Wasserzeichens darunter leiden.

Die Motivation dieser Arbeit ist es, die Einbettungsgeschwindigkeit des ImageMark Bildwasserzeichens zu beschleunigen. Das Prinzip des digitalen Wasserzeichen-Containers (vgl. Kapitel 4.2) stellt eine Möglichkeit dar, um die Einbettungsdauer zu verkürzen. Im folgenden Kapitel wird zunächst das ImageMark Bildwasserzeichen detailliert beschrieben und auf das Prinzip des Container-Verfahrens eingegangen. Auf Grund der Eigenschaften des ImageMark Bildwasserzeichens ist das Konzept des Wasserzeichen-Containers auf dieses nicht ohne Weiteres anwendbar (vgl. Kapitel 5.1). Die Herausforderung dieser Arbeit ist es daher, trotz der nicht unmittelbaren Vereinbarkeit von ImageMark und dem Konzept des Wasserzeichen-Containers, ein Container-Verfahren für ImageMark zu entwickeln.

4 Grundlagen

In diesem Kapitel werden die Grundlagen der in dieser Arbeit vorgestellten Verfahren beschrieben. Zum Einen wird die Funktionsweise des ImageMark-Bildwasserzeichens dargestellt, auf welches der entwickelte Bildwasserzeichen-Container aufbaut. Zum Anderen werden das Konzept des Wasserzeichen-Containers, sowie zwei Beispiele existierender Container-Verfahren aus dem Video- und Audio-Bereich beschrieben.

4.1 ImageMark-Bildwasserzeichen

Das am Fraunhofer SIT entwickelte Bildwasserzeichen-Verfahren ähnelt den beiden in Kapitel 2 beschriebenen Verfahren sehr. Auch ImageMark ist ein blindes und robustes Bildwasserzeichen, bei dem die Einbettung der Wasserzeichennachricht in der Frequenzdomäne vorgenommen wird. Außerdem verwendet ImageMark das selbe Verfahren zur Template-Einbettung [16] wie das Verfahren von Kang et al [12] (vgl. Kapitel 2.1). Der in dieser Arbeit vorgestellte Wasserzeichen-Container baut auf ImageMark auf, da für dieses Verfahren im Gegensatz zu den Verfahren aus Kapitel 2 eine vollständige Implementierung vorliegt.

Die robuste Einbettung der Wasserzeichennachricht wird bei ImageMark im Fourierraum vorgenommen. Über einen privaten Schlüssel werden pseudozufällig Koeffizienten ausgewählt, die abhängig von der einzubettenden Nachricht verändert werden. Die Einbettungsstärke des Wasserzeichens wird vom Benutzer ausgewählt und legt fest, wie stark die Koeffizienten im Fourierraum verändert werden. Bei der Detektion einer Wasserzeichennachricht aus einem markierten Bild werden wiederum über den privaten Schlüssel die zu untersuchenden Koeffizienten im Fourierraum bestimmt, um die eingebettete Nachricht auszulesen.

Da die Wasserzeichenmarkierung im Frequenzraum vorgenommen wird, verteilt sich die Gesamtenergie des Wasserzeichens über das gesamte Bild. Somit kann die eingebettete Nachricht auch noch detektiert werden, wenn Teile des Bildes entfernt werden. Das Verfahren ist außerdem robust gegen verlustbehaftete Komprimierung des markierten Bildmaterials, wie JPEG. Durch die Einbettung eines Templates kann das Wasserzeichen auch noch ausgelesen werden, nachdem affine Transformationen an einem markierten Bild vorgenommen wurden.

Als Wasserzeichen-Nachricht kann der Benutzer eine Bit-Sequenz mit maximaler Länge $l - 12$ oder aber auch eine ASCII-Zeichensequenz mit maximaler Länge $l/3 - 12$ verwenden. Dabei bezeichnet l die Anzahl der insgesamt einbettbaren Wasserzeichen-Bits (standardmäßig $l = 48$). Zwölf Bits werden dabei zur Fehlererkennung mittels *CRC12* verwendet. Wählt der Benutzer eine

Nachricht, welche die Maximallänge nicht voll ausnutzt, so wird die Nachricht mehrmals eingebettet.

Im Folgenden werden der Einbettungsprozess und der Detektionsprozess dieses Verfahrens detailliert beschrieben.

4.1.1 Wasserzeichen-Einbettung

Der Einbettungsprozess erhält als Eingabe ein Bild B, welches in einem gängigen Bildformat vorliegen sollte. Zum Auslesen der Bilddaten wird die Bibliothek CxImage[1] verwendet. Nach dem Auslesen der Bilddaten liegen diese als RGB-Werte vor. Ein zu markierendes Bild B wird dann zunächst in Helligkeits- und Farbanteil (L und C) unterteilt (vgl. Abbildung 4.3). Die Helligkeit eines Pixels entspricht dabei dem Y-Wert in der YUV Farbdarstellung. Das Helligkeitsbild L wird dann in quadratische Blöcke mit Seitenlänge N unterteilt (standardmäßig $N = 1024$). Falls die Länge oder die Breite des Bildes kein Mehrfaches von N sind, werden die Randblöcke mit dem Wert 0 (schwarz) aufgefüllt, sodass alle Blöcke die Dimensionen $N \times N$ besitzen.

Jeder Block des Helligkeitsbildes wird einer zweidimensionalen Fourier-Transformation unterzogen. Dann werden die Fourier-Koeffizienten in die Polardarstellung transformiert. Bei der Einbettung des Wasserzeichens wird allerdings nur die Magnitude der Fourier-Koeffizienten manipuliert, die Phase bleibt unverändert.

Um gegen affine Transformationen resistent zu sein, wird zunächst ein Template eingebettet (vgl. [16] und Kapitel 2.1.2). Die Koeffizienten, die für die Einbettung des Templates verwendet werden, befinden sich auf den Diagonalen des Frequenzbild-Blockes (rote Linien in Abbildung 4.1). Die Radien und Winkel der Punkte hängen dabei nicht von einem privaten Schlüssel ab, sondern sind festgelegt. Die Einbettung wird vorgenommen, indem die Magnitude dieser t Koeffizienten verstärkt wird. Den neuen Wert eines Template-Koeffizienten erhält man durch Addition des lokalen Mittelwertes aller Nachbarn eines Template-Punktes auf die zwölffache Standardabweichung aus allen Nachbarn. In [12] und [16] wird dafür nur die zwei- bzw. fünffache Standardabweichung verwendet. Durch dieses enorm robust eingebettete Template können bei der Wasserzeichen-Detektion eines vorliegenden Bildes somit affine Transformationen, die ein Angreifer vorgenommen haben könnte, rückgängig gemacht werden.

Die Koeffizientengruppen, die für die Einbettung der Wasserzeichen-Nachricht verwendet werden, befinden sich jeweils zwischen den Diagonalen und den Seitenhalbierenden eines Blockes (blaue Bereiche in Abbildung 4.1). Da die Koeffizienten-Matrix im Fourierraum stets punktsymmetrisch im Mittelpunkt ist, erhält man somit vier voneinander unabhängige Regionen, die zum Einbetten der Nachrichten-Bits verwendet werden können. Es werden jeweils u

[1] vgl. sourceforge.net/projects/cximage, Zugriff: 08.11.2011

Nachrichten-Bits pro Region eingebettet, sodass insgesamt $l = 4 \cdot u$ Nachrichten-Bits eingebettet werden können.

Zunächst wird die einzubettende Nachricht in vier Teilnachrichten aufgeteilt. Jede dieser Teilnachrichten wird einer Region zugeordnet (vgl. Abbildung 4.1). Für jedes Nachrichten-Bit werden aus dem geheimen Schlüssel k mit Hilfe einer Pseudozufallsfunktion n Koeffizienten in der entsprechenden Region ermittelt, deren Magnitude verändert wird, um ein Nachrichten-Bit einzubetten. Um diese Koeffizienten zu ermitteln, wird zunächst der MD5-Hash[2] des Schlüssels k berechnet. Der Hash-Wert wird dann als Initialisierung der Pseudozufallsfunktion verwendet. Die n Koeffizienten, die die Pseudozufallsfunktion liefert, werden wiederum in zwei Gruppen unterteilt. Die erste Gruppe besteht aus den ersten $n/2$ Koeffizienten, die zweite Gruppe aus der anderen Hälfte der Werte.

Nun wird die Summe über die Koeffizienten der ersten Gruppe Σ_0 und der zweiten Gruppe Σ_1 gebildet. Um eine 0 einzubetten gilt es nun, die Koeffizienten der beiden Gruppen so zu verändern, dass gilt $\Sigma_0 < \Sigma_1$. Analog muss erreicht werden, dass $\Sigma 0 > \Sigma_1$ gilt, um eine 1 einzubetten.

Diese Verschiebung in den Summen der beiden Gruppen wird erreicht, indem die Koeffizienten der einen Gruppen vergrößert und die der anderen Gruppe verkleinert werden. Dazu wird der Betrag jedes Koeffizienten mit einem Faktor f multipliziert. Soll der Betrag verkleinert werden, so gilt $f < 1$, um den Betrag zu vergrößern, gilt $f > 1$. Wie weit f von 1 abweicht hängt dabei von der Wasserzeichen-Stärke s, der Differenz $\Sigma^* = \Sigma_0 - \Sigma_1$ und vom Abstand eines Koeffizienten zum Mittelpunkt des gesamten Blockes ab.

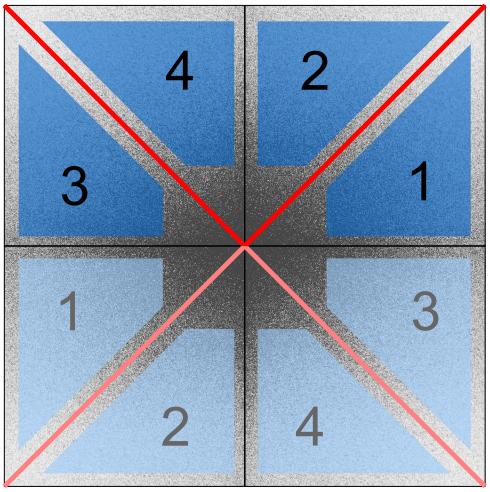

Abbildung 4.1.: Zur Markierung verwendete Fourier-Koeffizienten (blaue Bereiche: Wasserzeichen-Bits, rote Linien: Template)

[2] vgl. tools.ietf.org/html/rfc1321, Zugriff: 02.09.2011

Nachdem das Template und die Wasserzeichennachricht in einen Block eingebettet wurden, werden die Fourier-Koeffizienten zunächst wieder in die kartesische Form transformiert. Dann wird die inverse zweidimensionale Fourier Transformation durchgeführt. So erhält man aus einem mit Nachricht m und geheimem Schlüssel k markierten Block im Frequenzraum einen markierten Block im Pixelraum. Dieser gesamte Vorgang mit Fourier Transformation, Template-Einbettung, Wasserzeichenmarkierung und inverser Fourier Transformation wird für alle Blöcke des Helligkeits-Bildes L wiederholt, sodass ein markiertes Helligkeits-Bild $L_{m,k}$ entsteht.

Damit die Transparenz des Wasserzeichens gewahrt bleibt, wird eine visuelle Maskierung vorgenommen (vgl. [11], [19], [25]). Dabei wird zunächst aus dem Helligkeitsanteil des Originalbildes L eine Maske M berechnet. Diese Maske definiert für jeden Pixel, wie stark dessen Wert durch die Wasserzeichen-Markierung verändert werden darf. Wenn sich die direkten Nachbarwerte eines Pixels stark voneinander unterscheiden, erhält der Pixel einen hohen Maskenwert. Sind die Nachbarpixel kaum unterschiedlich, so erhält ein Pixel einen niedrigen Maskenwert. Besitzt ein Pixel einen hohen Masken-Wert, darf er stark verändert werden, bei einem kleinen Maskenwert, darf der zugehörige Pixel nur leicht verändert werden. Nachdem der Maskenwert für jeden Pixel bestimmt wurde, werden alle Werte auf das Intervall $[0, 1]$ normiert.

Aus dem Helligkeitsanteil des Originalbildes und des markierten Bildes wird nun ein Differenzbild $D = L_{m,k} - L$ berechnet. Die Maskierung wird blockweise vorgenommen, da die gesamte Wasserzeichenenergie eines Blockes von den Bildeigenschaften des Blockes abhängt. Neben der Verbesserung der Transparenz wird durch die Maskierung eine gleichmäßige Verteilung der Wasserzeichenenergie erreicht. Den Gewichtungsfaktor β eines Blockes x erhält man aus dem Verhältnis der Gesamtdifferenz zur maskierten Gesamtdifferenz des Blockes (Differenzpixel: D_{ij}, Maskenwerte: M_{ij}):

$$a_x = \sum D_{ij}^2 \qquad b_x = \sum (D_{ij}^2 \cdot M_{ij}^2) \qquad \beta_x = \sqrt{a_x / b_x}$$

Den neuen Wert eines Pixels im markierten, maskierten Bild $(L^*_{m,k})_{ij}$ erhält man unter Einbeziehung von Originalpixel L_{ij}, Differenzpixel D_{ij}, Maskenwert M_{ij} und Gewichtung $\beta_{x(ij)}$:

$$(L^*_{m,k})_{ij} = L_{ij} + \beta_{x(ij)} \cdot M_{ij} \cdot D_{ij}$$

Um nicht zu viel Wasserzeichenenergie durch die Maskierung zu verlieren, werden Pixel deren neuer Wert betragsmäßig kleiner als 1 ist, auf den Mindestdifferenzwert 1 bzw. -1 gesetzt, falls deren Maskenwert eine untere Grenze τ überschreitet:

$$(L^*_{m,k})_{ij} = \begin{cases} L_{ij} + 1 & \text{,falls} \quad 0 < \beta_{x(ij)} \cdot M_{ij} \cdot D_{ij} < 1 \quad \text{und} \quad M_{ij} > \tau \\ L_{ij} - 1 & \text{,falls} \quad -1 < \beta_{x(ij)} \cdot M_{ij} \cdot D_{ij} < 0 \quad \text{und} \quad M_{ij} > \tau \\ (L^*_{m,k})_{ij} & \text{,sonst} \end{cases}$$

Wie sich die visuelle Maskierung auf das markierte Bild auswirkt, ist in den verstärkten Differenzbildern in Abbildung 4.2 dargestellt. Ohne Markierung wiederholt sich das eingebettete Muster periodisch über das gesamte Bild. Durch die Maskierung werden glatte Flächen weniger stark verändert, Regionen mit Kanten und Details dagegen stärker hervorgehoben. Somit wird die Transparenz des Wasserzeichens enorm gesteigert ohne die Robustheit stark zu verringern.

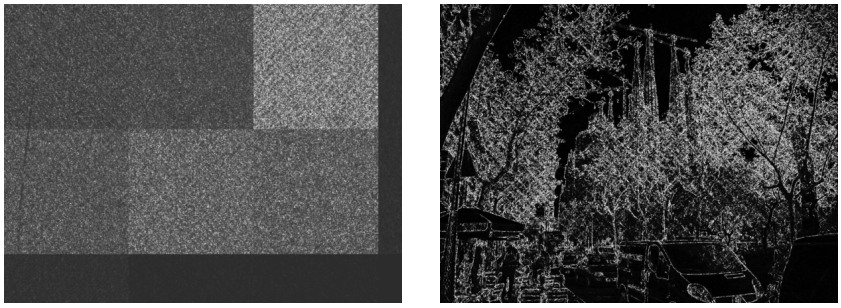

Abbildung 4.2.: Auswirkung der visuellen Maskierung - Gegenüberstellung von verstärkten Differenzbildern: links ohne visuelle Maskierung, rechts mit visueller Maskierung

Fügt man den Farb-Anteil des Originalbildes C wieder mit dem markierten und maskierten Helligkeitsanteil $L^*_{m,k}$ zusammen, so erhält man das markierte Bild $B^*_{m,k}$. Eine Übersicht über den gesamten Markierungsprozess ist in Abbildung 4.3 dargestellt (aus Gründen der Übersichtlichkeit ohne Template-Einbettung).

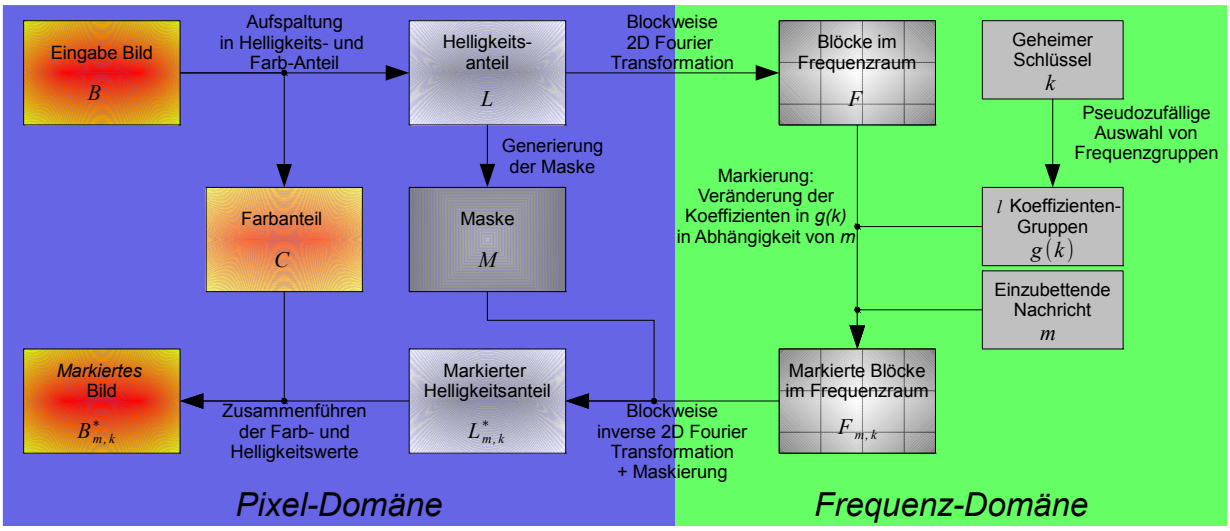

Abbildung 4.3.: Einbettungsprozesses des ImageMark Bildwasserzeichens

Als Bildformat für das markierte Bild kann der Benutzer aus einer Vielzahl gängiger digitaler Bildformate auswählen und für Formate, die dies zulassen, wie beispielsweise JPEG, einen Kompressionsfaktor q wählen. Außerdem kann der Benutzer die Stärke s des eingebetteten Wasserzeichens bestimmen. Die Dimensionen des resultierenden markierten Bildes sind mit denen des Eingangsbildes identisch. Die Veränderungen zum Originalbild sind auch für große Einbettungsstärken mit dem bloßen Auge nicht sichtbar. Lediglich durch die Erzeugung eines verstärkten Differenzbildes, wird das Wasserzeichen sichtbar (siehe Abbildung 4.4).

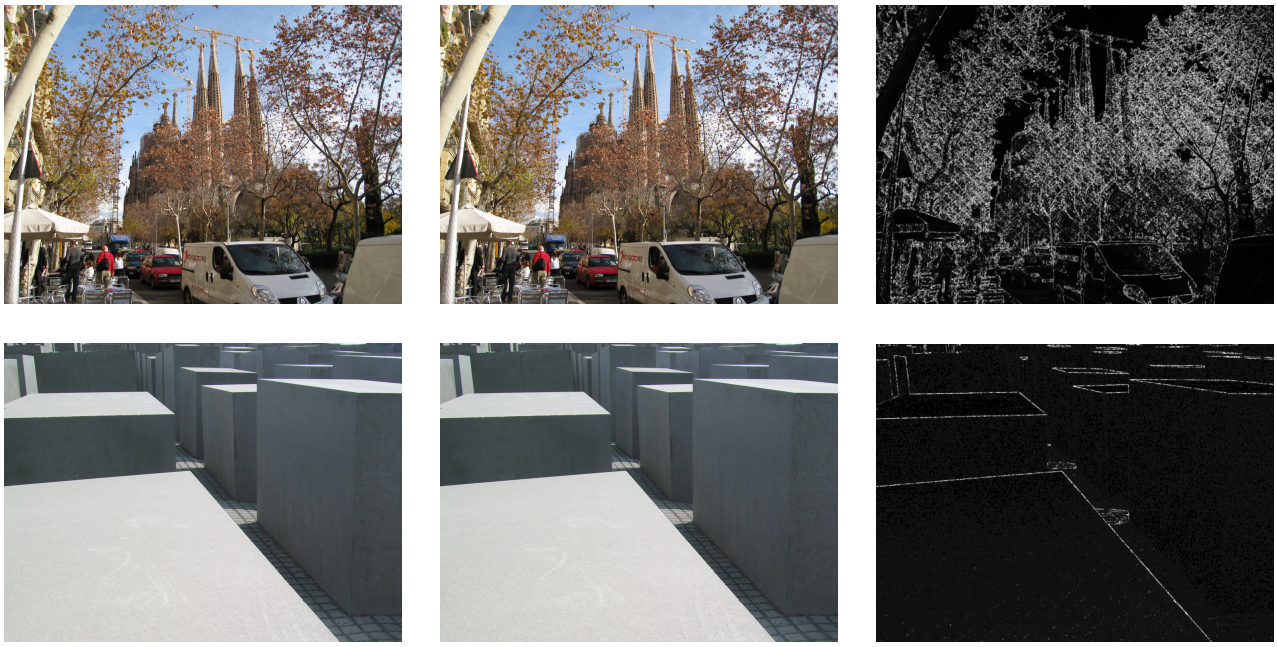

Abbildung 4.4.: Gegenüberstellung von Originalbild (links), markiertem Bild (mitte) und verstärktem Differenzbild (rechts)

4.1.2 Wasserzeichen-Detektion

Bei der Detektion kann aus einem Bild eine zuvor eingebettete Wasserzeichennachricht ausgelesen werden. Als Eingaben dieses Prozesses dienen die Bilddatei, aus der das Wasserzeichen ausgelesen werden soll, sowie ein geheimer Schlüssel. Die eingebettete geheime Nachricht kann nur mit dem selben Schlüssel ausgelesen werden, der bei der Einbettung verwendet wurde.

Analog zum Einbettungsprozess (vgl. letztes Unterkapitel) wird zunächst aus dem zu untersuchenden Bild B der Helligkeitsanteil des Bildes L extrahiert. Aus diesem wird blockweise durch eine zweidimensionale Fouriertransformation ein Frequenzbild F generiert. Durch eine deterministische Pseudozufallsfunktion, die als Initialisierung den MD5-Hash des privaten Schlüssels k erhält, werden l Gruppen von Koeffizientenpositionen generiert, die jeweils n Positionen enthalten. In jeder dieser Koeffizientengruppen wird nun die Summe aus den ersten $n/2$ Koeffizienten und die Summe der zweiten Hälfte der Werte (Σ_0 und Σ_1) gebildet. Aus dem Qutoienten der bei-

den Summen $\sigma = \Sigma_0/\Sigma_1$ kann nun das eingebettete Wasserzeichen-Bit bestimmt werden. Wenn gilt $\sigma < 1$, wird eine 0 ausgelesen, für $\sigma > 1$ eine 1. Des Weiteren kann σ dazu verwendet werden, einem detektierten Nachrichten-Bit eine Wahrscheinlichkeit zuzuordnen. Ist der Quotient σ größer als ein Schwellwert ϵ_1 oder kleiner als ϵ_0, so wurde mit großer Wahrscheinlichkeit das detektierte Nachrichten-Bit eingebettet. Gilt allerdings $\epsilon_0 < \sigma < \epsilon_1$, so kann nicht mit Sicherheit behauptet werden, dass das detektierte Nachrichten-Bit tatsächlich eingebettet wurde. In diesem Fall wird angenommen, dass die Verteilung der Koeffizienten zufällig ist und kein Nachrichten-Bit eingebettet wurde.

Aus jedem Block des Frequenzbildes wird somit eine Nachricht ausgelesen, deren Nachrichten-Bits Zusatzinformationen über die Stärke des detektierten Nachrichten-Bits enthalten. Die ausgelesenen Wasserzeichennachrichten aus allen Blöcken können dann zu einer Nachricht kombiniert werden. Die Gewichtung der einzelnen Nachrichten-Bits wird dabei mit einbezogen. Somit kann auch für die gesamte ausgelesen Nachricht eine Aussage darüber getroffen werden, wie sicher es ist, dass die detektierte Nachricht tatsächlich eingebettet wurde.

Falls die Stärke einzelner Nachrichten-Bits nicht hoch genug ist, oder falls die in der ausgelesenen Nachricht enthaltene Prüfsumme nicht gültig ist, so kann keine Wasserzeichen-Nachricht ausgelesen werden. In diesem Fall wird eine Suche nach dem eingebetten Template vorgenommen. Wird das Template detektiert, so können affine Transformationen, die an dem Bild vorgenommen wurden, rückgängig gemacht werden. Auf diesem rekonstruierten Bild wird dann erneut eine Wasserzeichendetektion nach dem beschriebenen Vefahren durchgeführt.

Abbildung 4.5 enthält eine Gesamt-Übersicht des Detektionsverfahrens (aus Gründen der Übersichtlichkeit ohne Template-Detektion).

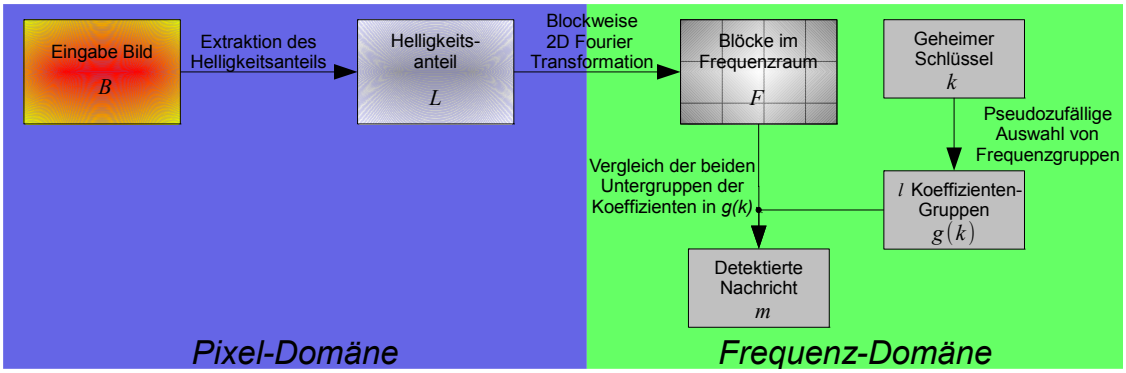

Abbildung 4.5.: Detektionsprozess des ImageMark Bildwasserzeichens

4.2 Container-Verfahren bei digitalen Wasserzeichen

Der Ansatz aller Container-Verfahren für digitale Wasserzeichen ist es, den Prozess der Wasserzeichen-Einbettung in zwei Phasen zu unterteilen. Die Intention ist es dabei alle aufwän-

digen Rechenoperationen der Wasserzeichen-Markierung nur einmalig in der ersten Phase, dem „Containering“ durchzuführen. Auf diese Weise kann eine markierte Version des Mediums in der zweiten Phase, dem Shuffling, jeweils sehr schnell erzeugt werden.

Beim Containering wird eine sogenannte Containerdatei erstellt, die das markierte Medium mit allen Permutationen einbettbarer Nachrichten enthält. Dabei wird für jeden Wert jedes Nachrichten Bits eine markierte Version des Mediums erzeugt.

Im Folgenden wird das Prinzip des Wasserzeichen-Containers für beliebige Wasserzeichenverfahren mit *zweiwertigen* Nachrichten-Bits beschrieben. Das Container-Verfahren ist im Allgemeinen aber nicht nur für Wasserzeichenverfahren mit zweiwertigen Nachrichten-Bits anwendbar. Die beschriebene Vorgehensweise ist analog auch auf Wasserzeichen-Verfahren mit beliebig n-wertigen Nachrichten-Bits anwendbar ($b \in \{0, 1, ..., n-1\}$). Sind die möglichen Werte eines Nachrichten-Bits also 0 und 1, so wird eine mit $m_0 = \{0\}^l$ und eine mit $m_1 = \{1\}^l$ markierte Version des Mediums erzeugt, wobei l die Länge der einbettbaren Nachrichten ist. Diese 0- und 1-markierten Versionen des Mediums werden dann in der Container-Datei abgelegt. Die Erzeugung dieser Container-Datei kann rechenaufwändig sein, muss aber nur ein einziges Mal für ein Medium durchgeführt werden.

In der zweiten Phase, dem Shuffling, wird aus einer Container-Datei ein markiertes Medium ausgegeben. Dabei wird für jedes Nachrichten-Bit der einzubettenden Nachricht der zugehörige markierte Teil des Mediums aus der Container-Datei entnommen und ausgegeben. So wird aus allen zu den Nachrichten-Bits gehörenden Teilen des Mediums das markierte Medium zusammengesetzt.

Das Beispiel in Abbildung 4.6 zeigt ein Video mit fünf voneinander unabhängigen Frames. Ist die einzubettende Nachricht beispielsweise “10011”, so wird aus dem Container für das erste Bit der mit 1-markierte Frame entnommen, für das zweite und das dritte Bit der 0-markierte Frame und für das vierte und fünfte Bit wiederum der 1-markierte Frame.

Um ein Container-Verfahren auf ein vorhandes Wasserzeichen-Verfahren anwenden zu können, müssen drei Voraussetzungen erfüllt sein (vgl. [26]):

1. Die Wasserzeichen-Nachricht besteht aus Nachrichten-Bits, die in voneinander unabhängige Bereiche des Mediums, „elementare Einheiten“, eingebettet werden

2. Das markierte Medium kann in diese elementaren Einheiten unterteilt werden

3. In welche elementare Einheit ein Nachrichten-Bit eingebettet wird, ist unabhängig von dem Wert des Nachrichten-Bits

Die ersten beiden Regeln bewirken, dass für jedes Nachrichten-Bit ein entsprechender Bereich im markierten Medium existiert, der dieses Nachrichten-Bit enthält. Diese „Elementaren Einheiten“ müssen im markierten Medium voneinander unabhängig und trennbar sein, damit der entsprechende Bereich für jedes Nachrichten-Bit im Container abgelegt werden kann. Die dritte

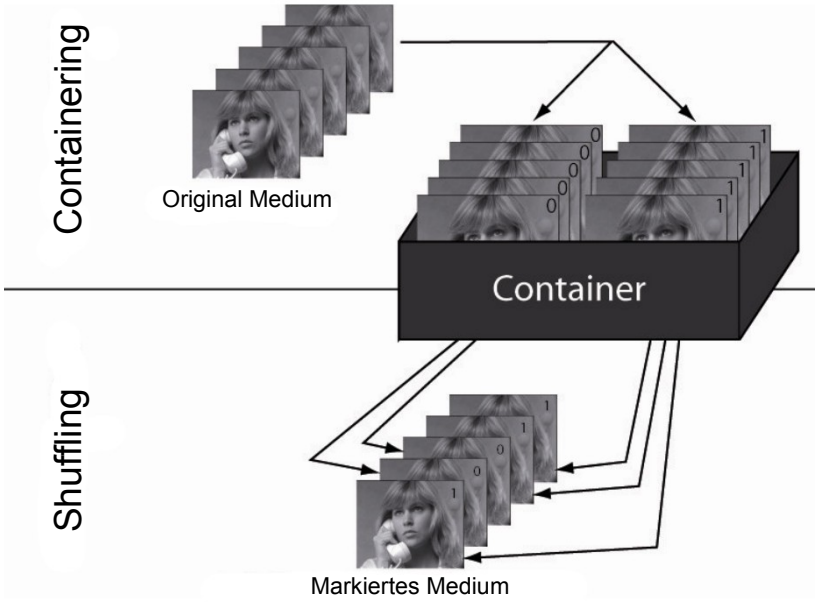

Abbildung 4.6.: Shuffling eines markierten Mediums mit der Nachricht "10011" aus einer Container-Datei (Quelle: [26])

Regel stellt sicher, dass die Bereiche im Medium, die das Nachrichten-Bit an einer bestimmten Position enthalten, unabhängig von dessen Wert in der 0- und 1-markierten Version stets an der selben Stelle des Mediums zu finden sind.

Im Folgenden werden zunächst die Vor- und Nachteile von Container-Verfahren beschrieben. Des Weiteren wird ein Container-Verfahren für Videowasserzeichen und ein Container-Verfahren für Audiowasserzeichen betrachtet. Dabei wird besonders auf die Unterschiede zu den Eigenschaften des ImageMark-Bildwasserzeichens eingegangen.

4.2.1 Vor- und Nachteile des Container-Verfahrens

Ein häufiges Anwendungsszenario für digitale Wasserzeichen ist die Verwendung von Transaktionswasserzeichen in einem Online-Shop. Beim Verkauf eines digitalen Werkes in einem solchen Online-Shop an einen Kunden wird dabei dessen eindeutige Kunden-Identifikationsnummer in das verkaufte Medium als robustes Wasserzeichen eingebettet. Diese Markierung wird in dem Moment gestartet, in dem der Kunde den Kauf abschließt und somit den Download anfordert.

In einem solchen Anwendungsszenario ist eine sehr geringe Markierungsdauer neben Robustheit und Transparenz eine wichtige Anforderung. Wünschenswert ist eine Markierungsgeschwindigkeit, die so schnell ist wie die maximale Upload-Geschwindigkeit des Servers für ein verkauftes Werk. Oder, falls die Video- oder Audiodaten vom Käufer in einem Stream abgerufen werden, sollte bei der Wasserzeichen-Einbettung mindestens Echtzeit-Geschwindigkeit

erreicht werden. Werden diese Anforderungen nicht erreicht, so müsste der Käufer mit einer Wartezeit rechnen, was dem Komfort-Gedanken eines Online-Shops eindeutig widerspricht.

Bei der herkömmlichen Wasserzeichen-Markierung ohne die Verwendung von Container-Verfahren werden Echtzeit-Faktoren von maximal 1 bis 10 erreicht (vgl. [23], [26]). Geht man von einem Echtzeitfaktor 5 und einer Auslastung des Prozessors von 50% durch einen Markierungsvorgang aus, so können maximal 10 Markierungsvorgänge parallel in Echtzeit erfolgen. Ein solches System ist somit kaum skalierbar und für die Anwendung in der realen Welt kaum geeignet.

Das Container-Verfahren liefert dafür eine Lösung. Das Shuffling ist im Vergleich zum herkömmlichen Markierungsprozess hoch-performant, da lediglich Lese- und Schreiboperationen und einfache Rechenoperationen durchgeführt werden müssen. Dabei werden für die Wasserzeichen-Markierung von Audio- und Videodateien Echtzeit-Faktoren von 50, 115 oder sogar 100 bis 1000 (vgl. [9], [22], [23]) erreicht.

Neben der höheren Performanz besitzt das Container-Verfahren einen weiteren Vorteil gegenüber der herkömmlichen Wasserzeichen-Markierung. Bei robusten Wasserzeichen-Verfahren wird im Allgemeinen ein geheimer Schlüssel zur Einbettung einer Nachricht in ein Medium, sowie zur Detektion einer Nachricht aus einem markierten Medium verwendet.

Für die Erzeugung eines markierten Mediums sind also das Original-Medium, sowie der geheime Schlüssel notwendig. Möchte man nun aber die Wasserzeichen-Markierung in einer unsicheren Umgebung vornehmen, wie z.B. einem Online-Shop, wo Entwickler, Administratoren und Andere Zugriff auf sämtliche Daten besitzen, so wird ein sicheres Schlüssel-Verwaltungssystem benötigt.

Die Verwendung des Container-Verfahrens löst dieses Problem. Das Containering kann in einer sicheren Umgebung unter Verwendung des geheimen Schlüssels vorgenommen werden. Die Container-Datei kann dann in die unsichere Umgebung übertragen werden. Da der geheime Schlüssel beim Shuffling nicht benötigt wird, kann die eigentliche Einbettung dann in dieser unsicheren Umgebung stattfinden, ohne dass der geheime Schlüssel preisgegeben werden muss (vgl. Abbildung 4.7).

Ein Nachteil der Verwendung des Container-Verfahrens ist der höhere Speicherbedarf. Die beim Containering erzeugten und für das Shuffling benötigten Container-Dateien benötigen mehr Speicherplatz als die unmarkierten Originaldateien, die als Eingabe für die herkömmliche Wasserzeichen-Markierung dienen. Die Containerdateigröße ist dabei von der jeweiligen Implementierung abhängig. Die Vergrößerungsfaktoren von bisherigen Implementierungen im Audio- und Videobereich liegen zwischen $1,5$ und 2 (vgl. [22], [26]).

Die lange Rechenzeit, die beim Containering anfällt, sollte nicht vernachlässigt werden. Jedoch muss dieser Vorgang, wie bereits erwähnt, nur einmal durchgeführt werden und ist daher im Vergleich zu der enorm hohen Geschwindigkeit beim Shuffling durchaus akzeptabel.

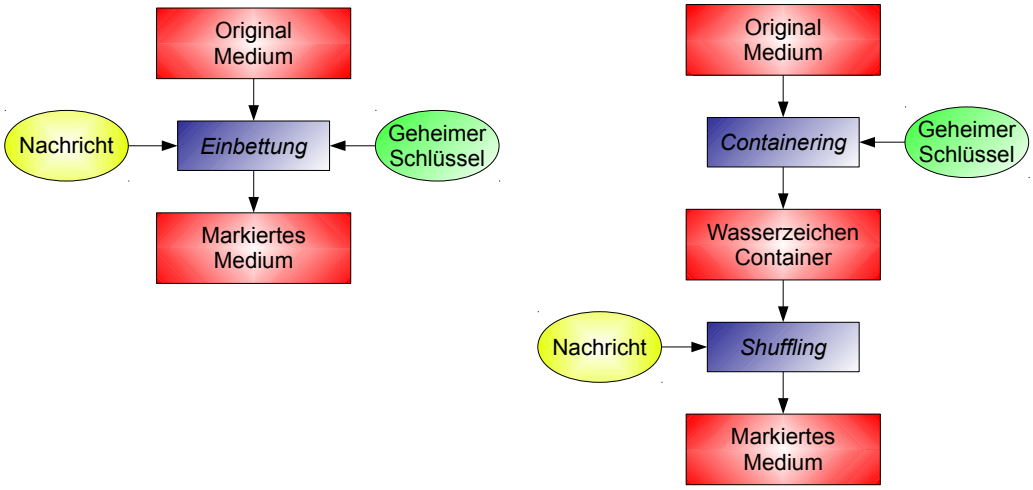

Abbildung 4.7.: Vergleich von normaler Wasserzeichen-Einbettung (links) und Container-Verfahren (rechts), vgl. [22]

4.2.2 Container-Verfahren bei Videowasserzeichen

In diesem Abschnitt wird ein Container-Verfahren für Videowasserzeichen beschrieben (vgl. [26]). Zur Einbettung der Wasserzeichennachrichten in ein Video wird die in [4] und [24] vorgestellte Methode verwendet, welche eine Erweiterung des Verfahrens von Fridrich aus [6] zum Einbetten von Mustern in Bilder darstellt. Bei dem betrachteten Wasserzeichen-Verfahren für Videos werden einzelne Frames eines Videos markiert. Die Einbettung erfolgt durch Subtraktion (einzubettende 0) bzw. Addition (einzubettende 1) eines Musters auf DCT-Luminanz-Blöcke der Video-Frames. Sowohl die Generierung des Musters, als auch die Auswahl der Blöcke, die markiert werden, geschieht in Abhängigkeit von einem privaten Schlüssel k. Bei der Einbettung können mehrere Bits in einen Video-Frame eingebettet werden.

Die Nachricht, die in jedem Frame eingebettet wird, besteht aus drei Teilen. Sie enthält eine Synchronisationssequenz s, die die Position des vorliegenden Nachrichtenteils in der gesamtem Wasserzeichennachricht festlegt, einen Teil der gesamten einzubettenden Wasserzeichennachricht $w(s)$, sowie eine Prüfsumme über Synchronisationssequenz und den Teil der Wasserzeichennachricht $CRC(s|w(s))$. Die Gesamtnachricht sieht somit folgendermaßen aus: $s|w(s)|CRC(s|w(s))$. Im Container-Modus wird bei diesem Verfahren stets nur ein Bit der Wasserzeichennachricht eingebettet, $w(s)$ hat also stets die Länge 1.

Beim Erzeugen einer Container-Datei wird dann wie folgt vorgegangen. Zunächst wird eine 0-markierte und eine 1-markierte Version des Videos erzeugt. Dabei wird der Wechsel zur nächsten Position der Nachricht, also die Veränderung von s ($w(s)$ bleibt jeweils immer 0 oder 1), stets am Beginn einer GOP (Group Of Pictures) vorgenommen. Ein Nachrichten-Bit Wechsel kennzeichnet somit den Anfang einer neuen elementaren Einheit. Somit entspricht eine elementare Einheit

im Video-Container einer GOP im Videostrom. Da es eine Eigenschaft einer GOP ist, dass die darin enthaltenen Frames keine Abhängigkeiten zu Frames außerhalb der GOP enthalten, wird somit sichergestellt, dass jede elementare Einheit in sich abgeschlossen ist. Somit entstehen keine Abhängigkeiten zwischen den elementaren Einheiten des Containers. In der Container-Datei wird dann eine 0-markierte und eine 1-markierte Version jeder GOP abgelegt. Somit enthält die Container-Datei stets alle möglichen Wasserzeichen-Nachrichten einer festgelegten Länge.

Beim Shuffling wird abhängig von der einzubettenden Nachricht jeweils die 0- oder 1-markierte GOP ausgewählt und ausgegeben (vgl. Abbildung 4.6).

Im Gegensatz zu einem digitalen Bild existiert bei digitalen Videos eine zeitliche-Dimension. Bei dem beschriebenen Videowasserzeichenverfahren wird über diese zeitliche Dimension, wie beschrieben, eine Unterteilung der Nachricht in einzelne Nachrichten-Bits und somit elementare Einheiten vorgenommen. Die direkte Übertragung des Container-Verfahrens für Videowasserzeichen auf ein Container-Verfahren für Bildwasserzeichen ist somit nicht möglich. Die zeitliche Dimension, über die die Aufteilung der Nachricht in einzelne Nachrichten-Bits erfolgt, existiert bei digitalen Bildern nicht.

4.2.3 Container-Verfahren bei Audiowasserzeichen

In diesem Abschnitt wird ein Container-Verfahren für Audiowasserzeichen vorgestellt. Dem in [22] beschriebenen Container-Verfahren liegt das PCM-Audiowasserzeichen zu Grunde, welches in [21] beschrieben wird. Das Container-Verfahren ist allerdings auch für andere PCM-Audiowasserzeichenverfahren anwendbar.

Als Eingabe für den Einbettungsprozess des verwendeten PCM-Audiowasserzeichens wird eine binäre Wasserzeichennachricht m, sowie ein geheimer Schlüssel k benötigt. Bei der Wasserzeicheneinbettung wird die zu markierende Datei zunächst in nicht überlappende Frames unterteilt. Jeder dieser Frames wird dann mit einem Wasserzeichen-Bit ($0/1$) der Nachricht m markiert. Die Einbettung wird, ähnlich wie bei ImageMark, durch die Manipulation der Fourier-Koeffizienten vorgenommen. Dabei werden über den privaten Schlüssel durch eine Pseudozufallsfunktion zwei Gruppen von Koeffizienten ausgewählt. Das Größenverhältnis zwischen den Summen der Koeffizienten dieser beiden Gruppen wird dann verändert. Abhängig davon, ob eine 0 oder eine 1 eingebettet wird, kann durch Vergrößerung der Koeffizienten der einen Gruppe und Verkleinerung der Koeffizienten der Anderen Gruppe ein Nachrichten-Bit eingebracht werden. Das Ausgabeprodukt des Einbettungsprozesses ist eine PCM-Audiodatei, die aus einzelnen markierten Audio-Frames besteht. Diese Frames überschneiden sich nicht und sind somit unabhängig voneinander, sodass sie elementare Einheiten darstellen.

Das Container-Verfahren teilt diesen Einbettungsprozess nun in zwei Teile auf: *Preprocessing*

Stage (Containering) und *Rendering Stage* (Shuffling). Beim Containering wird für jeden Frame *Sample*-weise das Differenzsignal DA des 0- und 1-markierten Frames ($A(0)$, $A(1)$) zum Original-Frame A berechnet:

$$DA(0) = A - A(0) \qquad DA(1) = A - A(1)$$

Durch ADPCM-Komprimierung erhält man aus den Differenz-Frames $DA(0)$, $DA(1)$ komprimierte Differenz-Frames $CA(0)$, $CA(1)$. Für jeden Frame wird dann in der Containerdatei die Originalversion des Frames, sowie die beiden komprimierten Differenz-Frames abgelegt. Eine Containerdatei hat somit die in Abbildung 4.8 dargestellte Struktur.

Frame A	CA(0)	CA(1)	Frame B	CB(0)	CB(1)	...

Abbildung 4.8.: Struktur einer PCM-Wasserzeichen Containerdatei (vgl. [21])

Beim Shuffling kann aus den in der Container-Datei enthaltenen Daten sehr schnell eine individuell markierte PCM Audiodatei erzeugt werden. Für jeden Frame wird zunächst der Original-Frame aus der Containerdatei ausgelesen. Abhängig davon, welches Nachrichten-Bit für einen Frame eingebettet werden soll, wird dann das komprimierte Differenzsignal $CA(0)$ oder $CA(1)$ ausgelesen. Dieses wird dann dekomprimiert und auf den Original-Frame *Sample*-weise aufaddiert, sodass man die markierte Version des Frames erhält:

$$A(0) = A + DA(0) \qquad \text{bzw.} \qquad A(1) = A + DA(1)$$

Man erhält die gesamte markierte Audiodatei, indem diese Operationen für alle im Container enthaltenen Frames durchgeführt werden. Beim Shuffling müssen somit nur ADPCM-Dekompression, Additionsoperationen, sowie Lese- und Schreiboperationen durchgeführt werden, um eine individuell markierte PCM Audiodatei zu erhalten. Man erreicht so Echtzeitfaktoren von über 50, wobei bei der Einbettung ohne Containering nur Echtzeitfaktoren von etwas mehr als 1 erreicht wurden. Die Container-Dateien besitzen in etwa die $1,5$-fache Größe der Eingabedateien.

Das Verfahren bietet außerdem noch mögliche Optimierungen. Will man beispielsweise mehrere individuell markierte Versionen aus einer Containerdatei generieren, so muss man die Dekompression und Addition nicht mehrmals durchführen. Es ist in diesem Fall sinnvoll, die Rechenoperationen nur einmalig durchzuführen und alle zu erzeugenden markierten PCM-Dateien gleichzeitig zu erzeugen. Jeder 0- und 1-markierte Frame kann dann mehrmals in die Ausgabe-

dateien geschrieben werden. Erzeugt man N markierte PCM-Dateien auf einmal, so spart man mindestens bei $N - 2$ Einbettungsoperationen Rechenzeit ein.

Im Gegensatz zu einem digitalen Bild existiert auch bei digitalen Audiodateien eine zeitliche-Dimension. Beim beschriebenen Container-Verfahren wird über diese zeitliche Dimension, die Unterteilung der zu den einzelnen Nachrichten-Bits gehörenden Audio-Frames, also den elementaren Einheiten, vorgenommen. Die direkte Übertragung dieses Container-Verfahrens auf ein Container-Verfahren für Bildwasserzeichen ist somit nicht möglich.

5 Konzeption eines Container-Verfahrens für ImageMark

In diesem Kapitel wird die Konzeption zweier Container-Verfahren für das ImageMark-Bildwasserzeichen beschrieben. Zunächst wird auf die Problematik eingegangen, dass das ImageMark-Bildwasserzeichen gegen die Voraussetzungen für Container-Verfahren verstößt. Daraufhin werden zwei Container-Verfahren beschrieben, die dieses Problem auf unterschiedliche Weise lösen.

5.1 Erfüllung der Container-Voraussetzungen

Zunächst gilt es zu überprüfen, ob das ImageMark Bildwasserzeichen die in Kapitel 4.2 genannten Voraussetzungen erfüllt:

1. Die Wasserzeichen-Nachricht besteht aus Nachrichten-Bits, die in voneinander unabhängige Bereiche des Mediums, *elementare Einheiten*, eingebettet werden
2. Das markierte Medium kann in diese *elementaren Einheiten* unterteilt werden
3. In welche *elementare Einheit* ein Nachrichten-Bit eingebettet wird, ist unabhängig von dem Wert des Nachrichten-Bits

Bei ImageMark wird die Einbettung der Nachrichten-Bits durch die Veränderung bestimmter Koeffizienten im Fourierraum vorgenommen (vgl. Kapitel 4.1.1). Die elementaren Einheiten bei diesem Verfahren entsprechen somit den Koeffizientengruppen $g(k)$, die abhängig vom einzubettenden Nachrichten Bit verändert werden. Diese Koeffizientengruppen sind im Fourierraum durchaus unabhängig voneinander, somit wird Voraussetzung 1 erfüllt.

Das Ausgabeprodukt der Einbettung ist allerdings ein Bild im Pixelraum. In der Pixeldomäne sind die elementaren Einheiten allerdings nicht mehr voneinander trennbar. Die Wasserzeichen-Energie ist nach der inversen Fourier-Transformation über das gesamte Bild verteilt. Somit wird Voraussetzung Zwei verletzt.

Voraussetzung Drei wird wiederum erfüllt. Die Koeffizientengruppen $g(k)$, die zur Markierung verwendet werden, sind ausschließlich vom geheimen Schlüssel k abhängig und werden den Nachrichten-Bits zugeordnet. Sie sind nicht von den Werten der jeweiligen Nachrichten-Bits abhängig.

Es gilt nun eine Lösung zu finden, die es möglich macht ein Container-Verfahren für ImageMark zu entwickeln, trotz der Verletzung von Voraussetzung Zwei. Offensichtlich kann ein Container-Verfahren für ImageMark nicht dadurch auskommen beim Shuffling nur Lese- und Schreiboperationen und einfache Rechenoperationen durchzuführen. Es muss vielmehr eine Möglichkeit

gefunden werden, die es erlaubt die elementaren Einheiten in einer möglichst kompakten Form abzubilden, die in einer Container-Datei gespeichert werden kann. Diese Abbildung der elementaren Einheiten sollte beim Shuffling sehr schnell in eine markierte Bild-Datei umgewandelt werden können.

5.2 Container-Verfahren im Pixelraum

Das Konzept des ersten Ansatzes ist es, für jedes Nachrichten-Bit eine eigene 0- und 1-markierte Version des gesamten Bildes zu erzeugen. Die zu den Nachrichten-Bits gehörenden Regionen des Bildes sind in der Pixeldomäne nicht voneinander trennbar, da die Wasserzeichen-Energien der Nachrichten-Bits im Pixelraum jeweils über die komplette Fläche des Bildes verteilt sind. Für eine Nachrichtenlänge l erhält man somit $2 \cdot l$ Bilder (je eine 0- und 1-markierte Version pro Nachrichten-Bit), die beim Containering erzeugt werden müssen. Diese Bilder enthalten jeweils ausschließlich Helligkeitswerte, da der Farbanteil des Originalbildes nicht markiert wird.

Aus dem originalen Helligkeitsbild und den markierten Helligkeitsbildern lassen sich Differenzbilder erzeugen. In der Container-Datei können dann nur die Differenzen der markierten Bilder zum Originalbild zusammen mit dem Originalbild gespeichert werden.

Da die Wasserzeichenenergie eines einzelnen eingebetteten Nachrichten-Bits in der Pixeldomäne sehr gering ist, häufig kleiner als ein halber Pixelwert, genügt es nicht, ganzzahlige Pixel-Werte in der Container-Datei zu speichern. Da es aber auch zu viel Speicherplatz benötigen würde, diese Werte als Fließkomma-Zahlen zu speichern, wird ein Mapping vorgenommen. Zunächst werden Minimum und Maximum des Differenzbildes berechnet. Für die somit errechnete Spanne an Werten des Differenzbildes wird dann eine Umrechnungsfunktion auf ganzzahlige Werte von $1-255$ erzeugt. Der Wert 0 bleibt für Zahlen reserviert, deren Betrag kleiner als eine Schranke ϵ ist:

$$m_i = \begin{cases} 0 & \text{,falls} \quad -\epsilon < p_i < \epsilon \\ \lceil ((p_i - min)\frac{254}{max-min}) + 1.5 \rceil & \text{,sonst} \end{cases}$$

m_i bezeichnet den zu berechnenden ganzzahligen Pixelwert, p_i den Fließkomma-Pixelwert, max und min bezeichnen Maximum und Minimum der Fließkomma-Pixelwerte des Differenzbildes, $\lceil x \rceil$ die Abrundungsfunktion.

Da die Pixelwerte eines markierten und maskierten Differenz-Helligkeitsbildes, bei der in dieser Arbeit verwendeten Parametrisierung, nie um mehr als 8 Pixelwerte von 0 abweichen, erhält man durch das Mapping eine Auflösung von ca. $0,07$. Ein einzelner (gerundeter) Fließkomma Differenzpixelwert kann somit durch einen 1 Byte großen Wert dargestellt werden. Speichert man nun zeilenweise die Pixel-Werte des Differenz-Helligkeitsbildes, so werden für ein Differenz-Helligkeitsbild mit den Dimensionen $X \times Y$ folglich $X \cdot Y$ Bytes an Speicherplatz benötigt.

Durch die visuelle Maskierung werden viele Pixelwerte der Differenzbilder auf Null gesetzt. Diese Beschaffenheit lässt sich ausnutzen, um den benötigten Speicherplatz etwas zu verringern. Über die Differenz-Helligkeitspixel wird eine Lauflängenkodierung vorgenommen. Allerdings ausschließlich für ganzzahlig gerundete Differenzpixel mit dem Wert 0. Für diese Werte wird jeweils die Anzahl der aufeinander folgenden Vorkommen als 1 Byte Wert geschrieben. Dann wird jedes der Differenz-Bilder mit dem Kompressionsverfahren ZIP komprimiert. Diese Komprimierung verringert die Dateigröße nochmals in etwa um den Faktor $0,6$.

Jedes der komprimierten $2 \times l$ Differenz-Helligkeitsbilder wird in der Container-Datei abgespeichert. Zusätzlich wird das Originalbild mit eingebettetem Template in der Container-Datei abgelegt. Um dabei ebenfalls Speicherplatz zu sparen wird dieses Bild im JPEG Format mit einem durch den Benutzer zu wählenden Qualitätsfaktor q abgespeichert.

Beim Shuffling wird zunächst das Originalbild dekomprimiert und in den RGB-Farbraum konvertiert, dabei müssen die RGB-Werte als Fließkommawerte behandelt werden. Abhängig von der einzubettenden Nachricht werden die zu den Nachrichten-Bits gehörenden Differenzbilder ausgewählt. Jedes dieser Differenz-Helligkeitsbilder muss zunächst dekomprimiert und dekodiert werden, da die Fließkomma-Pixelwerte ZIP-komprimiert und als abgebildete Ganzzahlwerte mit Lauflängenkodierung in der Container-Datei gespeichert sind. Nach der Dekodierung der Pixelwerte kann dann das Differenzbild Pixel-weise auf das RGB-Originalbild aufaddiert werden. Dabei muss keine Umrechnung vorgenommen werden, da Y-Werte im YUV-Farbraum exakt einem R-, G- und B-Pixelwert im RGB-Farbraum entsprechen.

5.2.1 Multi-Bit Differenzbilder

Um beim Shuffling kostenaufwendige Additionsoperationen einzusparen, kann man schon beim Containering mehrere Differenzbilder zusammenfassen. Anstatt für zwei Nachrichten-Bits je zwei Differenzbilder für 0- und 1-markierte Versionen zu erzeugen, können stattdessen auch direkt alle vier möglichen Differenzbilder für zwei Nachrichten-Bits zusammengefasst werden. Man erhält anstatt der vier Differenzbilder, die nur einzelne Nachrichten-Bits enthalten ($L_{\{0\}_0,k}$, $L_{\{1\}_0,k}$, $L_{\{0\}_1,k}$, $L_{\{1\}_1,k}$), vier Differenzbilder, die jeweils zwei Nachrichten-Bits enthalten ($L_{\{00\}_0,k}$, $L_{\{01\}_0,k}$, $L_{\{10\}_0,k}$, $L_{\{11\}_0,k}$). So kann beim Shuffling die Hälfte der Additionsoperationen bei gleich bleibender Dateigröße eingespart werden.

Diese Vorgehensweise lässt sich auf beliebig viele zusammengefasste Differenzbilder erweitern. Die so eingesparten Additionsoperationen gehen dann allerdings mit steigenden Dateigrößen einher, wie Tabelle 5.1 zeigt.

Zusammengefasste Differenzbilder	Benötigte Additionen	Vergrößerungsfaktor		
1	l	1	=	$2/2$
2	$l/2$	1	=	$4/4$
3	$l/3$	$1,\overline{3}$	=	$8/6$
4	$l/4$	2	=	$16/8$
5	$l/5$	$3,2$	=	$32/10$
6	$l/6$	$5,\overline{3}$	=	$64/12$
...	...	...		

Tabelle 5.1.: Dateigrößen bei Multi-Bit Differenzbildern

Die Containerdateien sind für dieses Verfahren im Verhältnis zu den Originaldateien allerdings enorm groß. Es ist daher fraglich ob diese Optimierung für mehr als zwei zusammengefasste Differenzbilder, also mit einer Vergrößerung der Containerdateien, sinnvoll ist.

5.2.2 Differenz aus 1- und 0-markierter Version

Vorausgesetzt es ist keine Anforderung, dass das Originalbild aus der Containerdatei extrahiert werden kann, so kann eine andere Optimierung vorgenommen werden, um Speicherplatz beim Schreiben der Container-Dateien, sowie Additionsoperationen beim Shuffling zu sparen. Anstatt das Originalbild mit eingebettetem Template in der Container-Datei abzulegen, wird dabei ein mit $\{0\}^l$ markiertes Bild mit eingebettetem Template in der Container-Datei abgelegt. Für jedes Nachrichten-Bit m_i wird dann nur noch ein komprimiertes Helligkeits-Differenzbild $L_{\{1\}_i-\{0\}_i,k}$ benötigt, welches die Differenz des $\{1\}_i$-markierten Bild zum $\{0\}_i$-markierten Bild enthält.

Für das mit $\{0\}^l$ markierte Bild wird einmalig die visuelle Maskierung mit Mindestdifferenzwert $|1|$ durchgeführt, um diese 0-Bits robust einzubetten. Damit Nachrichten-Bits mit dem Wert 1 ebenfalls robust eingebettet werden, muss die Mindestdifferenz auch in jedes Differenzbild eingebracht werden. Tut man dies nicht, so verlieren die 1-Bits deutlich an Robustheit. Ein Mindestdifferenzwert von $v = |1|$ würde dabei allerdings die Transparenz des Wasserzeichens zerstören, da diese Mindestdifferenz für jedes Nachrichten-Bit mit dem Wert 1 auf einige Pixel aufaddiert, bzw. subtrahiert würde. Um einen Kompromiss zwischen Robustheit und Bildqualität, bzw. Transparenz zu finden, wird daher für die Differenzbilder ein Mindestdifferenzwert $v < |1|$ gewählt (vgl. Kapitel 7.3).

Durch die Anwendung dieser Optimierung wird zum Einen der Speicherbedarf der Differenzbilder in der Container-Datei um 50% reduziert, da statt $2 \times l$ Differenzbildern nur noch l Differenzbilder benötigt werden. Zum Anderen wird beim Shuffling durchschnittlich die Hälfte der Additionsoperationen eingespart. Allerdings wird die Rechenzeit, die für das Shuffling benötigt wird, durch diese Optimierung unregelmäßig. Abhängig von den Nachrichten-Bits der

einzubettenden Nachricht werden unterschiedlich viele Additionsoperationen benötigt. Enthält die einzubettende Nachricht wenige 0 Bits und viele 1 Bits, so fallen viele Pixel-weise Additionsoperationen an und das Shuffling dauert überdurchschnittlich lange. Bei wenigen 1 Bits wird die Einbettungsdauer dagegen unter dem Mittelwert bleiben. In jedem Fall wird die Einbettungsdauer aber durch die Einführung dieser Optimierung reduziert. Ohne die Anwendung der Optimierung beträgt die Anzahl der benötigten Additionsoperationen $l \times X \times Y$, wobei l die Länge der Wasserzeichennachricht, X die Breite und Y die Höhe des Bildes in Pixeln bezeichnet. Die Anzahl der benötigten Pixel-weisen Additionsoperationen durch die angewandte Optimierung beträgt dagegen nur noch $l_1 \times X \times Y$ mit $l_1 \leq l$, wobei l_1 die Anzahl der 1 Bits in der Wasserzeichen-Nachricht bezeichnet. Im schlechtesten Fall wird also die Nachricht $\{1\}^l$ eingebettet, wobei dann $l_1 = l$ gelten würde. In diesem Fall entspricht die Anzahl der benötigten Pixel-weisen Additionen exakt der Anzahl der Additionen ohne die Optimierung. In allen anderen Fällen kann die Einbettungsgeschwindigkeit gesteigert werden.

Die Container-Datei enthält somit folgende Komponenten:

- Mit Template und $\{0\}^l$ markierte Version des Bildes $B^*_{\{0\}^l,k}$ (JPEG komprimiert)
- Pro Nachrichten-Bit ein Differenzbild aus der mit $\{1\}_i$ und $\{0\}_i$ markierten Version des Helligkeitsbildes $L_{\{1\}_i-\{0\}_i,k}$ (ZIP komprimierte, auf ganzzahlige Werte abgebildete Fließkommawerte)

Die Abbildungen 5.1 und 5.2 enthalten eine Übersicht des Containerings und des Shufflings für das Verfahren im Pixelraum (aus Gründen der Übersichtlichkeit ohne Template-Einbettung). Die rot markierten Elemente sind in der Container-Datei enthalten und beschreiben somit die Grenze zwischen dem Containering und dem Shuffling.

5.3 Container-Verfahren im Fourierraum

Die robuste Wasserzeichen-Einbettung wird bei ImageMark durch eine Veränderung einzelner Koeffizienten im Fourierraum vorgenommen. Die Wasserzeichen-Energie eines Nachrichten-Bits ist dadurch nach der inversen Fourier-Transformation über das gesamte Bild verteilt. Eine Zuordnung der Nachrichten-Bits auf bestimmte Region des Bildes oder sogar einzelne Pixel ist somit nicht mehr möglich. Das erste beschriebene Container-Verfahren für ImageMark benötigt daher für jedes Nachrichten-Bit eine 0- und eine 1-markierte Version des gesamten Bildes.

Diesem zweiten Ansatz liegt die Idee zu Grunde, die elementaren Einheiten, also die veränderten Koeffizientengruppen im Fourierraum, ohne Transformation in die Pixeldomäne in der Container-Datei abzulegen. Da die elementaren Einheiten im Fourierraum unabhängig voneinander sind, muss somit für das gesamte Bild nur eine 0-markierte und eine 1-markierte Version

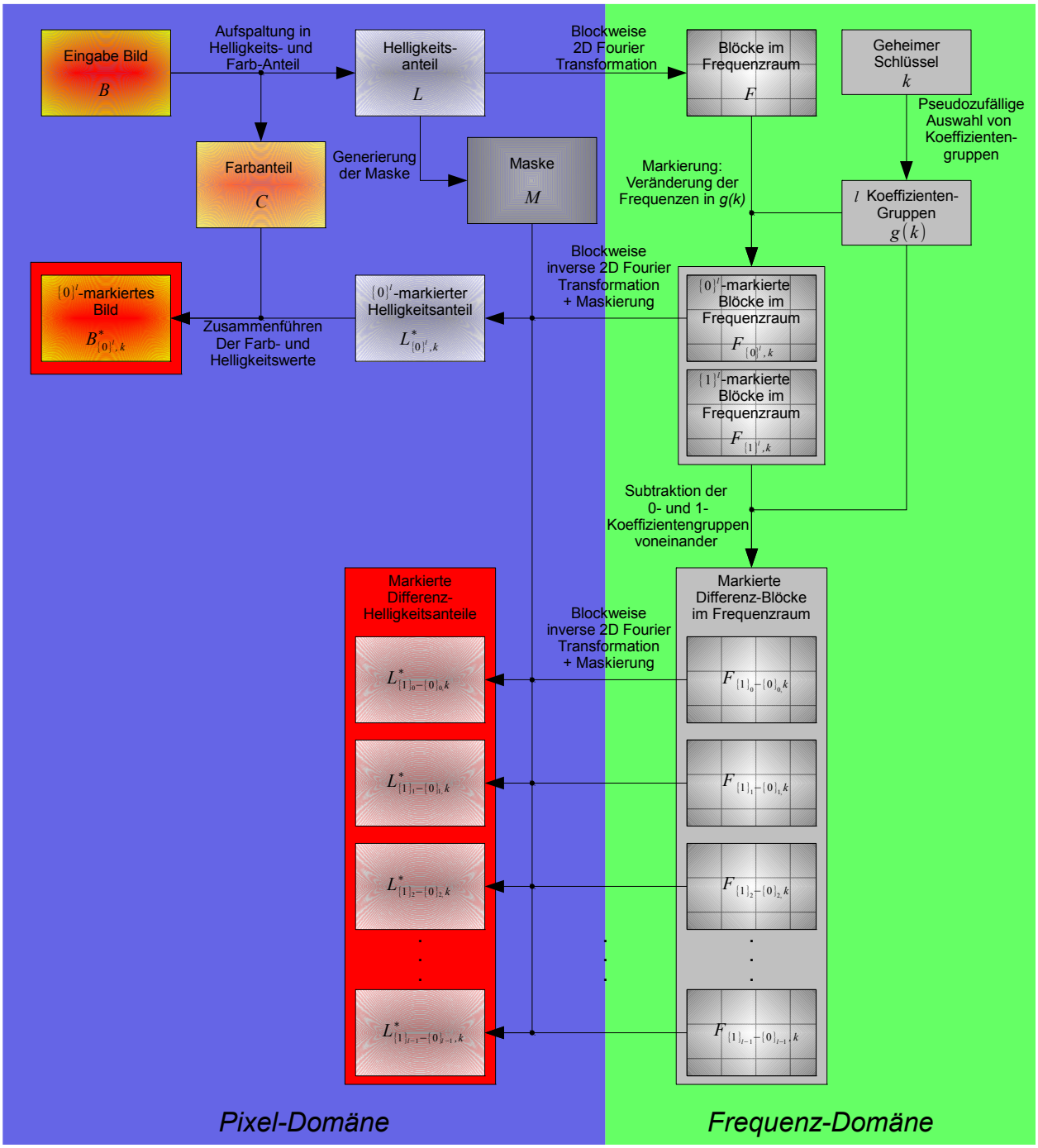

Abbildung 5.1.: Containering-Prozess des ImageMark Bildwasserzeichens in der Pixeldomäne (rot markierte Elemente werden in der Container-Datei gespeichert)

des Frequenzbildes ($F_{\{0\}^l,k}$ und $F_{\{1\}^l,k}$) im Container vorhanden sein. Beide Frequenzbilder enthalten auch die bei der Einbettung des Templates manipulierten Koeffizienten. Sie unterscheiden sich lediglich in den $l \cdot n$ Koeffizienten, die zur Einbettung verwendet werden. Die übrigen Koeffizienten stimmen überein. Im Container werden somit nur die beiden markierten Versionen des Helligkeitsbildes im Frequenzraum ($F_{\{0\}^l,k}$ und $F_{\{1\}^l,k}$), sowie die Positionen der Koeffizientengruppen $g(k)$, die zu einem Nachrichten-Bit gehören, benötigt.

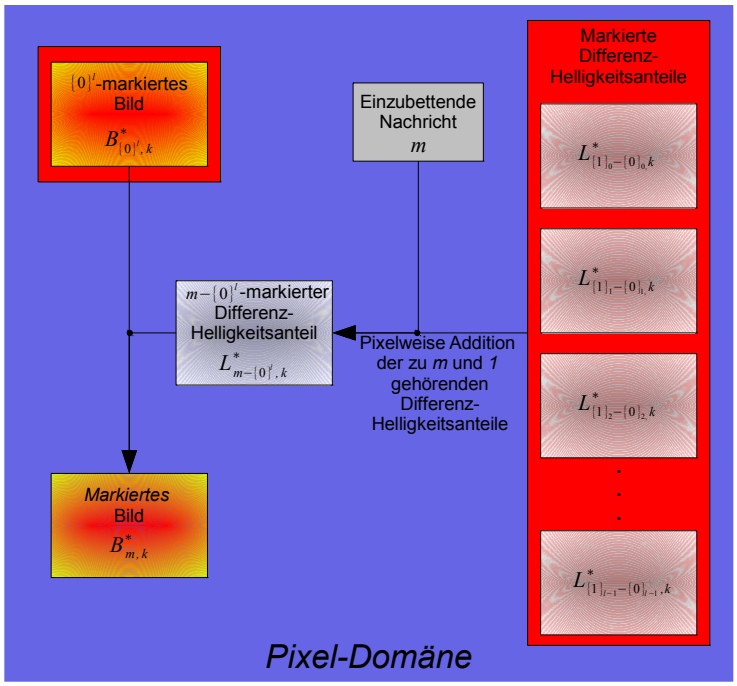

Abbildung 5.2.: Shuffling-Prozess des ImageMark Bildwasserzeichens in der Pixeldomäne (rot markierte Elemente sind in der Container-Datei enthalten)

Bei der Wasserzeicheneinbettung liegen die Koeffizienten in Polarform vor. Dabei wird, wie in Kapitel 4.1.1 beschrieben, lediglich die Magnitude manipuliert, während die Phase unverändert bleibt. Da die Fourier-Koeffizienten bei der inversen Fourier-Transformation in der kartesischen Form benötigt werden, werden die beiden markierten Frequenzbilder zunächst in diese Form konvertiert, bevor sie in der Container-Datei abgelegt werden. Zusätzlich werden im Container noch die Maske M, die pro Pixel einen Fließkomma-Wert aus dem Intervall $[0, 1]$ enthält, sowie die Farbanteile des Originalbildes C benötigt.

Beim Shuffling kann ein Frequenzbild mit der eingebetteten Nachricht m erzeugt werden, indem zunächst eine Kopie des mit 0 markierten Frequenzbildes $F_{m,k} = F_{\{0\}^l,k}$ erzeugt wird. Dann werden für jedes Nachrichten-Bit aus m mit dem Wert 1 die zugehörigen Koeffizienten aus dem mit 1 markierten Frequenzbild $F_{\{1\}^l,k}$ in $F_{m,k}$ übertragen. Aus diesem markierten Frequenzbild kann durch die blockweise inverse Fourier-Transformation das zugehörige Bild im Pixelraum $L_{m,k}$ generiert werden.

Dieses Bild muss dann zunächst mit der im Container vorliegenden Maske M blockweise maskiert werden. Durch das Zusammenführen des markierten und maskierten Helligkeitsbildes $L^*_{m,k}$ und des Farbanteils C erhält man dann dann das markierte Bild $B^*_{m,k}$.

5.3.1 Differenz zum Original

Das beschrieben Verfahren lässt sich optimieren, indem man nicht das gesamte markierte Frequenzbild der Helligkeitsanteile in der Containerdatei ablegt, sondern nur die Differenz zu den Koeffizienten des Originalbildes. Dabei werden zunächst die mit $\{0\}^l$ und $\{1\}^l$ markierten Frequenzbilder $F_{\{0\}^l,k}$ und $F_{\{1\}^l,k}$, jeweils inklusive Template, generiert. Dann werden aus diesen die Differenzbilder zum Frequenzbild des Originalbildes (ohne Template) berechnet:

$$F_{\{0\}^l-,k} = F_{\{0\}^l,k} - F$$

$$F_{\{1\}^l-,k} = F_{\{1\}^l,k} - F$$

Die Fourierkoeffizienten in $F_{\{0\}^l,k}$ und $F_{\{1\}^l,k}$ stimmen mit denen in F an den Stellen überein, die weder durch die Wasserzeicheneinbettung noch durch die Einbettung des Templates verändert werden. Die Differenzbilder $F_{\{0\}^l-,k}$ und $F_{\{1\}^l-,k}$ enthalten somit nur noch wenige Koeffizienten mit einem Wert ungleich Null.

Speichert man für diese Differenzbilder nur die Koeffizienten, die ungleich Null sind, so erhält man $l \cdot n + t$ zu speichernde Koeffizienten pro Block (l: Einzubettende Nachrichten-Bits, n: Manipulierte Koeffizienten pro Nachrichten-Bit, t: durch Template-Einbettung manipulierte Koeffizienten). Für jeden Koeffizienten ungleich Null werden dann die x- und y-Position im Frequenzbild, sowie der reale und der imaginäre Anteil des Koeffizienten in der kartesischen Form des 0- und des 1-markierten Frequenz-Differenzbildes benötigt. Da es sich bei dem realen und dem imaginären Anteil eines Koeffizienten um Fließkomma-Werte handelt, die viel Speicherplatz benötigen, wird für diese Werte ein Mapping auf ganzzahlige Werte vorgenommen. Berechnet man Minimum und Maximum aller vorliegender Werte, so kann hier ein Mapping auf 2 Byte große, ganzzahlige Werte (*Short*) vorgenommen werden.

Zusammen mit diesen Differenzbildern im Frequenzraum wird dann, anstatt des Farbanteils C des Originalbildes, das gesamten Originalbild im Container gespeichert. Um dabei Qualitätsverlust zu vermeiden, wird das Eingabebild unabhängig vom Dateiformat ohne erneute Kompression direkt übernommen.

Da die Repräsentation der Maskenwerte im Container nach dem Originalbild den meisten Speicherplatz benötigt (ein Fließkommawert pro Pixel) wird für diese Werte ebenfalls ein Mapping auf ganzzahlige Werte vorgenommen. Da die Maskenwerte im Intervall $[0, 1]$ liegen, werden hierfür nur 4 Bit verwendet. Somit liefern die gerundeten Maskenwerte eine Auflösung von $1/16 = 0,0625$. Außerdem werden die Maskenwerte nach diesem Mapping noch ZIP-komprimiert um zusätzlich Speicherplatz zu sparen.

Das bei der visuellen Maskierung verwendete Verfahren (vgl. 4.1.1) wird durch den so entstehenden Rundungsfehler allerdings verfälscht, da der β-Wert eines maskierten Bildblockes

durch die Rundung verändert wird. Eine Lösung dieses Problems besteht darin, den β-Wert eines Blockes beim Containering zu bestimmen und zusätzlich in der Container Datei abzulegen. Dieses Vorgehen ist möglich, da der β-Werte eines Blockes für verschiedene eingebettete Nachrichten kaum unterschiedliche Werte annimmt.

Die Container-Datei enthält somit folgende Komponenten:

- Unverändertes Eingabebild B
- Maske M (Auf ganze Zahlen abgebildete Fließkommawerte, ZIP komprimiert)
- Markierte Differenzbilder im Frequenzraum inklusive Template $F_{\{0\}^l-,k}$ und $F_{\{1\}^l-,k}$ (Position, Real- und Imaginärteil der Werte ungleich 0)
- Positionen der zu den Nachrichten-Bits gehörenden Koeffizienten $g(k)$ (x- und y- Position)

Beim Shuffling werden zunächst alle Werte aus $F_{\{0\}^l-,k}$ in ein neues Differenzbild $F_{m-,k}$ kopiert. Dann werden für jedes Nachrichten-Bit aus m mit dem Wert 1 die zugehörigen Koeffizienten aus $F_{\{1\}^l-,k}$ ausgewählt und in $F_{m-,k}$ kopiert. Nach Abschluss dieser Kopier-Operationen erhält man ein Differenzbild im Fourierraum $F_{m-,k}$, welches alle Fourierkoeffizienten der Nachricht m und des Templates enthält. Nach der inversen Fourier-Transformation erhält man ein markiertes Helligkeitsdifferenzbild $L_{m-,k}$, das noch blockweise zu maskieren ist. Um die visuelle Maskierung vornehmen zu können, werden die Maskenwerte M und die β-Werte (ein Wert pro Block) benötigt. Die β-Werte können direkt aus der Containerdatei ausgelesen werden. Da die Maskenwerte jedoch nicht nur als ganzzahlige 4-Bit Werte, sondern zusätzlich noch ZIP-komprimiert in der Containerdatei vorliegen, müssen diese Werte zunächst dekomprimiert werden, bevor sie zu Fließkommawerten dekodiert werden können. Diese Maskenwerte enthalten durch die Abbildung auf 4-Bit große Werte einen Rundungsfehler. Um durch diesen Rundungsfehler nicht an Robustheit zu verlieren wird jeder Maskenwert, der größer als Null ist, um einen geringen Anteil μ vergrößert (vgl. Kapitel 7.3). Durch die Maskierung erhält man dann aus $L_{m-,k}$ ein markiertes und maskiertes Helligkeitsdifferenzbild $L^*_{m-,k}$.

Das mit der Nachricht m markierte Bild $B^*_{m,k}$ erhält man dann durch Pixel-weise Addition des maskierten und markierten Helligkeitsdifferenzbildes $L^*_{m-,k}$ auf das mit in der Containerdatei enthalten Originalbild Bild B.

5.3.2 Differenz aus 1- und 0-markierter Version

Vorausgesetzt, das Originalbild wird nach dem Shuffling nicht mehr benötigt, so kann eine weitere Optimierung vorgenommen werden. Anstatt die Differenzbilder zum Originalbild zu berechnen, kann ein Differenzbild $F_{\{1\}^l-\{0\}^l,k} = F_{\{1\}^l,k} - F_{\{0\}^l,k}$ aus dem 1- und 0-markierten Frequenzbild

berechnet werden. Da die meisten Koeffizienten dieses Differenzbildes ebenfalls den Wert Null haben, kann hier analog zum Verfahren, das im letzten Abschnitt beschrieben wurde, eine kompaktes Format für die Speicherung des Differenzbildes gewählt werden. Für jeden Koeffizienten ungleich Null werden die x- und y-Position im Frequenzbild, sowie der reale und der imaginäre Anteil des Koeffizienten in der kartesischen Form benötigt. Außerdem wird ein Mapping auf 2 Byte große, ganzzahlige Werte (*Short*) vorgenommen werden.

Zusammen mit diesem Differenzbild im Frequenzraum wird dann, anstatt des Originalbildes, eine mit dem Template und $\{0\}^l$ markierte Version des gesamten Bildes im Container gespeichert. Um auch hier Speicherplatz zu sparen wird dieses Bild mit einem durch den Benutzer zu wählenden Kompressionsfaktor q JPEG-komprimiert.

Für die Werte der Maske M und die β-Werte wird auch hier analog zum Verfahren im letzten Abschnitt ein Mapping und eine ZIP-Kompression vorgenommen.

Die Container-Datei enthält somit folgende Komponenten:

- Mit Template und $\{0\}^l$ markierte Version des Bildes $B^*_{\{0\}^l,k}$ (JPEG komprimiert)
- Maske M (ZIP komprimierte, auf ganzzahlige Werte abgebildete Fließkommawerte)
- Differenzbild im Frequenzraum aus der mit $\{1\}^l$ und $\{0\}^l$ markierten Version $F_{\{1\}^l-\{0\}^l,k}$ (Position, Real- und Imaginärteil der Werte ungleich 0)
- Positionen der zu den Nachrichten-Bits gehörenden Koeffizienten $g(k)$ (x- und y- Position)

Beim Shuffling wird dann aus dem Differenzbild im Frequenzraum $F_{\{1\}^l-\{0\}^l,k}$ durch Eliminierung der Koeffizienten, die nicht zu Nachrichten-Bits mit dem Wert 1 gehören, ein Differenzbild $F_{m-\{0\}^l,k}$ erzeugt. Dieses Differenzbild im Fourierraum enthält nun nur noch die Differenz zwischen der einzubettenden Wasserzeichennachricht m und dem mit $\{0\}^l$ markierten Bild. Nach der inversen Fourier-Transformation erhält man ein markiertes Helligkeitsdifferenzbild $L_{m-\{0\}^l,k}$, das noch blockweise zu maskieren ist. Um die visuelle Maskierung vornehmen zu können, werden die Maskenwerte M und die β-Werte (ein Wert pro Block) benötigt. Dann wird die Maskierung, wie im vorigen Kapitel beschrieben, vorgenommen. So erhält man aus $L_{m-\{0\}^l,k}$ ein markiertes und maskiertes Helligkeitsdifferenzbild $L^*_{m-\{0\}^l,k}$.

Das mit der Nachricht m markierte Bild $B^*_{m,k}$ erhält man dann durch Pixel-weise Addition des maskierten und markierten Helligkeitsdifferenzbildes $L^*_{m-\{0\}^l,k}$ auf das mit $\{0\}^l$ markierte Bild $B^*_{\{0\}^l,k}$.

Die Abbildungen 5.3 und 5.4 enthalten eine Übersicht des Containerings und des Shufflings für das Verfahren in der Fourierdomäne (aus Gründen der Übersichtlichkeit ohne Template-Einbettung). Die rot markierten Elemente sind in der Container-Datei enthalten und beschreiben somit die Grenze zwischen dem Containering und dem Shuffling.

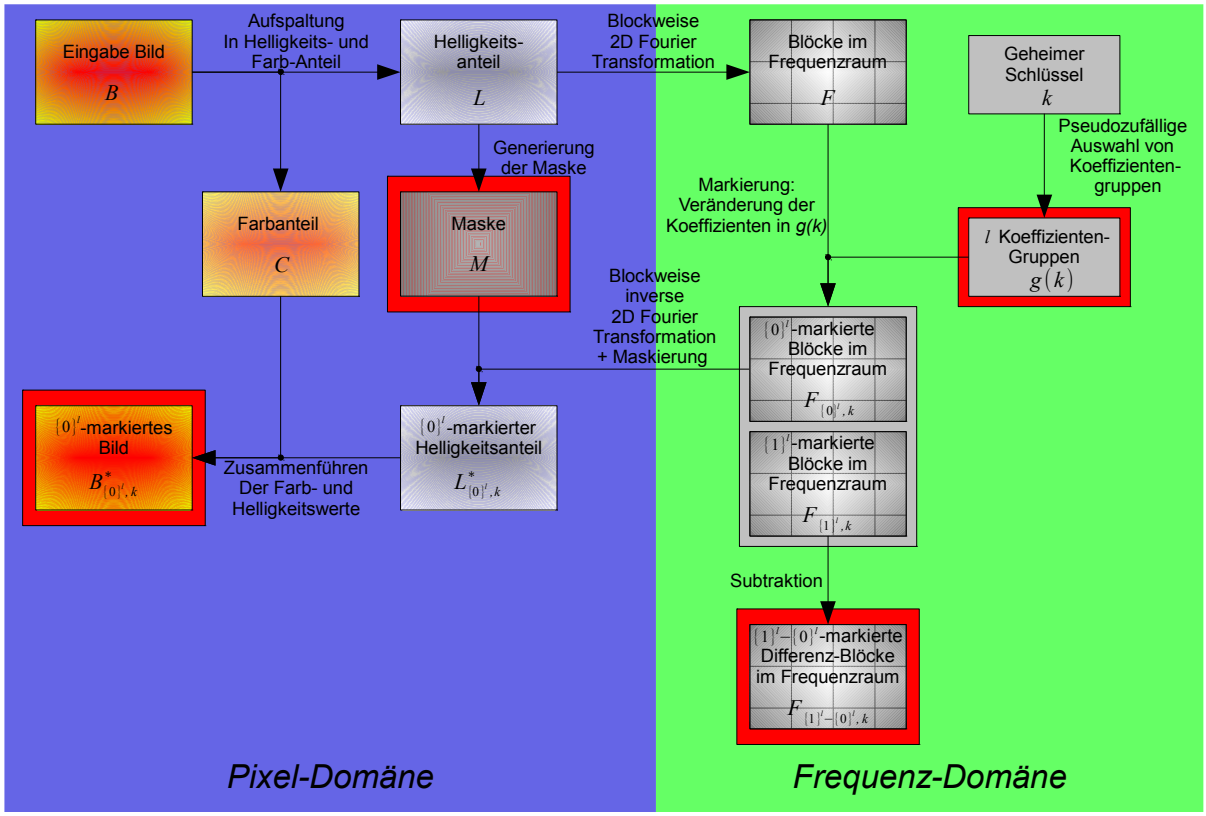

Abbildung 5.3.: Containering-Prozess des ImageMark Bildwasserzeichens in der Fourierdomäne (rot markierte Elemente werden in der Container-Datei gespeichert)

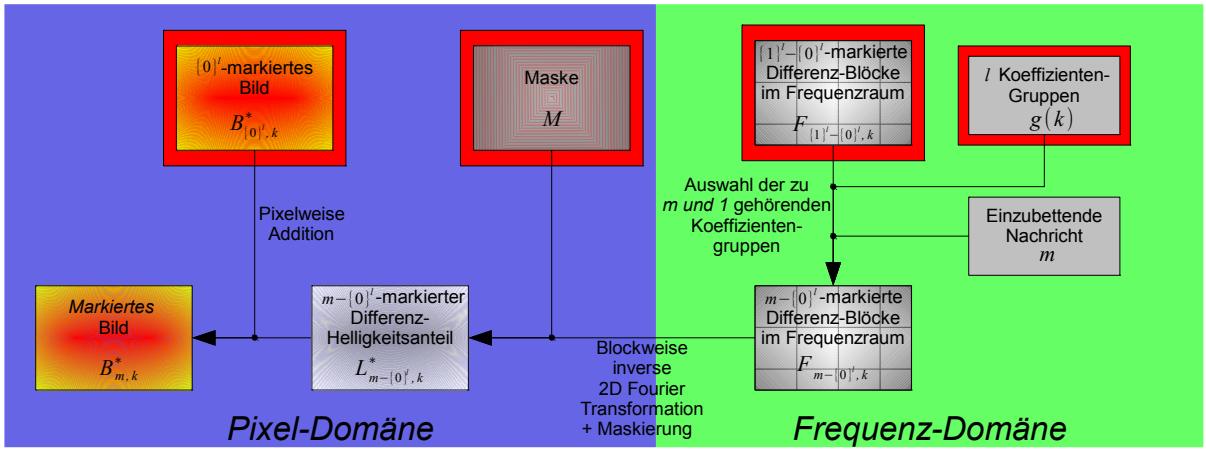

Abbildung 5.4.: Shuffling-Prozess des ImageMark Bildwasserzeichens in der Fourierdomäne (rot markierte Elemente sind in der Container-Datei enthalten)

6 Technische Umsetzung

In diesem Kapitel wird auf die technische Umsetzung der beiden Konzeptionen des ImageMark Containers eingegangen, die im vorigen Kapitel beschrieben wurden.

6.1 Verwendete Programmiersprachen und Frameworks

Die Programmierung für diese Arbeit besteht aus zwei voneinander unabhängigen Teilen. Der erste Teil beinhaltet das Containering, also die Erzeugung einer Container-Datei aus einem Eingabebild. Der zweite Teil besteht aus dem Shuffling. Dabei wird aus einer Container-Datei ein individuell markiertes Bild extrahiert. Offensichtlich müssen diese beiden Module somit nicht direkt miteinander interagieren.

Das Containering stellt eine Erweiterung des existierenden ImageMark Frameworks dar. Da das ImageMark Projekt in C++ implementiert ist, wurde das Containering ebenfalls in dieser Programmiersprache implementiert. Als Entwicklungsumgebung wurde Microsoft Visual Studio 2008 verwendet. Das Shuffling wurde in Java implementiert. Als Entwicklungsumgebung wurde dabei Eclipse verwendet.

6.2 Verwendung des SITMark MultiContainer Frameworks

In diesem Abschnitt wird die Verwendung des SITMark MultiContainer Frameworks beschrieben, welches am Fraunhofer Institut für Sichere Informationstechnologie entwickelt wurde. Das Framework ermöglicht eine unkomplizierte und vereinheitlichte Implementierung des Containerings und des Shufflings für beliebige Wasserzeichenverfahren. Es beinhaltet die Definition eines Container Dateiformats und stellt mit der Klasse `MultiContainer` (C++) Methoden zur Erzeugung von Packet-Headern, die dieser Spezifiktaion entsprechen, bereit. Das Programm SITMark Shuffler kann zum Shuffling von individuell markierten Werken aus Containerdateien verwendet werden, die diesen Spezifikationen entsprechen.

6.2.1 Verwendung der MultiContainer-Klasse bei der Container Erzeugung

Die Klasse `MultiContainer` (C++) des SITMark MultiContainer Frameworks beinhaltet Methoden zum Schreiben von Paket-Headern im SITMark MultiContainer-Format. Beim Erzeugen eines `MultiContainer` Objekts wird ein Ausgabestrom übergeben, der dann durch Aufruf von `writeData()` und den einzelnen Header-Erzeugungsmethoden mit Daten gefüllt wird:

- `MultiContainer(ostream* osOutputStream);`

Die `writeData`-Methode kann verwendet werden, um beliebige Datenpakete in die Multicontainer Datei zu schreiben:

- `void writeData(char* pcAData, unsigned int uiDataLength);`

Die verwendeten Header-Erzeugungsmethoden sind folgende:

- `void writeContainerHeader();`

  Schreibt den globalen MultiContainer-Header.

- ```
  void writeStreamHeaderMarker(unsigned char  ucIsMultiplex,
                               unsigned char  ucNumberOfStreams,
                               unsigned char* pucAStreamIds,
                               unsigned char* pucAStreamTypes,
                               char**         ppcAStreamNames,
                               unsigned int*  puiMsgLengths,
                               unsigned int*  puiCrcModes,
                               unsigned int*  puiForwardErrorCorrections,
                               unsigned int*  puiMulBits);
  ```

  Diese Methode erzeugt den Streamlevel-Header, welcher Informationen über alle im Container enthaltenen Streams und die besonderen Eigenschaften bei der Markierung dieser Streams enthält. Beim ImageMark Container handelt es sich dabei jeweils nur um einen einzelnen Stream. Über den Typ eines Streams (`pucAStreamTypes`) werden die beiden entwickelten Versionen des ImageMark Containers unterschieden.

- ```
  void writeEUPHeader(unsigned char      ucStreamId,
                      unsigned int       uiEuId,
                      unsigned int       uiEuByteLength,
                      unsigned char      ucMessagePart,
                      unsigned char      ucComputationType,
                      unsigned long long ullOutputBytePos);
  ```

  Diese Methode wird dazu verwendet die Header der elementaren Einheiten (*Elementary Units*) zu erzeugen. Beim Verfahren im Pixelraum sind das die Differenzbilder $L^*_{\{1\}_i-\{0\}_i,k}$. Beim Verfahren im Fourierraum handelt es sich um die Positionen der Koeffizienten $g(k)$, die zu den jeweiligen Nachrichten-Bits gehören.

  Außerdem werden elementare Einheiten erzeugt, die die übrigen im Daten der Containerdatei enthält. Bei beiden Verfahren handelt es sich dabei um das mit $\{0\}^l$ markierte Bild $B^*_{\{0\}^l,k}$. Beim Verfahren im Fourierraum kommen noch die Maske M und die Koeffizienten des Differenz-Frequenzbildes $F_{\{1\}^l-\{0\}^l,k}$ hinzu (vgl. Kapitel 5.2 und Kapitel 5.3).

Elementare Einheiten, die Differenzsignale enthalten werden über den `ucComputationType` Parameter identifiziert. Die elementaren Einheiten, die zur Initialisierung dienen und keine Daten einzelner Nachrichten-Bits enthalten, werden von den elementaren Einheiten, die Daten einzelner Nachrichten-Bits enthalten, über den `ucMessagePart` Parameter unterschieden.

6.2.2 Verwendung des SITMark Shufflers

Das Paket `de.fraunhofer.sit.watermarking.multicontainer.shuffler` enthält den SITMark Shuffler. Dieses Programm kann dazu verwendet werden, aus Containerdateien, die der SITMark MultiContainer Spezifikation entsprechen, individuell markierte Medien zu extrahieren.

Bei den in dieser Arbeit entwickelten Container-Verfahren werden im Vergleich zu anderen Container-Verfahren beim Shuffling nicht nur Lese- und Schreibeoperationen, sondern zusätzlich aufwendige Rechenoperationen durchgeführt (vgl. Kapitel 5). Um mit dem SITMark Shuffler Containerdateien verarbeiten zu können, die mit ImageMark erzeugt wurden, ist es daher notwendig, dass dieses Shuffling Programm um die Verarbeitungsschritte der beiden entwickelten Verfahren erweitert wird.

Das Interface DifferenceSignalHandler kann zum Verarbeiten von Differenz-Signalen verwendet werden. Beiden entwickelten Verfahren ist gemeinsam, dass deren elementare Einheiten Differenzsignale enthalten. Durch die Implementierung dieses Interfaces kann somit die Einbindung in den Shuffler erfolgen.

Im Interface DifferenceSignalHandler sind lediglich zwei Methoden definiert:

- ```
  ByteBuffer initialize(ByteBuffer initData,
                        String     embeddedMessage,
                        int        bitsPerDiffImage);
  ```

- `ByteBuffer addDifference(ByteBuffer difference)`

Die Methode `initialize()` liefert Daten, die zur Initialisierung des `DifferenceSignalHandlers` benötigt werden. Die Methode `addDifference()` liefert dagegen die Daten der elemenaren Einheiten, die Informationen über einzelne Nachrichten-Bits enthalten. Falls durch die übergebenen Daten Teile des Ausgabemediums produziert werden können, gibt die Methode diese produzierten Daten zurück.

Es müssen zwei `DifferenceSignalHandler` Klassen implementiert werden. Eine Klasse, die das Container-Verfahren im Pixelraum abbildet und eine Klasse für das Verfahren im Fourierraum.
  ```

DifferenceSignalHandlerPixel

Die Klasse `DifferenceSignalHandlerPixel` ist die Implementierung des Interfaces `DifferenceSignalHandler` für das Container-Verfahren im Pixelraum. Bei diesem Verfahren erhält die `initialize()` Methode als Parameter ausschließlich das mit $\{0\}^l$ markierte Bild $B^*_{\{0\}^l,k}$. Dieses liegt als JPEG-komprimiertes Bild vor und muss somit zunächst dekomprimiert werden. Dazu wird die statische Methode `read(InputStream input)` der Klasse `javax.imageio.ImageIO` verwendet. Normalerweise wird von der `initialize()` Methode `null` zurückgegeben, da nach der Initialisierung noch keine markierten Bilddaten existieren. Falls die einzubettenden Nachricht allerdings keine 1-Bits enthält, so werden die Daten des $\{0\}^l$-markierten Bildes zurückgegeben, ohne diese zu dekomprimieren.

Die `addDifference()` Methode erhält als Parameter pro Nachrichten-Bit aus m mit dem Wert 1 ein Differenz-Helligkeitsbild $L^*_{\{1\}_i-\{0\}_i,k}$. Diese Differenzbilder liegen jeweils ZIP komprimiert vor und müssen zunächst dekomprimiert werden. Dazu wird die Klasse `java.util.zip.ZipInputStream` verwendet. Nach der Dekomprimierung müssen die Differenzbilder Pixel-weise eingelesen werden, wobei für jeden Pixel eine Dekodierung durchgeführt werden muss. Die Werte müssen von ganzzahligen Werten aus dem Intervall $[0, 255]$ auf Fließkomma-Werte aus dem Intervall $[min, min + range]$ abgebildet werden, wobei min und $range$ zusammen mit dem Differenzbild aus der Container-Datei ausgelesen werden. Um an Geschwindigkeit zu gewinnen wird jede dieser Dekodierungen in einen eigenen Thread ausgelagert.

Da durch die Markierung nicht alle Pixelwerte verändert werden, ist es effizienter zunächst alle Differenzbilder aufzusummieren und daraufhin die Summe aller Differenzbilder einmalig auf das mit $\{0\}^l$ markierte Bild zu summieren. Die Addition von zwei Differenzbildern nimmt sehr viel Zeit in Anspruch, da im schlechtesten Fall eine Addition pro Pixel des Bildes vorgenommen werden muss. Anstatt die Additionen nacheinander durchzuführen, macht es an dieser Stelle ebenfalls Sinn, die Differenzbildaddition in einzelne Threads auszulagern. Sobald zwei Differenzbilder dekomprimiert vorliegen, wird eine Addition initiiert. So wird die Anzahl der Differenzbilder zunächst halbiert. Im nächsten Schritt wird mit den paarweise summierten Differenzbildern die selbe Operation durchgeführt. Die Addition aller Differenzbilder erfolgt somit stufenweise in Form einer Pyramide (vgl. Abbildung 6.1).

Bei dieser Vorgehensweise wird allerdings sehr viel Speicherplatz benötigt, da alle Differenzbilder auf der untersten Stufe als Float-Arrays dargestellt werden müssen. Somit werden alleine für die Differenzbilder $w \cdot h \cdot l_1 \cdot 4$ Byte an Speicherplatz benötigt (w, h: Breite, Höhe des Bildes, l_1: Anzahl der 1-Bits in der Wasserzeichennachricht, 4: Anzahl der Bytes pro `Float` Wert). Daher wird an dieser Stelle überprüft, ob der *Heapspace* der *Java Virtual Machine* (JVM) genügend Speicherplatz zur Verfügung hat, um alle `Float-Arrays` verwalten zu können. Ist dies nicht der Fall, so wird keine parallele Addition in Pyramidenform durchgeführt. Stattdessen wird das Par-

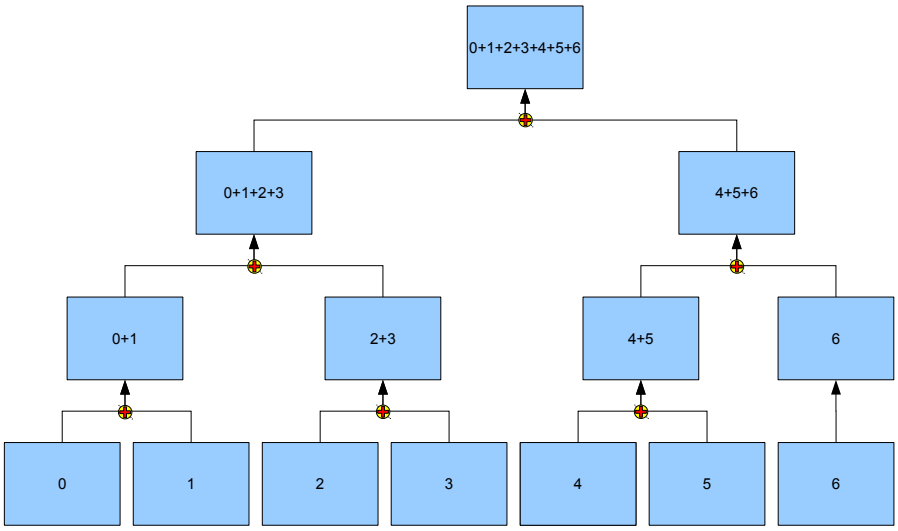

Abbildung 6.1.: Stufenweise Addition von sieben Differenzbildern in Pyramidenform

sen der Differenzbilder im Haupt-Thread durchgeführt und die ausgelesenen Pixelwerte werden direkt auf einen globalen Differenzbild-`Array` aufaddiert.

Die `addDifference()` Methode erwartet als Rückgabewert einen `ByteBuffer`. Erst wenn das letzte Differenzbild vorliegt, kann das markierte Medium erzeugt werden. Somit wird von allen `addDifference()`-Aufrufen, außer dem Letzten, `null` zurück gegeben. Wenn das letzte Differenzbild übergeben wird und dekomprimiert vorliegt, wird mittels `wait()` darauf gewartet, dass alle Addition abgeschlossen sind. Sobald der letzte Additions-Thread terminiert, wird mittels `notify()` signalisiert, dass das Differenzbild $L^*_{m-\{0\}^l,k}$ nun vorliegt. Dieses Differenzbild kann dann auf das eingelesene Bild Pixel-weise aufaddiert werden. Sobald diese Addition abgeschlossen ist, kann das Bild unter Verwendung der statischen Methode `write(RenderedImage im, String formatName, OutputStream output)` der Klasse `javax.imageio.ImageIO` zunächst in einen `java.io.ByteArrayOutputStream` geschrieben werden. Aus diesem wird dann ein `ByteBuffer` erzeugt, der dann von der `addDifference()` zurückgegeben wird.

DifferenceSignalHandlerFrequency

Die Klasse `DifferenceSignalHandlerFrequency` ist die Implementierung des Interfaces `DifferenceSignalHandler` für das Container-Verfahren im Fourierraum. Bei diesem Verfahren liefert die `initialize()` Methode das mit $\{0\}^l$ markierte Bild $B^*_{\{0\}^l,k}$, sowie die Koeffizienten des Differenz-Frequenzbildes $F_{\{1\}^l-\{0\}^l,k}$, die Maske M und die zur Maske gehörenden β-Werte (vgl. Kapitel 6.2.1).

Das $\{0\}^l$-markierte Bild wird hier analog zum Vorgehen in der Klasse `DifferenceSignalHandlerPixel` mittels `javax.imageio.ImageIO` ausgelesen. Die Maskenwerte liegen ZIP komprimiert vor und müssen zunächst dekomprimiert werden. Dazu wird auch hier die Klasse `java.util.zip.ZipInputStream` verwendet. Nach der Dekomprimierung müssen die Maskenwerte dekodiert werden. Zunächst muss jeder Byte in zwei 4 Bit Werte unterteilt werden. Diese müssen dann von ganzzahligen Werten aus dem Intervall $[0, 15]$ unter Einbeziehung des Verstärkungsfaktors μ auf Fließkomma-Werte aus dem Intevall $[0, 1]$ abgebildet werden. Die Koeffizienten des Differenz-Frequenzbildes $F_{\{1\}^l-\{0\}^l,k}$ liegen ebenfalls komprimiert vor. Pro Koeffizient werden zunächst vier *Short*-Werte ausgelesen. Die ersten beiden Werte sind die x-, und y-Position im zweidimensionalen Frequenzbild. Die anderen beiden Werte sind der Real- und Imaginärteil des Koeffizienten. Diese müssen von ganzzahligen Werten aus dem Intervall $[-32768, 32767]$ auf Fließkomma-Werte aus dem Intevall $[min, min + range]$ abgebildet werden, wobei min und $range$ zusammen mit dem Differenzbild aus der Container-Datei ausgelesen werden. Die β-Werte der Maske werden als *Integer* Werte ausgelesen und müssen durch den mit den Maskenwerten in der Containerdatei abgelegten Streckfaktor dividiert werden. Um an Geschwindigkeit zu gewinnen wird jeder dieser Ausleseprozesse in einen eigenen Thread ausgelagert.

Die `addDifference()` Methode liefert pro Nachrichten-Bit aus m mit dem Wert 1 die Positionen $g(k)_i$ der Koeffizienten innerhalb eines Bildblocks, die zu dem Nachrichten-Bit gehören. Für jeden Bildblock im Frequenzbild werden nun die Koeffizienten aus $F_{\{1\}^l-\{0\}^l,k}$ an den Position $g(k)_i$ in ein zu Beginn leeres Differenzbild-Array kopiert. Dieses Differenzbild-Array enthält somit nach dem letzten Aufruf von `addDifference()` das Frequenzbild $F_{m-\{0\}^l,k}$.

Sobald dieses Differenzbild im Frequenzraum vorliegt, wird blockweise eine inverse Fourier-Transformation durchgeführt. Um nicht unnötig Performanz zu verlieren, wird dazu die Bibliothek FFTW verwendet (vgl. Kapitel 6.3).

Nach der inversen Fourier-Transformation wird jeder Block des Differenz-Helligkeitsbildes $L_{m-\{0\}^l,k}$ maskiert. Dazu wird das in Kapitel 4.1.1 beschriebene Verfahren verwendet. Der einzige Unterschied zu dem beschriebenen Verfahren besteht darin, dass hier der β-Wert eines Blockes nicht berechnet werden muss, sondern schon vorliegt. Die blockweise Maskierung wird ebenfalls in eigene Threads ausgelagert.

Nachdem alle Blöcke des markierten und maskierten Differenz-Helligkeitsbildes $L^*_{m-\{0\}^l,k}$ vorliegen, kann hier analog zum Vorgehen in der Klasse `DifferenceSignalHandlerPixel` das Ausgabebild mittels Pixel-weiser Addition und unter Verwendung der `write()` Methode der Klasse `javax.imageio.ImageIO` erzeugt werden.

6.3 Verwendung von FFTW

Beim Shuffling des Container-Verfahrens im Fourierraum muss eine inverse zweidimensionale Fouriertransformation durchgeführt werden. Da diese Operation sehr zeitaufwendig ist, wird hierfür eine effiziente Implementierung benötigt.

FFTW, „The Fastet Fourier Transform in the West“ (vgl. [7]), ist eine Bibliothek, die Funktionalitäten zur Berechnung von Fourier Transformationen bereitstellt. FFTW ist eine Implementierung des Konzepts der *Schnellen Fourier Transformation*, kurz FFT, die erstmal 1965 in [8] von Cooley und Turkey veröffentlicht wurde. Mittlerweile wird die erste Verwendung der FFT allerdings Carl Friedrich Gauß zugeordnet, der mit dem Algorithmus im Jahr 1805 die Flugbahn von Asteroiden bestimmte (vgl. [10]).

Bei FFTW handelt es sich um eines der schnellsten derzeit existierenden Programme zur Berechnung von Fourier Transformationen[1]. Somit ist die Verwendung von FFTW für die Durchführung der inversen zweidimensionalen Fouriertransformation beim Shuffling eine sinnvolle Wahl.

Da der SITMarkShuffler in Java und FFTW in C implementiert ist, muss an dieser Stelle über das Java Native Interface (JNI) eine Anbindung für FFTW in Java definiert werden. Es existieren einige Projekte, die diese Anbindung realisieren. In dieser Arbeit wurde dafür jFFTW[2] verwendet.

Um die inverse Fouriertransformation über jFFTW aufzurufen wird zunächst eine Instanz der Klasse `FFTWReal` erzeugt. An diesem Objekt kann dann die Methode `twoDimensionalBackward()` aufgerufen werden. Diese Operation führt die zweidimensionale inverse Fouriertransformation mittels FFTW durch und gibt die transformierten Daten zurück.

Um für den Performanzvergleich des ImageMark Einbettungsprozesses und des Shufflings der beiden entwickelten Container-Verfahren (vgl. Kapitel 7.2.2) gleiche Voraussetzungen zu schaffen und um sicher zu stellen, dass der gesamte Geschwindigkeitsgewinn nicht ausschließlich auf die Verwendung der FFTW-Bibliothek zurück zu führen ist, wurde FFTW nicht nur in den SITMarkShuffler, sondern auch in das ImageMark Projekt integriert. Das ImageMark Projekt beinhaltet eine Klasse `CFFT`, die Vorwärts- und Rückwärts-Fouriertransformationen berechnet. In diese Klasse wurde die Verwendung von FFTW integriert.

Im Abschnitt 7.2.3 wird beschrieben, wie sich die Performanz der ImageMark Einbettung und Detektion durch die Verwendung von FFTW verändert.

[1] vgl. `www.fftw.org`, Zugriff: 25.10.2011

[2] vgl. `www2.ph.ed.ac.uk/~wjh/teaching/Java/fft`, Zugriff: 25.10.2011

7 Evaluierung

In diesem Kapitel werden Speicherbedarf, Performanz, Robustheit und Bildqualität der entwickelten Container-Verfahren untersucht. Dabei werden die Unterschiede zwischen den beiden Verfahren, sowie die Vor- und Nachteile gegenüber dem ImageMark Einbettungsprozess herausgearbeitet. Des Weiteren werden einige der implementierten Optimierungen bewertet. Bei den genannten Tests wird das in Kapitel 5.3.2 beschriebene Verfahren als Verfahren im Frequenzraum verwendet. Containergröße und Shuffling-Performanz dieses Verfahrens unterscheiden sich nur geringfügig von dem in Kapitel 5.3.1 beschriebenen Verfahren. Als Verfahren im Pixelraum wird das in Kapitel 5.2.2 beschriebene Verfahren verwendet. Das Verfahren aus Kapitel 5.2.1 zeichnet sich durch größere Containerdateien bei ähnlicher Shuffling-Performanz aus.

Tabelle 7.1 zeigt die Parametrisierung von ImageMark für die in diesem Kapitel durchgeführten Untersuchungen.

Parameter	Wert	Kurzbeschreibung
N	1024	Seitenlänge der Blöcke
u	12	Eingebettete Nachrichten-Bits Pro Region eines Frequenzbild-Blockes
l	$48 = 4 \cdot u$	Gesamtlänge der eingebetteten Nachricht
n	100	Anzahl der manipulierten Fourier-Koeffizienten innerhalb eines Blockes pro Nachrichten-Bit
t	14	Anzahl der manipulierten Fourier-Koeffizienten innerhalb eines Blockes für die Template-Einbettung
s	2	Einbettungsstärke
q	90%	JPEG Kompressionsfaktor

Tabelle 7.1.: Verwendete Parametrisierung von ImageMark

Grundlage für die Ergebnisse, die in diesem Kapitel dargestellt werden, ist eine Testumgebung mit 100 Bildern. Tabelle 7.2 zeigt die Eigenschaften dieser Bilddateien. Bei allen Bildern handelt es sich um JPEG-Farbbilder mit einer Farbtiefe von 24 Bit pro Pixel. Die Originalbilder sind JPEG-Dateien mit verschiedenen Qualitätsstufen. Eine Abbildung aller verwendeten Bilder ist im Anhang zu finden (Abbildung A.1 und Abbildung A.1).

Größen-kategorie	Abmessungen	Megapixel	Anzahl
1	1024×768	$0,75$	12
2	1600×1200	$1,83$	33
3	2048×1536	$3,00$	11
4	2448×1836	$4,29$	10
5	3264×2448	$7,62$	21
6	3472×2604	$8,62$	10
7	4288×3216	$13,15$	3

Tabelle 7.2.: Verwendete Bilddateien und deren Größe

7.1 Speicherbedarf

Abgesehen von kürzlichen Unregelmäßigkeiten[1], werden die Kosten für Festplattenspeicher tendenziell immer geringer. Um ein Verfahren zu entwickeln, was für den Anwender interessant ist, ist es dennoch wichtig den Speicherbedarf der Containerdateien möglichst gering zu halten. Gerade für Szenarien in denen große Mengen von Containerdateien angelegt werden, wie beispielsweise bei einem Onlineshop, spielen die Speicherkosten eine wichtige Rolle. Besonders um mit dem normalen ImageMark Einbettungsverfahren konkurrieren zu können, sollte die Containerdateigröße eines Bildes nicht bedeutend größer sein als der Speicherplatz den ein JPEG-Bild benötigt.

Tabelle 7.3 zeigt den Vergleich der Containergrößen für die beiden entwickelten Verfahren an Hand des Vergrößerungsfaktors im Vergleich zu den Originalbildern. Dieser Vergleich wird zu den Bildern im unkomprimierten BMP-Format, sowie im komprimierten JPEG-Format gezogen. Dabei ist zu beachten, dass bei einem Vergleich mit JPEG-komprimierten Originalbildern der Vergrößerungsfaktor stark von der Kompressionsfähigkeit des Originalbildes beeinflusst wird. Die Tabelle zeigt den Vergleich nicht Bild für Bild. Für jede der sieben Bild-Kategorien wird ein Mittelwert des Größenfaktors berechnet.

Der Vergleich der Dateigrößen zeigt, dass die Containerdateien des Verfahrens im Pixelraum sehr viel größer sind als die des Container-Verfahrens im Fourierraum. Mit durchschnittlichen Vergrößerungsfaktoren von 392% wird für diese Containerdateien enorm viel Speicherplatz benötigt. Im Vergleich zur Einbettung ohne Container-Verfahren ist dabei zu bedenken, dass nicht nur unkomprimierte BMP-Dateien, sondern auch komprimierte Dateiformate, wie JPEG, als Eingabe für die Einbettung mittels ImageMark verwendet werden können. Vergleicht man beispielsweise den Speicherbedarf eines JPEG-komprimierten Bildes mit dem Speicherplatz, den

[1] vgl. www.zdnet.de/magazin/41558239/festplatten-steigende-preise-und-heftige-machtkaempfe.htm, Zugriff: 10.12.2011

Größen-kategorie	Dateigröße der Originalbilder in MB		Größenfaktor Containerdatei Fourierdomäne		Größenfaktor Containerdatei Pixeldomäne	
	JPEG	BMP	vs. JPEG	vs. BMP	vs. JPEG	vs. BMP
1	0,24	2,25	170%	18%	4099%	421%
2	0,50	5,49	201%	18%	4864%	431%
3	0,71	9,00	189%	15%	4786%	374%
4	1,11	12,86	185%	16%	4852%	417%
5	1,75	22,86	188%	14%	4673%	347%
6	1,49	25,87	209%	12%	6140%	347%
7	1,75	39,45	232%	10%	6859%	300%
1 - 7	0,95	12,93	193%	16%	4910%	392%

Tabelle 7.3.: Speicherbedarf der Containerdateien (Mittelwerte aller Bilder der jeweiligen Größenkategorie)

eine Containerdatei des Verfahrens im Pixelraum benötigt, so werden Vergrößerungsfaktoren von bis zu 7700% erreicht.

Die Containerdateien des Verfahrens im Fourierraum benötigen dagegen sehr viel weniger Speicherplatz als die Originalbilder im BMP Format. Vergleicht man den Speicherbedarf der Containerdateien dieses Verfahrens mit dem Speicherbedarf der Originalbilder im JPEG Format, so werden Vergrößerungsfaktoren von 150% bis zu 270% erreicht. Das Verfahren im Fourierraum kommt somit mit wesentlich weniger Speicherplatz aus als das Verfahren im Pixelraum.

Betrachtet man die Unterschiede zwischen den Bild-Kategorien, so fällt auf, dass die Vergrößerungsfaktoren der Containerdateien vergleichen mit den Original-BMP Dateien bei beiden Verfahren mit steigender Bildgröße tendenziell abnimmt. Dies ist darauf zurückzuführen, dass sowohl JPEG-Kompression, als auch die bei beiden Verfahren verwendete ZIP-Kompression bei größeren Datensätzen bessere Kompressionsraten erreicht.

Auffällig ist außerdem, dass sich die Größen der Containerdateien von Bildern aus der gleichen Größenkategorie mitunter sehr stark voneinander unterscheiden. Bei den Vergleichswerten gegenüber den JPEG-komprimierten Originalbildern ist zudem keine eindeutige Tendenz erkennbar. Diese Tatsache ist auf die unterschiedliche Beschaffenheit der Bilddateien zurückzuführen. Bilder mit glatten Flächen können durch das JPEG Verfahren stark komprimiert werden. Bilder mit vielen Details und wenigen glatten Flächen können dagegen weniger von der JPEG Komprimierung profitieren. Da die Containerdateien beider Verfahren eine mit $\{0\}^l$ markierte, JPEG-komprimierte Version des Bildes enthalten, wirkt sich dies somit direkt auf die Containerdateigröße aus.

Hinzu kommt, dass glatte Flächen auf Grund der visuellen Maskierung durch die Wasserzeichenmarkierung kaum verändert werden. Die Differenzbilder beim Container-Verfahren im Pixelraum haben an diesen Stellen somit viele benachbarte Pixel mit dem Wert 0. Da die Differenzbilder ZIP-komprimiert werden, verringert dies zusätzlich die Dateigröße.

Beim Verfahren im Fourierraum ist die Maske, die zur visuellen Maskierung benötigt wird, in der Containerdatei enthalten. Diese enthält an Stellen mit glatten Flächen ebenfalls viele 0-Werte. Da die Masken ebenfalls ZIP-komprimiert werden, schlägt sich diese Beschaffenheit auch beim Verfahren im Fourierraum in einer geringen Dateigröße nieder.

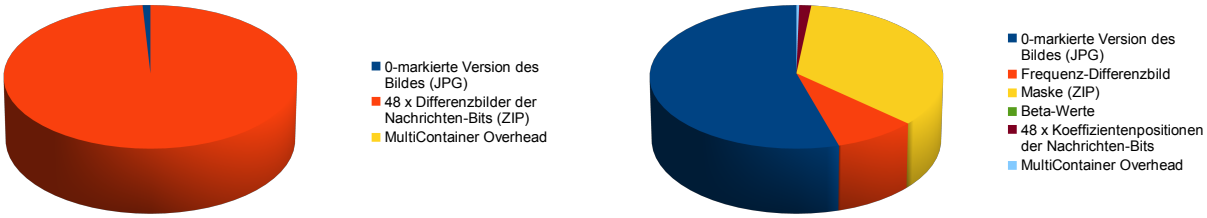

Abbildung 7.1.: Zusammensetzung der Containerdateien (links Container-Verfahren im Pixelraum, rechts Container-Verfahren im Frequenzraum)

Abbildung 7.1 zeigt die durchschnittliche Zusammensetzung von Containerdateien der beiden entwickelten Verfahren. Der Grund dafür, dass die Containerdateien für das Verfahren im Pixelraum sehr viel größer sind, als die des Verfahrens im Frequenzraum, wird hier deutlich. Die blau markierten Bereiche im Kreisdiagramm beschreiben jeweils das mit $\{0\}^l$ markierte, JPEG-komprimierte Bild, welches bei beiden Verfahren in der Containerdatei enthalten ist. Dieser Teil der Containerdatei ist für beide Verfahren identisch. Beim Verfahren im Pixelraum nehmen diese Daten allerdings nur einen sehr kleinen Teil des gesamten Speicherbedarfs in Anspruch. Der Großteil wird für die ZIP-komprimierten Differenzbilder benötigt. Pro Nachrichten-Bit wird dabei ein Differenzbild mit einem zwei Byte großen Wert pro Pixel benötigt. Diese Differenzbilder sind für die enormen Dateigrößen verantwortlich. Da die Differenzbilder allerdings recht hohe Genauigkeit in der Darstellung der Differenzpixel benötigen und an dieser Stelle schon eine ZIP-Komprimierung vorgenommen wird, ist eine starke Verkleinerung dieses Anteils der Containerdatei nicht möglich.

Beim Verfahren im Frequenzraum wird dagegen für die Daten der Nachrichten-Bits nur ein sehr geringer Anteil des Speicherplatzes benötigt. Der meiste Speicherplatz wird hier für das mit $\{0\}^l$ markierte, JPEG-komprimierte Bild und die ZIP-komprimierten Maskenwerte benötigt. Diese beiden Datensätze sind allerdings schon sehr stark komprimiert, sodass an dieser Stelle keine starke Verkleinerung des Speicheraufkommens mehr möglich ist.

Die Containerdateien des Verfahrens im Fourierraum benötigen sehr viel weniger Speicherplatz als die des Verfahrens im Pixelraum. Somit ist das Verfahren im Fourierraum im Bezug auf den

benötigten Speicherplatz zu bevorzugen. Dennoch benötigen auch diese Containerdateien bis zu $2,5$ mal mehr Speicherplatz als die JPEG-komprimierte Bilddaten eines Originalbildes.

7.2 Geschwindigkeit

In diesem Kapitel wird die Performanz des Containerings und des Shufflings der entwickelten Verfahren untersucht. Einige der Tests wurden auf mehreren Systemen durchgeführt, um beurteilen zu können, wie sich die Leistung des Einbettungsprozesses und des Shufflings in verschiedenen Umgebungen verändert. Die verwendete Hardware und die darauf laufenden Betriebssysteme sind in Tabelle 7.4 dargestellt.

	Architektur	**Prozessoren**	**Hauptspeicher**	**Betriebssystem**
1	Intel Pentium 4 HT 3.0	1 × 3,0 GHz	1,5 GB (SDRAM)	Ubuntu 11.04 Kernel: 2.6.38-12-generic-pae (32 Bit)
2	Intel Mobile Core 2 Duo T7200	2 × 2,0 GHz	2,0 GB (DDR2)	Windows 7 Professional Service Pack 1 (64 Bit)
3	Intel Core 2 Duo E8400	2 × 3,0 GHz	3,2 GB (DDR2)	Windows XP Professional Service Pack 3 (32 Bit)
4	Intel Core i7-2600	4 × 3,4 GHz	8,0 GB (DDR3)	Ubuntu 11.10 Kernel: 3.0.0-12-generic (64 Bit)

Tabelle 7.4.: Zur Evaluierung verwendete Hardware/Betriebssysteme

7.2.1 Geschwindigkeit des Containerings

Das Containering muss nur ein einziges mal durchgeführt werden, um aus einer Bilddatei mehrere individuell markierte Bilder mit dem selben geheimen Schlüssel erzeugen zu können. Dennoch ist es von Bedeutung den Zeitaufwand dieses Vorgangs zu betrachten.

Die Container-Erzeugung benötigt für das Verfahren im Frequenzraum durchschnittlich 260% der Rechenzeit der normalen Einbettung. Die Spanne der Berechnungsdauer reicht dabei von 225% bis zu 355%. Das Containering für das Verfahren im Pixelraum benötigt durchschnittlich 4355% der normalen Einbettungsdauer. Die Spanne der Berechnungsdauer reicht dabei von 3500% bis zu 5555% der Rechenzeit der normalen Einbettung. Die Testdaten wurden auf System Zwei durch einmalige Erzeugung der Containerdateien für alle 100 Testbilder ermittelt.

Das Containering ist somit beim Verfahren im Frequenzraum um mehr als das zehnfache schneller als beim Verfahren im Pixelraum.

7.2.2 Geschwindigkeit des Shufflings

Die Hauptmotivation dieser Arbeit ist es, den Zeitaufwand der Wasserzeichenmarkierung eines Bildes mit einer individuellen Nachricht durch die Anwendung eines Container-Verfahrens zu verringern. An dieser Stelle wird daher ein Vergleich zwischen der Geschwindigkeit der ImageMark Einbettung und dem Shuffling der beiden entwickelten Verfahren gezogen. Abbildung 7.2 enthält einen relativen Vergleich der Markierungsdauer der beiden entwickelten Verfahren mit dem ImageMark Einbettungsprozess. Das Diagramm zeigt den Faktor um den die Dauer des Shufflings von der Einbettungsdauer abweicht. Dabei handelt es sich jeweils um den Mittelwert der zehnfachen Durchführung für alle 100 Testbilder. Die maximale Größe des Java-Heapspace wurde auf 768 MB festgelegt.

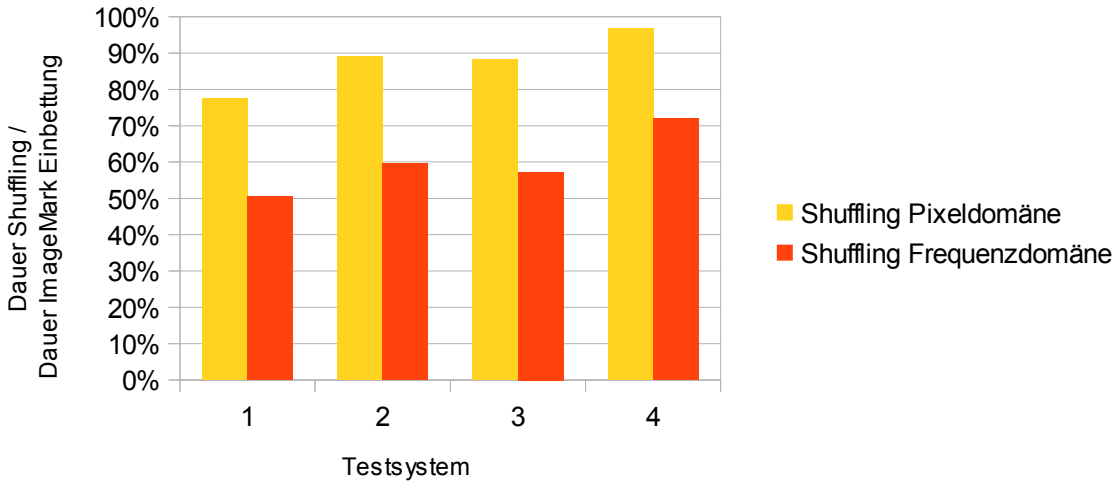

Abbildung 7.2.: Performanz des Shufflings im Vergleich zur normalen Einbettung - Mittelwerte aus zehn Durchführungen für 100 Testbilder verschiedener Größe auf vier Testsystemen

Wie dem Diagramm zu entnehmen ist, wird beim Verfahren im Fourierraum je nach Testsystem durchschnittlich 50% bis 72% der Gesamtdauer der normalen Einbettung benötigt. Beim Verfahren im Pixelraum sind es durchschnittlich 75% bis 97%. Somit ist das Shuffling für beide Verfahren auf allen vier Testsystemen schneller als die normale ImageMark-Einbettung. Die Performanz des Verfahrens im Fourierraum ist allerdings deutlich besser als die des Verfahrens im Pixelraum.

Abbildung 7.3 zeigt ein Diagramm, das zwischen den sieben Bildkategorien differenziert. Die Werte für jedes der 100 Bilder wurden auch hierbei aus dem Mittelwert von zehn Durchläufen berechnet. Die Daten stammen von Testsystem Zwei.

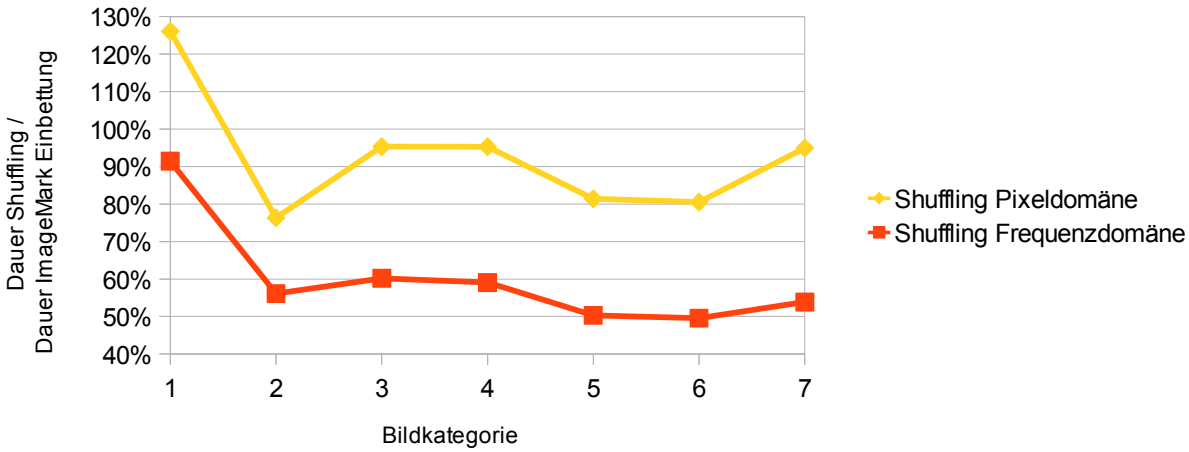

Abbildung 7.3.: Performanzvergleich des Shufflings für verschiedene Bildgrößen - Mittelwert aus zehn Durchführungen für 100 Testbilder verschiedener Größe auf Testsystem Zwei

Beide Container-Verfahren zeigen eine ähnliche Tendenz für die einzelnen Bildkategorien. Für Bildkategorie Eins ist das Shuffling am langsamsten im Vergleich zur ImageMark-Einbettung. für die Bildkategorien Zwei, Fünf und Sechs am schnellsten. Nimmt man Bildkategorie Eins heraus, so wird auf Testsystem Zwei nur zwischen 50% und 60% (Shuffling Fourierdomäne), bzw. zwischen 75% und 95% (Shuffling Pixeldomäne) der Rechenzeit der normalen Einbettung benötigt.

Die Erklärung für den starken Unterschied zwischen Kategorie Eins und den übrigen Kategorien ist in den Eigenschaften der verwendeten Programmiersprache zu finden. Bevor der tatsächliche Shuffling-Prozess angestoßen werden kann, muss zunächst die *Java Virtual Machine* gestartet werden. Bei sehr kurzer Shuffling-Dauer wirkt sich dieser konstante Zeitanteil deutlicher aus, als bei längeren Berechnungszeiten. ImageMark ist in C++ implementiert, also handelt es sich beim Programmaufruf der ImageMark-Einbettung um einen nativen Aufruf. Wird der SITMarkShuffler auf einem System gestartet, bei dem die JVM permanent läuft, so kann der Zeitanteil, der zum Starten der JVM benötigt wird, eliminiert werden.

Von dieser Besonderheit abgesehen ist ansonsten kein Zusammenhang zwischen der Bildgröße und dem Performanzunterschied zwischen herkömmlicher Einbettung und dem Shuffling zu erkennen. Die Schwankungen zwischen den einzelnen Gruppen sind vermutlich auf die Bildeigenschaften zurückzuführen. Enthält ein Bild viele glatte Flächen, so ist die Shuffling-Dauer kürzer. Bei Bildern mit hohem Detailgrad benötigt das Shuffling dagegen sehr viel länger.

Allgemein bewirken glatte Flächen, dass die Maske und somit auch maskierte Bilder wenige Werte ungleich Null enthalten. Beim Verfahren im Pixelraum bedeutet das konkret, dass bei der Erzeugung des Gesamt-Differenzbildes weniger Additionen durchgeführt werden müssen, da die

Differenzbilder viele 0-Werte enthalten. Bilder mit hohem Detailgrad und wenigen glatten Flächen bewirken dagegen eine erhöhte Anzahl von Additionsoperationen. Außerdem hängt auch die Dauer der Addition des Gesamt-Differenzbildes auf das 0-markierte Bild stark von der Anzahl der Differenzpixel ungleich Null ab. Jeder Differenzpixel muss auf den R-, G- und B-Wert des entsprechenden Pixels im 0-markierten Bild addiert werden. Der deutlich größere Anteil dieser Schwankungen kommt jedoch von den Additionen der einzelnen Differenzbilder (vgl. Abbildung 7.6). Durch diese Abhängigkeit von der Bildbeschaffenheit werden so, ausgenommen Bildkategorie Eins, Schwankungen in den relativen Geschwindigkeitsfaktoren von 75% erreicht (57% - 132%).

Beim Verfahren im Fourierraum wirken sich die Eigenschaften eines Bildes deutlich weniger stark auf die Rechenzeit aus. Die größte Schwankung in der relativen Performanz des Shufflings zwischen zwei Bildern liegt hier bei 30%. Hier sind die Schwankungen hauptsächlich auf die Addition des Gesamt-Differenzbildes auf das 0-markierte Bild zurückzuführen.

Es ist zu beobachten, dass die Rechendauer für das Shuffling in der Pixeldomäne bei den Bildkategorien Fünf, Sechs und Sieben relativ gesehen kaum höher ist als die bei den kleineren Bildkategorien. Die Größe des Java Heapspace (768 MB) reicht für diese Bildgrößen nicht aus, um die Differenzbild-Additionen in Pyramidenform durchzuführen. Dennoch nimmt die Markierungsgeschwindigkeit nicht signifikant ab.

Abbildung 7.4 zeigt die absolute mittlere Rechenzeit aller Testbilder für die normale ImageMark Einbettung und das Shuffling der beiden entwickelten Verfahren.

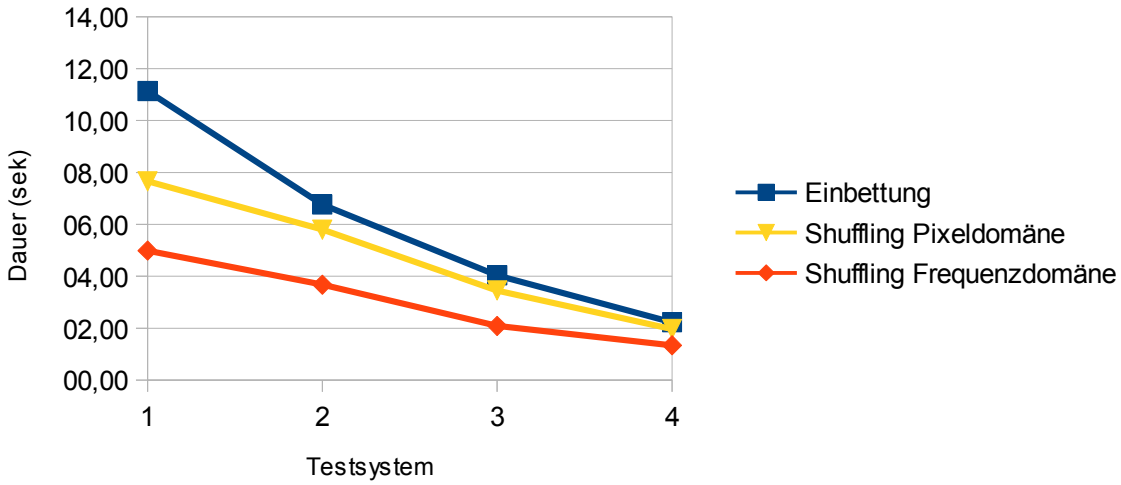

Abbildung 7.4.: Absolute Dauer der Einbettung und des Shufflings - Mittelwert aus zehn Durchführungen für 100 Testbilder verschiedener Größen

Hier wird der Unterschied der absoluten Rechendauer der verschiedenen Testsysteme sehr deutlich. Auf Testsystem Vier wird für die normale Einbettung nur 20% der Zeit benötigt, die auf Testsystem Eins für die selbe Operation verbraucht wird. Für das Verfahren im Frequenzraum verbraucht das Shuffling auf Testsystem Vier nur 27% der Rechenzeit, die auf Testsystem Eins benötigt wird. Für das Verfahren im Pixelraum sind es 26%.

Bei dieser Darstellung der Performanz muss bedacht werden, dass sich Unterschiede zwischen den Verfahren bei größeren Bilddateien und somit längeren Rechenzeiten stärker auf den Mittelwert auswirken als die Unterschiede bei kleinen Bilddateien. Um einen relativen Vergleich zu erhalten, der das Verhältnis zwischen Einbettung und Shuffling für alle Bilder gleich gewichtet, sollte Abbildung 7.2 herangezogen werden.

Zusammensetzung der Shuffling-Rechenzeit

In diesem Abschnitt wird die Rechenzeit der einzelnen Berechnungsschritte des Shuffling-Prozesses untersucht. Die Abbildungen 7.5 und 7.6 zeigen die Zusammensetzung der Rechenzeit des Shuffling-Prozesses der beiden entwickelten Container-Verfahren für ein Bild aus Kategorie Vier.

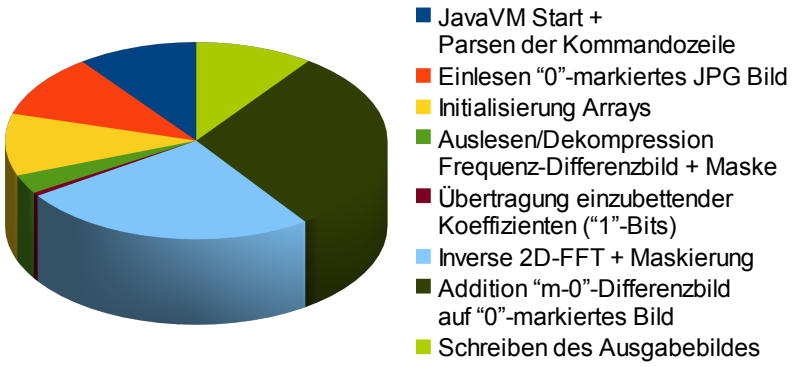

Abbildung 7.5.: Zusammensetzung der Gesamtrechenzeit beim Shuffling (Frequenzdomäne)

Der größte Anteil der Rechenzeit beim Verfahren im Frequenzraum wird mit jeweils über 25% für die inverse Fourier-Transformation und die Maskierung, sowie für die Addition des Gesamt-Differenzbildes auf das 0-markierte Bild benötigt. Alle drei Vorgänge sind allerdings schon stark optimiert. Für die Berechnung der inversen Fourier-Transformation wird FFTW verwendet (vgl. Kapitel 6.3) und bei der visuellen Maskierung kann auf die Berechnung der β-Werte verzichtet werden, da diese schon beim Containering vorberechnet werden. Des Weiteren wird die block-

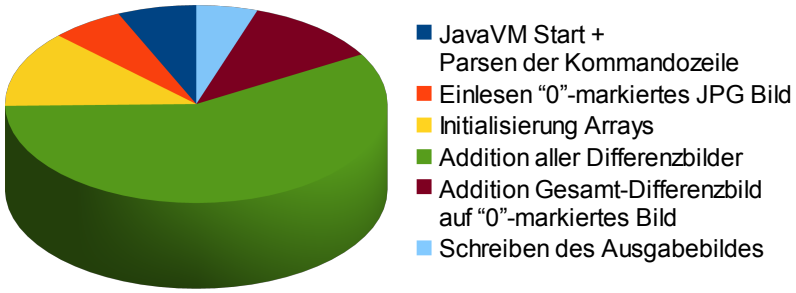

Abbildung 7.6.: Zusammensetzung der Gesamtrechenzeit beim Shuffling (Pixeldomäne)

weise Maskierung in eigene Threads ausgelagert. Bei der Addition des Gesamt-Differenzbildes auf das 0-markierte Bild wird auf das Auslesen und Zurückschreiben der RGB-Werte für Differenzpixel, deren Wert kleiner als $0,5$ ist, verzichtet. Durch die Verstärkung der Maskenwerte um μ gibt es allerdings nur sehr wenige Differenzpixel, die nicht mindestens auf 1 abgebildet werden.

Ein jeweils gleicher Anteil von ca. 10% wird durch das Einlesen des 0-markierten Bildes, die Initialisierung der Arrays, sowie durch das Schreiben des markierten Bildes verbraucht. Ein nur kleiner Anteil von 3% wird für für das Auslesen und Dekomprimieren des Frequenz-Differenzbildes und der Maske benötigt. Die eigentliche Markierung („Übertragung einzubettender Koeffizienten“) verbraucht dagegen nur einen verschwindend kleinen Anteil der Gesamtrechenzeit.

Beim Shuffling für das Verfahren im Pixelraum wird über 50% der Rechenzeit für die Addition aller Differenzbilder verbraucht. Diese Berechnungen werden, wenn genügend Speicherplatz zur Verfügung steht, parallel in Pyramidenform durchgeführt und sind somit schon stark optimiert. Bei der Initialisierung der Datenstrukturen wird mit ca. 12% der zweitgrößte Anteil der Gesamtrechenzeit verbraucht, da für die Darstellung der Differenzbilder sehr viel Speicherplatz benötigt wird. Die Addition des Gesamt-Differenzbildes auf das mit $\{0\}^l$ markierte Bild benötigt hier 11%. Weniger als 7% wird jeweils für das Einlesen des 0-markierten Bildes und für das Schreiben des markierten Bildes benötigt.

Bei beiden Verfahren wird für das Starten der Java Virtual Machine ein nicht zu vernachlässigender Anteil der Gesamtrechenzeit verbraucht (dunkelblau in beiden Abbildungen). Dieser Anteil würde allerdings wegfallen, wenn die JVM bereits auf einem System läuft, wenn eine individuell markierte Bilddatei durch Shuffling erzeugt werden soll. Wird das Shuffling auf einem Server durchgeführt, sollte daher stets eine aktive JVM vorhanden sein.

7.2.3 Geschwindigkeitsgewinn durch die Verwendung von FFTW

Durch die Verwendung von FFTW soll eine Beschleunigung der ImageMark-Detektion, der herkömmlichen Einbettung, des Containerings und des Shufflings erreicht werden. Das ImageMark Projekt enthält die Klasse `CFFT`, die zur Berechnung von zweidimensionalen Fourier Transformation verwendet werden kann. Diese ist allerdings weder Multithreading-fähig ist, noch auf andere Weise Performanz-optimiert.

Beim Shuffling des Container-Verfahrens im Fourierraum wird FFTW mittels JNI verwendet, um die zweidimensionale inverse Fouriertransformation durchzuführen (vgl. Kapitel 6.3). Außerdem wurde FFTW auch in die ImageMark Klasse `CFFT` integriert. Dabei wurde nicht nur die Implementierung der inversen Fouriertransformation angepasst, sondern auch die Implementierung der Vorwärts-Transformation. Die Verwendung von FFTW sollte sich somit nicht nur auf die Einbettungsgeschwindigkeit, sondern auch auf die Detektionsgeschwindigkeit auswirken.

Auf Testsystem Zwei wird die Wasserzeicheneinbettung und -detektion mittels ImageMark für alle 100 Testbilder mit und ohne die Verwendung von FFTW durchgeführt. Sowohl für die Einbettung als auch für die Detektion wird durch die Verwendung von FFTW eine Beschleunigung um ca. 20% erreicht. Dabei gibt es keine nennenswerten Schwankungen zwischen den einzelnen Bildkategorien und auch keine Unregelmäßigkeiten innerhalb dieser.

7.2.4 Geschwindigkeitsgewinn durch Multithreading

In diesem Abschnitt wird untersucht wie sich die Parallelisierung der Berechnungen beim Shuffling auf die Rechenzeit auswirkt.

Beim Shuggling des Container-Verfahrens im Frequenzraum werden folgende Operationen in eigene Threads ausgelagert:

- Auslesen der Koeffizienten des Frequenz-Differenzbildes
- Dekompression und Dekodierung der Maskenwerte
- Übertragung einzubettender Koeffizienten
- Maskierung

Die Berechnung der inversen Fourier-Transformationen wird nicht in eigene Threads ausgelagert, da FFTW bereits stark optimiert ist.

Beim Shuffling des Container-Verfahrens im Pixelraum werden folgende Operationen in eigene Threads ausgelagert:

- Dekompression und Dekodierung der Differenzbilder
- Paarweise Addition von Differenzbildern

Auf Testsystem Zwei beträgt die Rechenzeit für das Shuffling mit den genannten Multithreading Optimierungen 96% (Fourierdomäne), bzw. 98% (Pixeldomäne) der Rechenzeit, die ohne diese Optimierungen benötigt wird. Auf Testsystem Vier wird der Unterschied dagegen schon deutlicher. Hier beträgt die Rechenzeit des Shufflings mit Multithreading 88% (Fourierdomäne), bzw. 94% (Pixeldomäne) der Rechenzeit, die ohne Multithreading benötigt wird.

7.3 Robustheit

In diesem Abschnitt wird untersucht, wie sich die Verwendung der beiden entwickelten Container-Verfahren auf die Robustheit des ImageMark Wasserzeichens auswirkt. Ein minimaler Verlust an Robustheit ist durch einen starken Geschwindigkeitsgewinn annehmbar. Allerdings darf die Robustheit durch die Verwendung des Container-Verfahrens nicht deutlich abnehmen.

Um eine differenzierte Beurteilung treffen zu können, genügt es nicht zu untersuchen, ob ein Wasserzeichen nach dem Containering- und Shufflingprozess wieder ausgelesen werden kann. Vielmehr muss die Einbettungsstärke der einzelnen Nachrichten-Bits betrachtet werden. Wie in Kapitel 4.1.2 beschrieben, wird bei der Detektion eines eingebetteten Nachrichten-Bits der Quotient σ aus den Summen der beiden Koeffizientengruppen eines Nachrichten-Bits bestimmt. Dieser Quotient definiert die Stärke des detektierten Nachrichten-Bits. Die Robustheit lässt sich daher über die Verteilung dieser Quotienten ermitteln.

Um die Robustheit der beiden entwickelten Verfahren zu bewerten, wird nun die Verteilung dieses Quotienten für alle 100 Testbilder betrachtet. Das Diagramm in Abbildung 7.7 zeigt die Verteilung des Quotienten σ für die herkömmliche ImageMark-Einbettung und die beiden entwickelten Container-Verfahren.

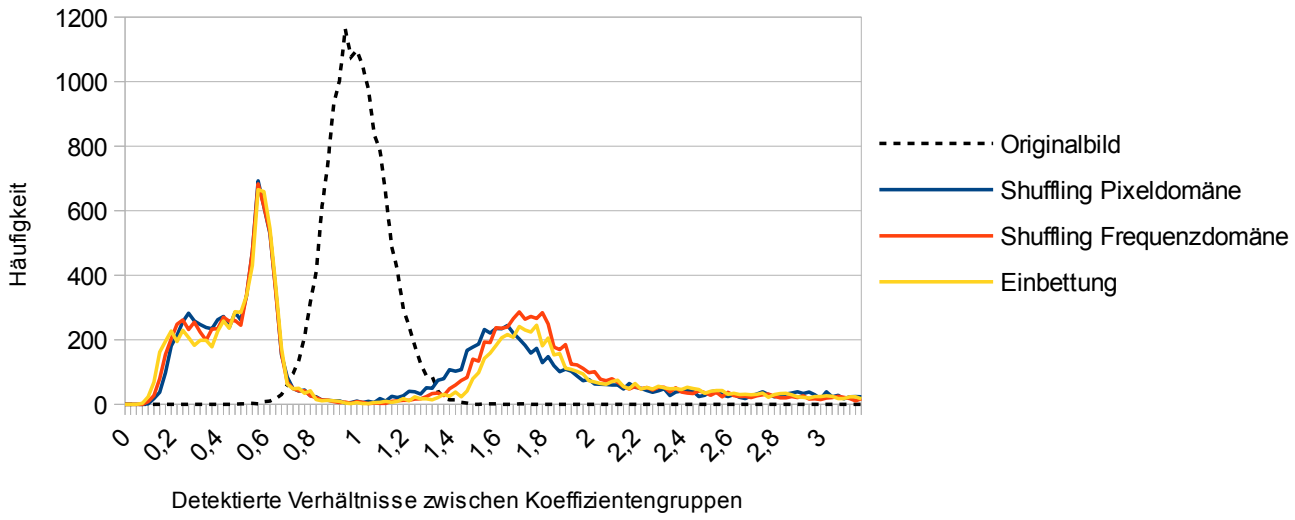

Abbildung 7.7.: Wasserzeichenstärke der ImageMark-Einbettung und der Container-Verfahren

Für eingebettete Nachrichten-Bits mit dem Wert 0, also für $\sigma < 1$, zeigt das Histogramm eine sehr ähnliche Verteilung für die normale Einbettung und die beiden Container-Verfahren. Nachrichten-Bits mit dem Wert 0 verlieren somit durch die Verwendung beider Container-Verfahren nicht an Robustheit gegenüber der normalen ImageMark-Einbettung.

Im Bereich der 1-Bits lassen sich allerdings leichte Unterschiede feststellen. Insgesamt fällt die Kurve für das Verfahren im Pixelraum etwas flacher aus und beginnt näher an 1 als die Verteilung der σ Werte für die anderen beiden Verfahren. Somit werden Nachrichten-Bits mit dem Wert 1 für das Verfahren im Pixelraum etwas weniger robust eingebettet als bei der herkömmlichen Einbettung. Diese Tatsache ist auf die in Kapitel 5.2.2 beschriebene Anpassung des Mindestdifferenzwertes ν bei der visuellen Maskierung der Differenzbilder beim Containering zurückzuführen. Um einen Kompromiss zwischen Robustheit der 1-Bits und Transparenz zu finden, wurde der Mindestdifferenzwert wurde auf $\nu = 0,11$ gesetzt. Diese Anpassung schwächt die Wasserzeichenenergie 1-Bits leicht ab, begrenzt aber den Verlust an Bildqualität und Transparenz des Wasserzeichens.

Der Verlauf oberhalb von 1 stimmt dagegen für die Kurven der normale Einbettung und des Verfahrens im Fourierraum recht genau überein. Diese Übereinstimmung wird durch die in Kapitel 5.3.2 beschriebene Vergrößerung der gerundeten Maskenwerte um $\mu = 20\%$ erreicht. Das Maximum der Kurve des normalen Einbettungsverfahrens ist etwas niedriger als das der Kurve für das Verfahren im Fourierraum. Der Grund dafür ist, dass diese Kurve im Gegensatz zu der Kurve der beiden Container-Verfahren noch sehr weit flach ausläuft.

Das Container-Verfahren im Fourierraum ist somit ähnlich robust wie das normale Einbettungsverfahren. Die Wasserzeichenenergie der Nachrichten-Bits mit dem Wert 1 ist für das Verfahren im Pixelraum etwas geringer als bei der herkömmlichen Einbettung. Allerdings handelt es sich dabei nur um eine sehr geringe Abnahme der Robustheit.

7.4 Bildqualität

In diesem Kapitel wird untersucht, wie sich die beiden entwickelten Container-Verfahren auf die Bildqualität auswirken. Um die Auswirkung einer Manipulation auf die Bildqualität zu bewerten, kann der PSNR-Wert (aus dem Englischen: *peak signal-to-noise ratio*) verwendet werden. Dieser Wert wird häufig für die Bewertung der subjektiven visuellen Qualität von verlustbehafteten Kompressionsverfahren verwendet. Der PSNR-Wert von zwei Bildern wird berechnet, indem zunächst die mittlere quadratische Abweichung (MSE) der Pixelwerte beider Bilder berechnet wird. Dieser Wert wird dann auf eine logarithmische Skala abgebildet.

Um die Bildqualität der beiden Container-Verfahren zu bewerten wird auch an dieser Stelle ein relativer Vergleich zur normalen Einbettung gezogen. Die Bildqualität des ImageMark-Einbettungsprozesses wird an dieser Stelle nicht bewertet; vielmehr gilt es zu zeigen, dass die

Bildqualität im Vergleich zum herkömmlichen Einbettungsprozess nicht abnimmt. Abbildung 7.8 enthält ein Diagramm, das den PSNR-Wert der 100 markierten Testbilder für die normale Einbettung, sowie für die beiden entwickelten Container-Verfahren enthält. Sowohl bei der Wasserzeicheneinbettung mittels ImageMark, als auch beim Containering und Shuffling wurde bei der Speicherung eines Bildes im JPEG-Format stets eine Qualitätsstufe von 90% verwendet (vgl. Tabelle 7.1).

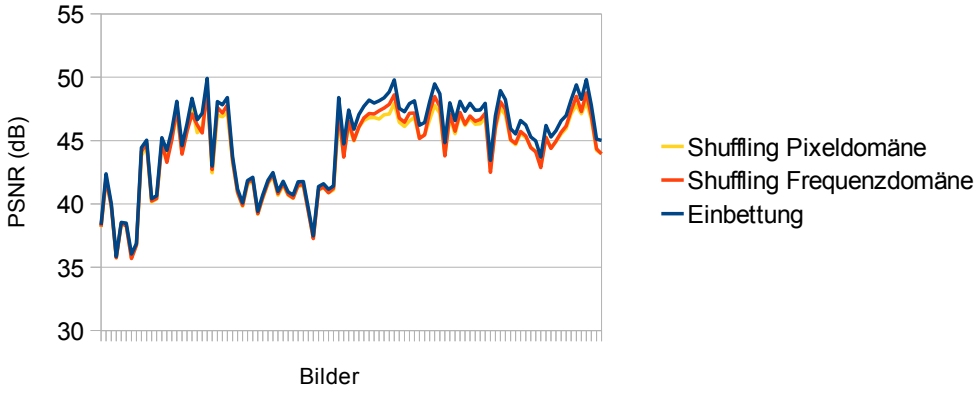

Abbildung 7.8.: Vergleich der Bildqualität zwischen Einbettung und Container-Verfahren

Die PSNR-Werte stimmen bei den meisten Bildern beinahe überein. Bei einigen Bildern gibt es leichte Verschlechterungen der Bildqualität gegenüber dem normalen Einbettungsverfahren. Diese Qualitätsverluste sind auf die zweifache JPEG-Kompression zurückzuführen. Bei beiden Container-Verfahren wird die 0-markierte Version des Originalbildes als JPEG-komprimiertes Bild in der Containerdatei abgelegt. Am Ende des Shufflings wird bei beiden Verfahren wiederum eine JPEG-Kompression vorgenommen. Beim herkömmlichen ImageMark-Einbettungsprozess wird dagegen nur eine JPEG-Kompression durchgeführt. Da JPEG ein verlustbehaftetes Kompressionsverfahren ist, wirkt sich diese zusätzliche Kompression negativ auf die Bildqualität aus.

Die Abweichungen von den PSNR-Werten für das normale Einbettungsverfahren sind jedoch sehr gering und liegen für das Verfahren im Fourierraum bei durchschnittlich $0,64$ dB, bzw. für das Verfahren im Pixelraum bei $0,81$ dB. Die maximale Differenz beträgt beim Verfahren im Fourierraum $1,55$ dB und beim Verfahren im Pixelraum $1,85$ dB. Die subjektive Bildqualität nimmt somit durch die Verwendung der Containainerverfahren nur in sehr geringem Maße ab.

8 Fazit

Zunächst wurde der aktuelle Stand der Forschung auf dem Gebiet der Bildwasserzeichen an Hand eines Verfahrens in der DWT/DFT Domäne und eines Verfahrens in der DFT/LPM Domäne beschrieben. In der Motivation wurde dargestellt, dass die Einbettungsgeschwindigkeit von robusten Bildwasserzeichen für die meisten Anwendungsfälle nicht genügend schnell ist. Die Funktionsweise des ImageMark Bildwasserzeichens, welches die Grundlage dieser Arbeit ist, wurde detailliert beschrieben. Das Konzept des digitalen Wasserzeichen-Containers, welches eine Verkürzung der Einbettungszeit für ein vorhandenes Wasserzeichenverfahren ermöglicht, wurde im Folgenden vorgestellt. Es wurden zwei neuartige Konzepte eines Container-Verfahrens für das ImageMark Bildwasserzeichen und deren technische Umsetzung beschrieben. Es konnte gezeigt werden, dass beide Mechanismen eine Verkürzung der Einbettungsdauer bei beinahe gleich bleibender Bildqualität und Robustheit ermöglichen. Dabei erreicht das Verfahren in der Fourierdomäne bei geringerer Containerdateigröße eine kürzere Einbettungsdauer als das Verfahren im Pixelraum. Eine Containerdatei des Verfahrens im Fourierraum ist in etwa doppelt so groß wie ein Originalbild im JPEG-Format. Das Shuffling für dieses Verfahren benötigt durchschnittlich nur 60% der ImageMark Einbettungsdauer.

9 Ausblick

Beim Shuffling des Verfahrens in der Pixeldomäne wird der Großteil der Rechenzeit für die Addition der Differenzbilder benötigt. Beim Verfahren im Fourierraum wird der größte Anteil für die Addition des Gesamtdifferenzbildes auf das 0-markierte Bild benötigt. Diese Additionsoperationen werden jeweils Elemente-weise und unabhängig voneinander auf Arrays durchgeführt. Ein Geschwindigkeitsgewinn könnte hier erreicht werden, wenn die Fähigkeit moderner Architekturen, Rechenoperationen parallel durchzuführen, besser genutzt wird. Diese Fähigkeit von Prozessoren wird als *Single instruction, multiple data* (SIMD) bezeichnet und ist beispielsweise in den MMX- und SSE-Befehlssätzen (SSE, SSE2,..., SSE5) enthalten. Durch das Anpassen des Programmcodes auf die Verwendung dieser Befehle könnte ein Geschwindigkeitszuwachs erreicht werden. Moderne Grafikkarten unterstützen die parallele Ausführung von Rechenoperationen allerdings noch besser. Die Ausführung der Wasserzeicheneinbettung oder des Shufflings auf einer Grafikkarte verspricht daher enorm verringerte Rechenzeiten. Dies könnte beispielsweise mittels CUDA[1] umgesetzt werden. Lin et al. erreichen in [15] durch die Portierung des Programmcodes eines Bildwasserzeichens auf CUDA eine Beschleunigung um das bis zu 36-fache der ursprünglichen Rechenzeit des Wasserzeichens.

Eine mögliche Optimierung des Verfahrens im Pixelraum besteht darin, anstatt der Differenz aus den Luminanz-Pixelwerten der 0- und 1-markierten Bilder, die Differenz der DCT-Luminanz-Blöcke dieser Bilder zu berechnen und in der Containerdatei zu speichern. Beim Shuffling wird dann die Addition der Wasserzeichen-Differenzbilder in der DCT-Domäne durchgeführt. Die DCT-Transformation bewirkt im Allgemeinen eine Konzentrierung der Energie eines Bildblockes, sodass ein DCT-Luminanzblock mehr Null-Werte enthält als dessen Repräsentation im Pixelraum. Die Differenz aus zwei ähnlichen Bildblöcken in der DCT-Domäne enthält somit auch wesentlich mehr Null-Werte als die Differenz aus zwei Bildblöcken in der Pixeldomäne. Diese höhere Anzahl an Null-Werten könnte zum Einen die Containerdateigröße und zum Anderen die Anzahl der benötigten Additionsoperationen verringern. Eine Stufe der JPEG-Komprimierung/Dekomprimierung beinhaltet ohnehin die blockweise Darstellung der Luminanz eines Bildes in der DCT-Domäne. Somit verspricht eine Integration dieses DCT-Luminanz Container-Verfahrens in JPEG eine starke Beschleunigung des Shufflings.

Das Container-Verfahren in der Frequenzdomäne enthält ein Sicherheitsproblem. Bei diesem Verfahren sind die Positionen der zur Markierung verwendeten Frequenzkoeffizienten unverschlüsselt in der Container-Datei enthalten. Von diesen Positionen kann ein Rüchschluss auf den

[1] www.nvidia.com/object/cuda_home.html, Zugriff: 10.12.2011

bei der Container-Erzeugung benutzten geheimen Schlüssel, oder zumindest auf den Hash-Wert dieses Schlüssels, gezogen werden. Das Erzeugen von individuell markierten Versionen eines Originalbildes aus einer Containerdatei in einer unsicheren Umgebung (vgl. Kapitel 4.2.1) kann somit nicht durchgeführt werden ohne Informationen über den geheimen Schlüssel preiszugeben. Die Aufgabe zukünftiger Arbeiten ist es somit, einen Mechanismus zu finden, der dieses Sicherheitsproblem behebt.

Insgesamt konnte zwar eine Beschleunigung des Markierungsvorgangs erreicht werden, jedoch ist der Faktor, um den das Shuffling der Container-Verfahren schneller ist als die ImageMark-Einbettung, nicht so groß wie erwartet. Die Aufgabe zukünftiger Arbeiten sollte es daher sein, die entwickelten Verfahren weiter zu optimieren und neue Ansätze zu finden, die eine effizientere Wasserzeicheneinbettung ermöglichen.

Literaturverzeichnis

[1] J. Buchmann:
Einführung in die Kryptographie. 4., erweiterte Auflage, Springer-Verlag, Deutschland, ISBN: 3-540-74451-7, 2008.

[2] I. J. Cox, M. L. Miller, J. A. Bloom:
Digital Watermarking. Morgan Kaufmann Publishers, ISBN: 1-55860-714-5, 2002.

[3] H. Dobbertin, A. Bosselaers, B. Preneel:
RIPEMD-160, a strengthened version of RIPEMD. Fast Software Encryption, LNCS 1039, D. Gollmann, Ed., Springer-Verlag, 1996, pp. 71-82.

[4] J. Dittmann:
Digitale Wasserzeichen. Springer Verlag Berlin Heidelberg, ISBN: 3-540-66661-3, 2000.

[5] G. F. Elmasry, Y. Q. Shi:
2-D interleaving for enhancing the robustness of watermark signals embedded in still image. In Proceedings of IEEE International Conference on Multimedia & Expo (ICME00), New York, July 31 to August 2, 2000.

[6] J. Fridrich:
Methods for data hiding. Center for Intelligent Systems & Department of Systems Science and Industrial Engineering, SUNY Binghampton, 1997.

[7] M. Frigo, S. G. Johnson:
The Design and Implementation of FFTW3. In Proceedings of the IEEE 93 (2), 216-231 (2005). Invited paper, Special Issue on Program Generation, Optimization, and Platform Adaptation.

[8] J. W. Cooley, J. W. Tukey:
An algorithm for the machine calculation of complex Fourier series. In Math. Comput. 19, 1965, S. 297-301.

[9] E. Hauer:
Digitale Wasserzeichen für MPEG-Videos zur Authentifizierung des Urhebers und Videos. Dissertation, Fakultät für Informatik der Otto-von-Guericke-Universität Magdeburg, 2009.

[10] M. T. Heideman, D. H. Johnson, C. S. Burrus:
Gauss and the History of the Fast Fourier Transform. In Arch. Hist. Sc. 34, Nr. 3, 1985.

[11] J. Huang, Y. Q. Shi:
An adaptive image watermarking scheme based on visual masking. Electron. Lett., vol. 34, no. 8, pp. 748-750, 1998.

[12] X. Kang, J. Huang, Y. Q. Shim Y. Lin:
A DWT-DFT Composite Watermarking Scheme Robust to Both Affine Transform and JPEG Compression. IEEE Transactions on Circuits and Systems for Video Technology, vol. 13, no. 8, pp. 776-786, August 2003.

[13] S. Katzenbeisser, F. A. P. Petitcolas:
Information Hiding techniques for steganography and digital watermarking. Artech House, INC., ISBN: 1-58053-035-4, 2000.

[14] G. C. Langelaar, I. Setyawan, R. L. Lagendijk:
Watermarking digital image and video data. A state-of-the-art overview. Signal Processing Magazine, IEEE , vol.17, no.5, pp.20-46, Sep 2000

[15] C. Lin, L. Zhao, J. Yang:
A CUDA Based Implementation of an Image Authentication Algorithm. Information Engineering and Computer Science (ICIECS), 2010 2nd International Conference on , vol., no., pp.1-5, 25-26 Dec. 2010.

[16] S. Pereira, T. Pun:
Robust Template Matching for Affine Resistant Image Watermarks. In Proceedings of IEEE Transactions on Image Processing, vol. 9, no. 6, June 2000.

[17] R. Ridzoň, D. Levický:
Robust digital watermarking in DFT and LPM domain. 50th International Symposium ELMAR-2008, September 2008, Zadar, Coratia.

[18] R. Ridzoň, D. Levický:
Robust image watermarking based on the synchronization template. Radioelektronika, 2008 18th International Conference , vol., no., pp.1-4, 24-25 April 2008.

[19] C. I. Podilchuk, W. Zeng:
Image-adaptive watermarking using visual models. IEEE J. Select. Areas Mommun., vol. 16, pp. 525-539, 1998.

[20] Y. Q. Shi, X. M. Zhang:
A new two-dimensional interleaving technique using successive packing. IEEE Trans. Circuits Syst. I, vol. 49, pp. 779-789, June 2002.

[21] M. Steinebach:
Digitale Wasserzeichen für Audiowasserzeichen. ISBN 3-8322-2507-2, Shaker Verlag, 2004.

[22] M. Steinebach, S. Zmudzinski, F. Chen:
The digital watermarking container: Secure and efficient embedding. In Processings of the ACM Multimedia and Security Workshop, 20.-21. September 2004, Magdeburg.

[23] M. Steinebach, E. Hauer, P. Wolf:
Efficient Watermarking Strategies. In Proceedings of the Third International Conference on Automated Production of Cross Media Content for Multi-Channel Distribution, 2007. pp. 65-71.

[24] S. Thiemert:
Werkzeuge zur Qualitätsevaluierung und Vorschläge zur Optimierung von MPEG-Video-Wasserzeichen. Diplomarbeit, Hochschule Anhalt, 2002.

[25] S. Voloshynovskiy, F. Deguillaume, T. Pun:
Content adaptive watermarking based on a stochastic multiresolution image modeling. In Proc. 10th European Signal Processing Conf. (EUSIPCO'2000), Tampere, Finland, Sept. 2000.

[26] P. Wolf, E. Hauer, M. Steinebach:
The Video Watermarking Container - efficient real-time transaction watermarking. In Proc. SPIE 6819, 68190K (2008), DOI:10.1117/12.766544.

A Anhang

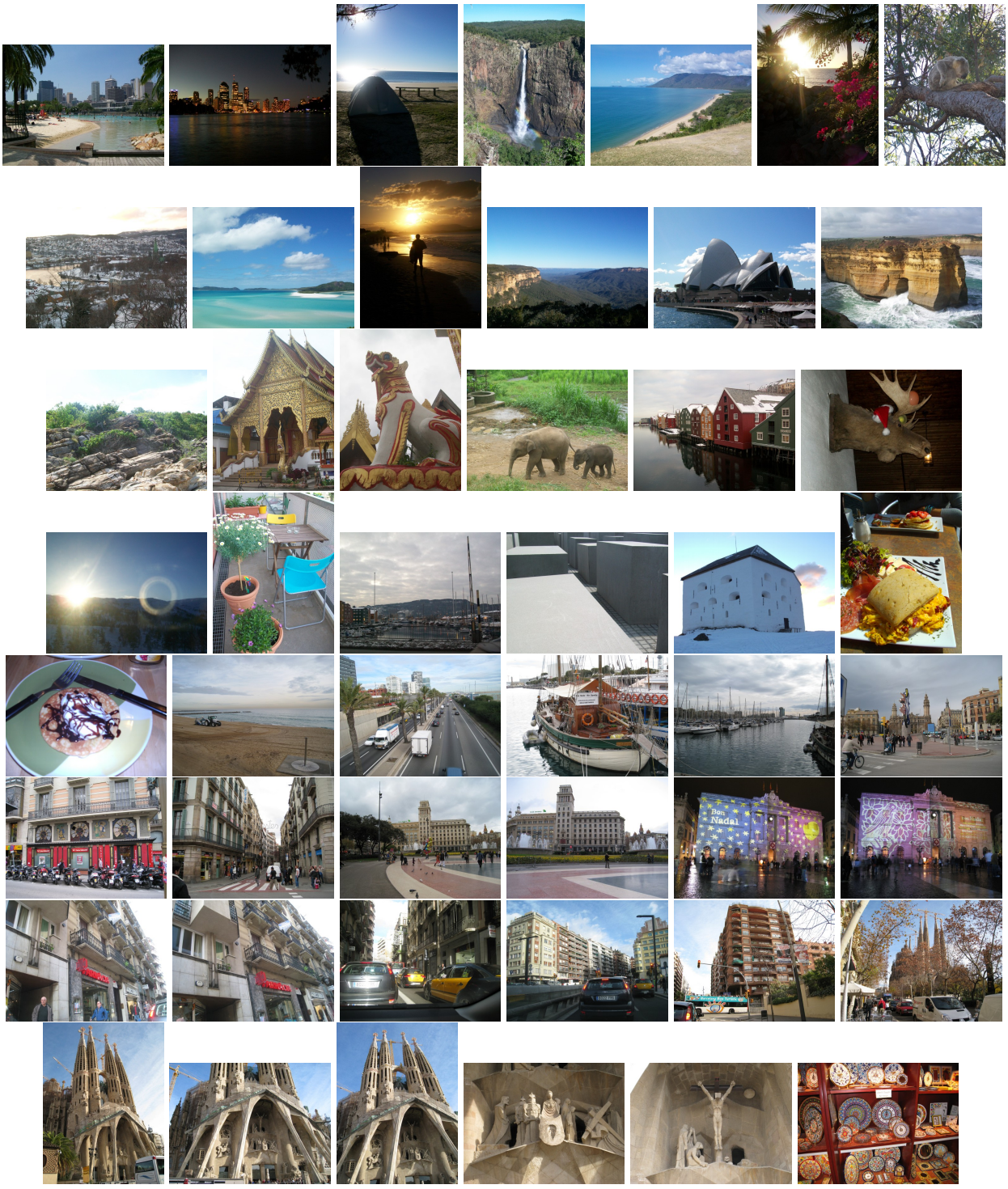

Abbildung A.1.: Zur Evaluierung verwendete Bilder (1/2)

Abbildung A.2.: Zur Evaluierung verwendete Bilder (2/2)