F5 – ein steganographischer Algorithmus

Hohe Kapazität trotz verbesserter Angriffe

Andreas Westfeld

Technische Universität Dresden
Institut für Systemarchitektur
01062 Dresden
westfeld@inf.tu-dresden.de

Zusammenfassung Viele steganographische Systeme zeigen Schwächen gegenüber visuellen und statistischen Angriffen. Systeme, die diese Schwächen nicht zeigen, bieten nur relativ geringe Kapazität für steganographische Nachrichten. Der neu entwickelte Algorithmus F5 hält visuellen und statistischen Angriffen stand und bietet dennoch eine hohe steganographische Kapazität. F5 verwendet Matrixkodierung zur Erhöhung der Einbettungseffizienz. Dadurch verringert sich die Zahl nötiger Änderungen. Durch permutative Spreizung wird die Nachricht bei geringer Ausnutzung der Kapazität gleichmäßig im Steganogramm verteilt.

1 Einführung

Steganographische Algorithmen betten vertrauliche Nachrichten in andere, umfangreichere Nachrichten (Trägermedien) ein. Durch das Einbetten in ein Trägermedium entsteht ein Steganogramm, aus dem der Empfänger die Nachricht wieder extrahieren kann. Ein Angreifer soll Steganogramme und Trägermedien aber möglichst nicht unterscheiden können, so dass er weder die vertrauliche Nachricht erfährt noch den Umstand, dass etwas eingebettet wurde. Die Mehrzahl der ca. 40 im Internet erhältlichen steganographischen Programme kann zwar viel, aber nicht besonders unauffällig einbetten [8].

Visuelle Angriffe beruhen darauf, dass steganographische Algorithmen wesentliche Informationen im Trägermedium überschreiben [5]. Adaptive Techniken, die die Einbettungsintensität an den Bildinhalt anpassen, verhindern visuelle Angriffe, jedoch verringern sie auch den relativen Anteil steganographischer Information im Trägermedium. Verlustbehaftet komprimierte Trägermedien (JPEG, MP3, ...) sind von Haus aus gegen visuelle Angriffe gefeit.

Das steganographische Programm Jsteg [4] bettet Nachrichten in verlustbehaftet komprimierte JPEG-Dateien ein, hat eine hohe Kapazität – z. B. 12 % der Dateigröße des Steganogramms – und ist gegen visuelle Angriffe sicher. Es gibt jedoch einen statistischen Angriff auf Jsteg, der die steganographischen Änderungen nachweisen kann [5].

Auch MP3Stego [3] und IVS-Stego [6] sind gegen auditive/visuelle Angriffe sicher. Die extrem niedrige Einbettungsrate verhindert darüber hinaus auch alle bekannten statistischen Angriffe. Die beiden steganographischen Programme bieten nur wenig Raum für steganographische Nachrichten (weniger als 1 % der Dateigröße des Steganogramms).

In Abschnitt 2 wird zunächst kurz auf das JPEG-Dateiformat eingegangen, insbesondere auf die Verteilung der JPEG-Koeffizienten, die durch das Programm Jsteg auffällig verändert wird. Jsteg (Abschnitt 3) dient als Ausgangspunkt für eine schrittweise Verbesserung. Als Konsequenz aus den statistischen Angriffen wird der Einbettungsalgorithmus von Jsteg ersetzt (Abschnitt 4). Dabei entsteht Schwund, der durch Wiederholen erfolglos eingebetteter Bits ausgeglichen wird. Infolgedessen entsteht erneut eine auffällige Verteilung der JPEG-Koeffizienten, wofür aber nicht länger das Einbetten, sondern die Wiederholung verantwortlich ist (welche ausschließlich steganographische Nullen betrifft). Abschnitt 5 ersetzt nach der Einbettungsoperation nun auch noch die steganographische Interpretation. Der Schwund trifft nun steganographische Nullen und Einsen gleichermaßen. Die beiden wichtigsten charakteristischen Eigenschaften der Verteilung überstehen den Einbettungsprozess, so dass der bekannte statistische Angriff fehl schlägt. Abschnitt 6 stellt permutative Spreizung vor, die die Einbettungsdichte auch bei geringer Ausnutzung der steganographischen Kapazität gleichmäßig verteilt. Durch Anwendung der Matrixkodierung verringert sich die Anzahl der nötigen steganographischen Änderungen.

2 JPEG Dateiformat

Das von der Joint Photographic Expert Group (JPEG) definierte Dateiformat speichert Bilddaten in verlustbehaftet komprimierter Form als quantisierte Frequenzkoeffizienten ab.

Abbildung 1 zeigt die Schritte, die zur Kompression durchgeführt werden. Der JPEG-Kompressor schneidet zunächst den unkomprimierten Bildinhalt (z. B. eine BMP-Datei) in 8×8-Teilbilder. Die diskrete Kosinustransformation überführt jeweils 8×8 Helligkeitswerte in 8×8 Frequenzkoeffizienten (reelle Zahlen). Die darauffolgende Quantisierung rundet die Frequenzkoeffizienten geeignet zu ganzen Werten (verlustbehafteter Schritt). Das Diagramm in Abbildung 2 zeigt die diskrete Verteilung der Häufigkeit dieser Werte.

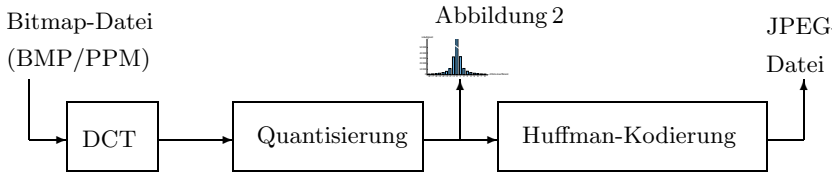


Abbildung 1. Informationsfluss im JPEG-Kompressor

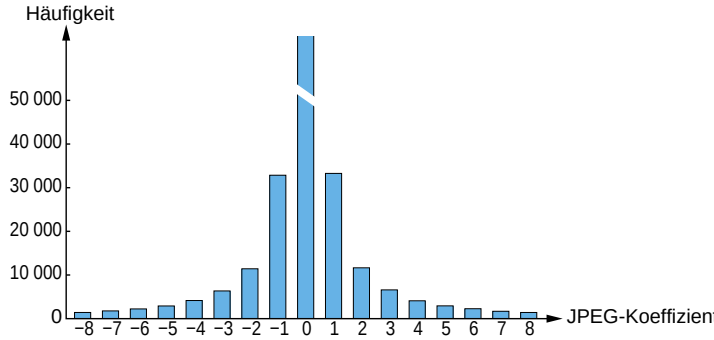


Abbildung 2. Histogramm der JPEG-Koeffizienten nach der Quantisierung

Wenn wir die Verteilung in Abbildung 2 betrachten, können wir zwei charakteristische Eigenschaften erkennen:

1. Die Häufigkeit der Koeffizienten nimmt mit zunehmendem Betrag ab.
2. Die Abnahme der Häufigkeit nimmt mit zunehmendem Betrag ab, d. h. der Unterschied zwischen zwei Säulen des Histogramms ist in der Mitte größer als am Rand.

Nach der verlustbehafteten Komprimierung sorgt die Huffman-Kodierung (verlustfrei) für möglichst redundanzarme Speicherung der quantisierten Koeffizienten. Eine genauere Beschreibung der JPEG-Kompression ist z. B. in [2] enthalten. Die folgenden Abschnitte beziehen sich hauptsächlich auf die Verteilung in Abbildung 2. Angaben von Dateigrößen und steganographischen Kapazitäten beziehen sich auf das True-Color-Bild *Expo*, das in Abbildung 3 zu sehen ist.

Abbildung 3. Trägermedium (Weltausstellung in Hannover 2000)

3 Jsteg

Der Algorithmus Jsteg von Derek Upham dient hier als Ausgangspunkt für die Betrachtung, da er gegen visuelle Angriffe resistent ist und dennoch eine erstaunliche Kapazität (12,8 %) für steganographische Nachrichten bietet. Nach der Quantisierung ersetzt Jsteg die niederwertigsten Bits der Frequenzkoeffizienten durch die steganographische Nachricht.[1] Der Einbettungsmechanismus überspringt dabei die Koeffizienten mit dem Wert 0 oder 1. Abbildung 4 zeigt die Einbettungsfunktion von Jsteg im C-Quelltext.

Der statistische Angriff [5] auf Jsteg erkennt das Vorhandensein eingebetteter Nachrichten jedoch zuverlässig, da Jsteg Bits ersetzt und damit einen Ausgleich zwischen den Häufigkeiten von Werten schafft, die sich nur in dieser Bitposition (hier LSB) unterscheiden.

Jsteg beeinflusst die Häufigkeiten c_i im Histogramm von JPEG-Koeffizienten i, was in Abbildung 5 zu sehen ist. Wird ein JPEG-Bild mit Jsteg verändert, dann erwarten wir, dass benachbarte Häufigkeiten $L = c_{2i}$ und $R = c_{2i+1}$ paarweise ausgeglichen werden (L und R bezeichnen die linken und rechten Elemente der entstehenden Pärchen im Histogramm). Diese Ausgleichung kann statistisch mit dem Chi-Quadrat-Test nachgewiesen werden. Dieser Test wird sehr oft verwendet, um Verteilungen miteinander zu vergleichen. Veranschaulichen wir uns den Angriff zunächst an einem Beispiel, das die Excel-Funktion chitest( ) verwendet (siehe Abbildung 6). In Spalte A sind die Häufigkeiten der JPEG-Koeffizienten vor dem Einbetten, in Spalte C die Werte nach dem Einbetten einer Nachricht mit Jsteg zu sehen. Die Verteilung der erwarteten Werte (Mittelwerte in

[1] Wir gehen von einer gleichverteilten Nachricht aus. Das vereinfacht nicht nur die Darstellung, sondern ist zudem plausibel, wenn die Nachricht komprimiert und verschlüsselt ist.

```
short use_inject = 1;                /* wird 0 bei Nachrichtenende */
short inject(short inval)       /* inval ist ein JPEG-Koeffizient */
{
    short inbit;
    if ((inval & 1) != inval)        /* nicht in 0 und 1 einbetten */
        if (use_inject) {            /* Nachrichtenbits vorhanden? */
            if ((inbit=bitgetbit()) != -1) { /* hole nächstes Bit */
                inval &= ~1;          /* überschreibe das LSB ...  */
                inval |= inbit;      /* ... mit diesem Bit (inval) */
            } else
                use_inject = 0;                /* Nachrichtenende */
        }
    return inval;  /* gib modifizierten JPEG-Koeffizienten zurück */
}
```

Abbildung 4. Die Jsteg-Einbettungsfunktion von Derek Upham

den Spalten B und D) ändert sich durch das Einbetten nicht. Deshalb kann auf den Vergleich mit dem Trägermedium verzichtet werden, dem Angreifer genügt also das potenzielle Steganogramm. Der Chi-Quadrat-Test ermittelt nun in Zelle D9 die Einbettungswahrscheinlichkeit für das Histogramm in Abbildung 5 und in Zelle B9 für Abbildung 2.

Der Angriff läuft also wie folgt ab: Wir berechnen das arithmetische Mittel

$$n_i^* = \frac{c_{2i} + c_{2i+1}}{2}, \tag{1}$$

um die erwartete Verteilung zu bestimmen, und vergleichen sie mit der beobachteten Verteilung

$$n_i = c_{2i}. \tag{2}$$

Der Unterschied zwischen den beiden Verteilungen n_i und n_i^* sei

$$\chi^2 = \sum_{i=1}^{k} \frac{(n_i - n_i^*)^2}{n_i^*} \tag{3}$$

mit $k-1$ Freiheitsgraden, was die um eins verminderte Anzahl der Klassen im Histogramm ist.

Abbildung 7 zeigt den beschriebenen statistischen Angriff auf ein Steganogramm, in dem 50 % der Kapazität (7680 Bytes) eingebettet wurden. Im Diagramm ist die Einbettungswahrscheinlichkeit

$$p = 1 - \frac{1}{2^{\frac{k-1}{2}}\,\Gamma(\frac{k-1}{2})} \int_0^{\chi^2} \mathrm{e}^{-\frac{t}{2}} t^{\frac{k-1}{2}-1} \mathrm{d}t \tag{4}$$

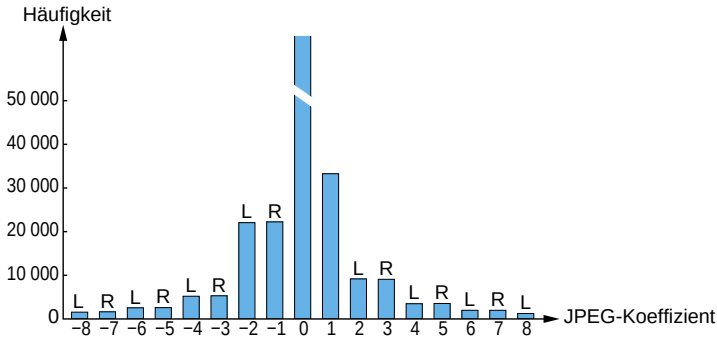
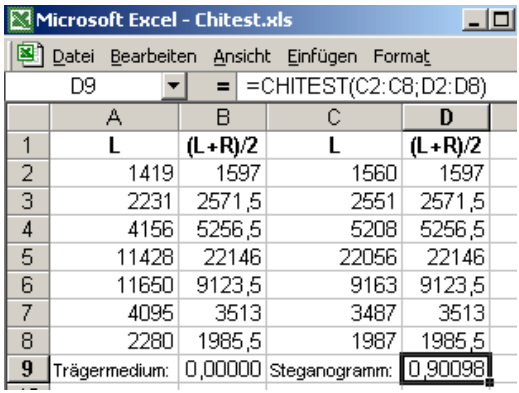


Abbildung 5. Jsteg gleicht Häufigkeiten paarweise aus.



	A	B	C	D
1	**L**	**(L+R)/2**	**L**	**(L+R)/2**
2	1419	1597	1560	1597
3	2231	2571,5	2551	2571,5
4	4156	5256,5	5208	5256,5
5	11428	22146	22056	22146
6	11650	9123,5	9163	9123,5
7	4095	3513	3487	3513
8	2280	1985,5	1987	1985,5
9	Trägermedium:	0,00000	Steganogramm:	0,90098

Abbildung 6. Statistischer Angriff mit Tabellenkalkulation

als Funktion einer kumulativen Stichprobe abgetragen, die zunächst 1 % der JPEG-Koeffizienten (vom Dateianfang beginnend) enthält. Die Stichprobe vergrößert sich dann auf die ersten 2 %, 3 %, ... und liefert bis 54 % eine Einbettungswahrscheinlichkeit von 1,00. Eine Stichprobe von 59 % und mehr enthält hingegen genügend unangetastete JPEG-Koeffizienten, um den p-Wert auf 0,00 fallen zu lassen.

4 F3

Der Algorithmus F3 dient als Lehrbeispiel. Er unterscheidet sich in zweierlei Hinsicht von Jsteg:

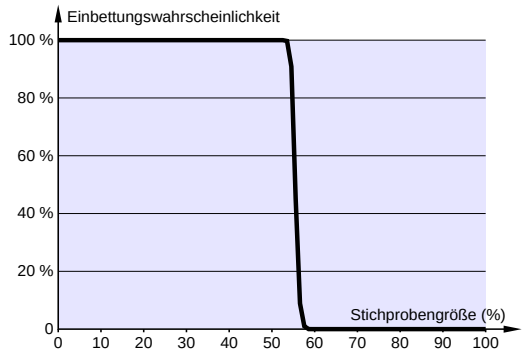


Abbildung 7. Statistischer Angriff auf ein Jsteg-Steganogramm (50 % der Kapazität ausgenutzt)

1. Statt Bits zu überschreiben, dekrementiert er den Betrag der Koeffizienten. Eine Ausnahme bilden Koeffizienten mit dem Wert 0, denn ihr Betrag lässt sich nicht dekrementieren und wird deshalb *nicht* steganographisch genutzt. Die niederwertigsten Bits der Koeffizienten stimmen also nach dem Einbetten mit der steganographischen Nachricht überein. Die Anpassung erfolgt aber nicht durch Überschreiben, denn das wäre mit dem Chi-Quadrat-Test leicht nachzuweisen [5]. Wir können also hoffen, dass keine Stufen in der Verteilung auftreten. Im Gegensatz zu Jsteg verwendet F3 Koeffizienten mit dem Wert 1. Die in Abbildung 2 sichtbare Symmetrie bleibt somit erhalten.
2. Manche Bits, die eingebettet werden, fallen dem Schwund zum Opfer. Schwund entsteht, wenn F3 den Betrag der Koeffizienten 1 und -1 dekrementiert und damit eine 0 erzeugt. Der Empfänger kann einen Koeffizienten mit dem Wert 0, der steganographisch ungenutzt ist, nicht von einer 0 unterscheiden, die durch Schwund entsteht. Er überspringt alle Koeffizienten mit dem Wert 0. Der Sender muss also das vom Schwund betroffene steganographische Bit – der Sender merkt ja, wenn er eine 0 erzeugt – wiederholt einbetten.

Verglichen mit Abbildung 2 enthält das Histogramm in Abbildung 8 eine relative Überzahl von geraden Koeffizienten. Diese Erscheinung ist auf das wiederholte Einbetten nach Schwund zurückzuführen. Schwund tritt nur auf, wenn in Koeffizienten mit dem Wert 1 oder -1 eine 0 eingebettet wird, aber nie beim Einbetten einer 1. F3 bettet die von Schwund betroffenen steganographischen Bits (das sind stets Nullen) wiederholt ein. Diese Wiederholung verschiebt das (ursprünglich ausgeglichene) Verhältnis von 0 und 1 zugunsten der 0. Damit entstehen im Histogramm bevorzugt ge-

rade Werte. Abbildung 8 zeigt die auffällige Verteilung von geraden und ungeraden Koeffizienten, die wir ebenfalls statistisch nachweisen können.

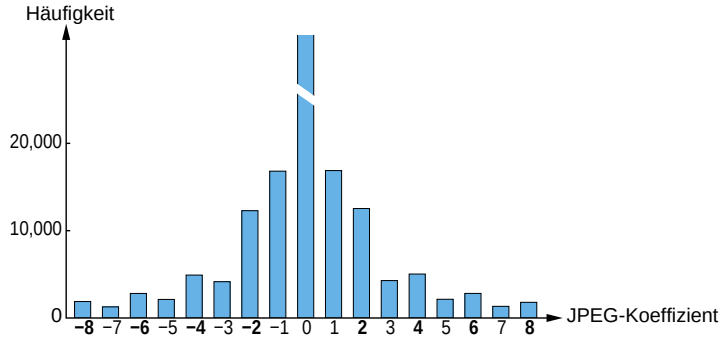


Abbildung 8. F3 erzeugt eine Überzahl an geraden Koeffizienten

Wenn wir den Schwund ignorieren und nicht wiederholt einbetten, entstehen keine Stufen im Histogramm. Leider erhält der Empfänger dann nur Bruchstücke der Nachricht. Die Benutzung eines fehlerkorrigierenden Kodes könnte das Problem wahrscheinlich lösen.

Wenn wir mit F3 scheinbare Nachrichten aus unveränderten Trägermedien auslesen, dann erhalten wir wesentlich mehr Einsen als Nullen. Eine Nachricht, die etwas mehr Einsen als Nullen (im geeigneten Verhältnis) enthält, lässt die Stufen im Histogramm ebenfalls verschwinden. Eine elegantere Lösung des Problems nutzt die Symmetrie in Abbildung 2 aus.

5 F4

F3 hat zwei Schwächen:

1. Durch den ausschließlichen Schwund von steganographischen Nullen bettet F3 effektiv wesentlich mehr Nullen als Einsen ein und erzeugt – wie auch Jsteg, nur auf andere Weise – Auffälligkeiten im Histogramm, die statistisch nachweisbar sind.
2. Das Histogramm unveränderter JPEG-Dateien (Abbildung 2) enthält wesentlich mehr ungerade als gerade Koeffizienten (0 ausgenommen). *Unveränderte* Trägermedien enthalten also (aus der Sicht von F3 und Jsteg) mehr steganographische Einsen als Nullen.

Der Algorithmus F4 beseitigt diese beiden Schwächen auf einen Streich, indem er negativen Koeffizienten den invertierten steganographischen Wert zuordnet: Negative gerade Koeffizienten stehen für eine steganographische Eins, negative ungerade für eine Null; positive gerade stehen wie bisher für eine Null, positive ungerade für eine Eins. Abbildung 9 lässt leicht erkennen, dass jeweils zwei Säulen gleicher Höhe steganographisch invers zueinander interpretiert werden (steganographische Nullen sind schwarz, steganographische Einsen weiß).

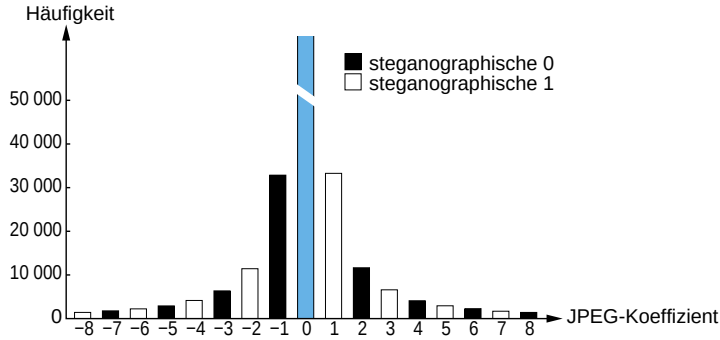


Abbildung 9. F4 interpretiert die JPEG-Koeffizienten (Abbildung 2) anders als Jsteg

Abbildung 10 zeigt die Einbettungsfunktion von F4 als Java-Quelltext. Das Feld coeff[ ] enthält sämtliche JPEG-Koeffizienten des Trägermediums.

Im Folgenden wird nachgewiesen, dass die in Abschnitt 2 genannten charakteristischen Eigenschaften trotz Anwendung von F4 erhalten bleiben: Seien X, Y Zufallsvariablen für die beobachteten Koeffizienten bevor und nachdem F4 eine Nachricht eingebettet hat. $P(X = x)$ bezeichnet dann die Wahrscheinlichkeit, dass die JPEG-Kompression einen Koeffizienten mit dem Wert x erzeugt (der von F4 später weiterverarbeitet wird). $P(Y = y)$ steht für die Wahrscheinlichkeit, mit der ein Koeffizient mit dem Wert y von F4 nach dem Einbetten ausgegeben wird. Die beiden charakteristischen Eigenschaften (vgl. Abschnitt 2) können wir mit diesen Wahrscheinlichkeiten ausdrücken. Mit zunehmendem Betrag treten die Koeffizienten seltener auf (Ungleichung 5) und mit zunehmendem Betrag sinkt die Wahrscheinlichkeit für das Auftreten immer langsamer (Ungleichung 6).

$$P(X = 1) > P(X = 2) > P(X = 3) > P(X = 4) \tag{5}$$

$$P(X = 1) - P(X = 2) > P(X = 2) - P(X = 3) > P(X = 3) - P(X = 4) \tag{6}$$

```
int nextBitToEmbed = embeddedData.readBit();
for(int i=0; i<coeff.length; i++) {
    if (i%64 == 0) continue; // DC Koeffizienten und
    if (coeff[i] == 0) continue; // Nullen auslassen
    if (coeff[i] > 0) {
        if ((coeff[i]&1) != nextBitToEmbed)
            coeff[i]--; // Betrag um 1 verringern
    } else {
        if ((coeff[i]&1) == nextBitToEmbed)
            coeff[i]++; // Betrag um 1 verringern
    }
    if (coeff[i] != 0) { // erfolgreich eingebettet
        if (embeddedData.available()==0)
            break; // Ende der einzubettenden Nachricht
        nextBitToEmbed = embeddedData.readBit();
    }
}
```

Abbildung 10. Java-Quelltext der F4-Einbettungsfunktion (vereinfacht)

Wenn die Nachrichtenbits gleichverteilt sind, können wir schlussfolgern:

$$P(Y=1) = \frac{1}{2}P(X=1) + \frac{1}{2}P(X=2) \tag{7}$$

$$P(Y=2) = \frac{1}{2}P(X=2) + \frac{1}{2}P(X=3) \tag{8}$$

$$P(Y=3) = \frac{1}{2}P(X=3) + \frac{1}{2}P(X=4) \tag{9}$$

Wir bilden die Differenz der Gleichungen 7 und 8 bzw. 8 und 9 und erhalten Gleichung 10 bzw. 11.

$$P(Y=1) - P(Y=2) = \frac{1}{2}P(X=1) - \frac{1}{2}P(X=3) \tag{10}$$

$$P(Y=2) - P(Y=3) = \frac{1}{2}P(X=2) - \frac{1}{2}P(X=4) \tag{11}$$

Durch die erste charakteristische Eigenschaft (Ungleichung 5) wissen wir, dass die rechten Seiten der Gleichungen 10 und 11 positiv sind; somit liefern uns die linken Seiten die erste charakteristische Eigenschaft von Y:

$$P(Y=1) > P(Y=2) > P(Y=3) \tag{12}$$

Wenn wir $P(X=2) - P(X=3)$ zu Ungleichung 6 addieren, erhalten wir

$$P(X=1) - P(X=3) > P(X=2) - P(X=4) \tag{13}$$

Ungleichung 13 besagt, dass offensichtlich die rechte Seite von Gleichung 10 größer als die von Gleichung 11 ist. Der Vergleich der linken Seiten führt uns zur zweiten charakteristischen Eigenschaft von Y:

$$P(Y = 1) - P(Y = 2) > P(Y = 2) - P(Y = 3) \tag{14}$$

Analog können wir diese charakteristischen Eigenschaften auch für weitere von F4 modifizierte Werte zeigen.

Dieser Nachweis gilt übrigens auch für F3, wenn wir bei Schwund nicht wiederholt einbetten würden. Erst das wiederholte Einbetten mit F3 verletzt die Prämisse, dass die Bits der einzubettenden Nachricht gleichverteilt sind, denn F3 wiederholt nur Nullen.

F4 hingegen wiederholt beide steganographische Werte 0 und 1 gleichermaßen, da Koeffizienten mit dem Wert 1 (sorgt für Schwund beim Einbetten einer 0) und -1 (sorgt für Schwund bei einzubettender 1) gleich häufig erwartet werden.

Sicherlich beeinflusst F4 die Bildqualität des Steganogramms. Ein angenehmer Nebeneffekt wiegt diesen Nachteil jedoch auf: Der Schwund erzeugt – ähnlich wie die Quantisierung – mehr Koeffizienten mit dem Wert 0. Dadurch erzielt die nachfolgende Huffman-Kodierung höhere Kompressionsraten. F4 verringert neben der Qualität also auch die Dateigröße! Diesen Einfluss können wir leicht kompensieren, indem wir eine höhere Qualität im JPEG-Kompressor (d. h. einen geringeren Quantisierungsfaktor) einstellen.

6 F5

Im Gegensatz zu Trägermedien in Stromform (z. B. bei Videokonferenzen) stellen Dateien nur eine begrenzte steganographische Kapazität bereit. Eine eingebettete Nachricht benötigt oft nicht die volle Kapazität des Trägermediums (sofern sie hineinpasst). Somit bleibt ein Teil der Datei ungenutzt. Abbildung 11 zeigt, dass sich die Änderungen ($\times$) beim kontinuierlichen Einbetten auf den Dateianfang konzentrieren und der ungenutzte Rest am Dateiende zu finden ist.

Um Angriffe zu verhindern, sollte die Einbettungsfunktion das Trägermedium so gleichmäßig wie möglich ändern. Die Einbettungsdichte sollte an allen Stellen gleich groß sein.

6.1 Permutative Spreizung

Einige bekannte steganographische Algorithmen streuen die Nachricht über das gesamte Medium. Viele davon haben sehr schlechte Zeitkomplexität und werden langsamer, wenn die steganographische Kapazität fast

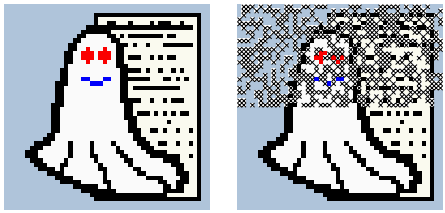


Abbildung 11. Kontinuierliches Einbetten konzentriert die Änderungen (×)

vollständig ausgenutzt werden soll. Spreizung ist einfach, wenn die Kapazität des Trägermediums vorher exakt bekannt ist.

Bei F4 können wir jedoch nicht vorhersagen, wie viel Schwund auftritt, denn das hängt davon ab, welches Bit an welcher Stelle eingebettet wird. Wir können lediglich einen Erwartungswert für die Kapazität im Steganogramm bestimmen.

Das bei F5 angewendete Spreizen mischt zunächst *alle* quantisierten Koeffizienten durch Anwendung einer Permutation. Die Anzahl der Koeffizienten ändert sich nicht durch den Schwund. (Lediglich die Anzahl der von Null verschiedenen – also steganographisch nutzbaren Koeffizienten – wird geringer.) F5 bettet dann in die permutierte Reihenfolge ein (kontinuierlich). Die Permutation ist abhängig von einem Schlüssel, der von einem Passwort abgeleitet wird. An die Huffmankodierung übergibt F5 die steganographisch veränderten Koeffizienten in ihrer ursprünglichen Reihenfolge. Der Empfänger kann die Permutation mit dem korrekten Schlüssel nachvollziehen. Die Permutation hat lineare Zeitkomplexität $O(n)$. Abbildung 12 zeigt, wie gleichmäßig die Permutation die Änderungen über das gesamte Bild verteilt. Die Bildpunkte in der Abbildung sind symbolisch zu verstehen. Sie stehen für JPEG-Koeffizienten, die ihrerseits mehrere Bildpunkte beeinflussen. Die steganographische Änderung eines Koeffizienten kann sich auf mehrere Bildpunkte auswirken.

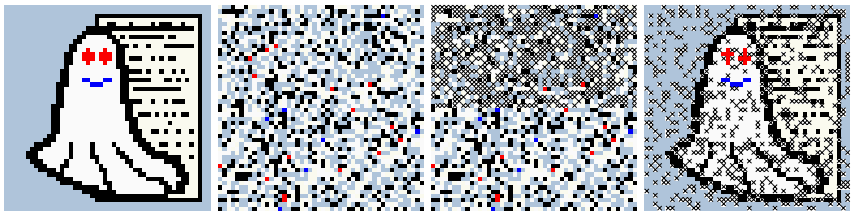


Abbildung 12. Permutatives Einbetten verteilt die Änderungen (×)

6.2 Matrixkodierung

Ron Crandall [1] führte die Matrixkodierung als eine neue Technik ein, mit der sich die Einbettungseffizienz erhöhen lässt. F5 ist möglicherweise die erste Implementierung der Matrixkodierung in einem steganographischen Algorithmus. Wenn der größte Teil der Kapazität eines Steganogramms ungenutzt bleibt, verringert die Matrixkodierung die Anzahl notwendiger Änderungen.

Betrachten wir zunächst den Fall ohne Matrixkodierung. Wenn wir eine gleichverteilte einzubettende Nachricht voraussetzen und auch gleichverteilte Werte an den Stellen im Trägermedium vorfinden, an denen sie eingebettet wird, dann erfordert nur die Hälfte aller Nachrichtenbits Änderungen im Trägermedium. Wir haben also eine Einbettungseffizienz von 2 Bits je Änderung. Durch den Schwund, der bei F4 entsteht, ist die Einbettungseffizienz noch etwas geringer, z. B. 1,5 Bits je Änderung. (Schwund bedeutet, dass manchmal geändert wird, ohne dass etwas einbettet wird. Das Ausmaß des Schwunds hängt vom Anteil JPEG-Koeffizienten mit dem Betrag 1 ab; vgl. Abschnitt 4.)

Wenn wir eine sehr kurze Nachricht einbetten, die nur 217 Bytes (1736 Bits) enthält, dann ändert F4 (ohne Matrixkodierung) 1157 Stellen im Expo-Bild. F5 kann die gleiche Nachricht dank Matrixkodierung mit nur 459 Änderungen einbetten. Das ist weniger als die Hälfte und entspricht einer Einbettungseffizienz von 3,8 Bits je Änderung.

Die Matrixkodierung fasst mehrere änderbare Stellen zu einem Block (Kodewort) zusammen und bettet darin einige Nachrichtenbits ein. Was im Detail passiert, zeigt das folgende Beispiel. Die Matrixkodierung setzen wir auf einen herkömmlichen steganographischen Algorithmus auf, der die Änderungen durchführt. Wir wollen zwei Bits x_1, x_2 in drei veränderbare Bitstellen a_1, a_2, a_3 einbetten und davon maximal eine ändern. Dabei können die folgenden vier Fälle auftreten:

$$\begin{aligned}
&x_1 = a_1 \oplus a_3,\ x_2 = a_2 \oplus a_3 \Rightarrow \text{nichts ändern}\\
&x_1 \neq a_1 \oplus a_3,\ x_2 = a_2 \oplus a_3 \Rightarrow a_1 \text{ ändern}\\
&x_1 = a_1 \oplus a_3,\ x_2 \neq a_2 \oplus a_3 \Rightarrow a_2 \text{ ändern}\\
&x_1 \neq a_1 \oplus a_3,\ x_2 \neq a_2 \oplus a_3 \Rightarrow a_3 \text{ ändern.}
\end{aligned}$$

In allen vier Fällen müssen wir höchstens ein Bit ändern. Im Allgemeinen haben wir ein Kodewort $\boldsymbol{a}$ mit n veränderbaren Bitstellen und k einzubettende Nachrichtenbits in $\boldsymbol{x}$. Wir haben eine Hashfunktion f, die aus einem n-stelligen Kodewort k Bits extrahieren kann. Mit der Matrixkodierung finden wir also zu jedem $\boldsymbol{a}$ und $\boldsymbol{x}$ ein passendes Kodewort $\boldsymbol{a'}$ mit $\boldsymbol{x} = f(\boldsymbol{a'})$, so dass die Anzahl der nötigen Änderungen (Hammingdistanz)

ein bestimmtes Maximum nicht übersteigt:

$$d(\boldsymbol{a}, \boldsymbol{a'}) \leq d_{\max} \tag{15}$$

Wir bezeichnen diesen Kode durch Tripel $(d_{\max}, n, k)$: Ein n-stelliges Kodewort wird höchstens in $d_{\max}$ Stellen geändert, um k Nachrichtenbits einzubetten. Folglich wird unser konkretes Beispiel mit $(1, 3, 2)$ bezeichnet. Im Algorithmus F5 ist die Matrixkodierung nur für den Fall $d_{\max} = 1$ implementiert. Die Kodewortlänge für den $(1, n, k)$-Kode hat $n = 2^k - 1$ Stellen. Bei steganographischen Algorithmen ohne Schwund erhalten wir eine Änderungsdichte

$$D(k) = \frac{1}{n+1} = \frac{1}{2^k} \tag{16}$$

und eine Einbettungsrate

$$R(k) = \frac{k}{n} = \frac{1}{n} \cdot \operatorname{ld}(n+1) = \frac{k}{2^k - 1}. \tag{17}$$

Mit der Änderungsdichte und der Einbettungsrate können wir die Einbettungseffizienz $W(k)$ definieren. Sie gibt an, wie viele Bits der steganographischen Nachricht je Änderung im Mittel eingebettet werden können:

$$W(k) = \frac{R(k)}{D(k)} = \frac{2^k}{2^k - 1} \cdot k. \tag{18}$$

Die Einbettungseffizienz ist also für den $(1, n, k)$-Kode stets größer als k. Tabelle 1 verdeutlicht, dass die Einbettungsrate mit zunehmender Einbettungseffizienz sinkt. Eine hohe Einbettungseffizienz können wir daher nur mit sehr kurzen Nachrichten erzielen.

Tabelle 1. Zusammenhang zwischen Änderungsdichte und Einbettungsrate

k	n	Änderungsdichte	Einbettungsrate	Einbettungseffizienz
1	1	50,00 %	100,00 %	2
2	3	25,00 %	66,67 %	2,67
3	7	12,50 %	42,86 %	3,43
4	15	6,25 %	26,67 %	4,27
5	31	3,12 %	16,13 %	5,16
6	63	1,56 %	9,52 %	6,09
7	127	0,78 %	5,51 %	7,06
8	255	0,39 %	3,14 %	8,03
9	511	0,20 %	1,76 %	9,02

Tabelle 2 gibt die Abhängigkeiten zwischen den Nachrichtenbits x_i und den Bitstellen des geänderten Kodeworts a'_j an. In Tabelle 2 ordnen wir die Abhängigkeiten in Spalte a'_j so zu, dass sie der Binärkodierung[2] von j entsprechen. Dann können wir die Hashfunktion

$$f(\boldsymbol{a}) = \bigoplus_{i=1}^{n} a_i \cdot i \tag{19}$$

besonders schnell bestimmen, ebenso schnell finden wir die zu ändernde Stelle[3]

$$s = \boldsymbol{x} \oplus f(\boldsymbol{a}). \tag{20}$$

Wir erhalten das geänderte Kodewort

$$\boldsymbol{a'} = \begin{cases} \boldsymbol{a}, & \text{falls } s = 0 \ (\Leftrightarrow \boldsymbol{x} = f(\boldsymbol{a})), \\ (a_1, a_2, \ldots, \neg a_s, \ldots, a_n) & \text{sonst.} \end{cases} \tag{21}$$

Tabelle 2. Abhängigkeit ($\times$) zwischen den Nachrichtenbits x_i Kodewortbits a'_j

$f(\boldsymbol{a'})$	a'_1	a'_2	a'_3
x_1	$\times$		$\times$
x_2		$\times$	$\times$

$f(\boldsymbol{a'})$	a'_1	a'_2	a'_3	a'_4	a'_5	a'_6	a'_7
x_1	$\times$		$\times$		$\times$		$\times$
x_2		$\times$	$\times$			$\times$	$\times$
x_3				$\times$	$\times$	$\times$	$\times$

Für jede einzubettende Nachricht und jedes Trägermedium, das hinreichende Kapazität zur Verfügung stellt, können wir einen optimalen Parameter k finden, bei dem die Nachricht gerade noch in das Trägermedium passt. Wenn wir z. B. 1000 Bits in ein Trägermedium mit einer Kapazität von 50000 Bits einbetten wollen, dann beträgt die nötige Einbettungsrate $R = 1000 : 50000 = 2\,\%$. Dieser Wert liegt in Tabelle 1 zwischen $R(k = 8)$ und $R(k = 9)$. Wir wählen $k = 8$ und können $50000 : 255 = 196$ Kodewörter der Länge $n = 255$ einbetten. Der $(1, 255, 8)$-Kode könnte $196 \cdot 8 = 1568$ Bits einbetten (mit max. 196 Änderungen). Würden wir $k = 9$ wählen, könnten wir die Nachricht nicht vollständig einbetten.

6.3 Erhaltung der charakteristischen Eigenschaften

Ein formaler Nachweis der Sicherheit steganographischer Algorithmen gestaltet sich extrem schwierig. Im Gegensatz zur Kryptographie, wo wir

[2] 0=„ “ (nichts) und 1=„$\times$“

[3] Den resultierenden Bitvektor $\boldsymbol{s}$ interpretieren wir als natürliche Zahl, die den Index der zu ändernden Stelle angibt.

informationstheoretische Beziehungen aufstellen können, besteht bei der Steganographie die Schwierigkeit, Eigenschaften wie „Erkennbarkeit“ formalisieren zu müssen. Deshalb beschränken wir uns hier auf den Nachweis, dass F5 gegen alle *bekannten* Angriffe resistent ist.

Der in [5] vorgestellte Angriff auf Jsteg weist statistische Abhängigkeiten im Steganogramm nach, die auf das Überschreiben niederwertigster Bits zurückzuführen sind. Das ist bei F4 und F5 nicht der Fall, da eine andere Einbettungsoperation verwendet wird. F4 bewahrt die charakteristischen Eigenschaften und gleicht keine Häufigkeiten aus (siehe Abschnitt 5). Das lässt sich auch für F5 zeigen: Sei $0 \leq \alpha \leq 1$ der steganographisch genutzte Anteil verwendbarer Koeffizienten.[4] Wenn wir die Gleichungen 7, 8 und 9 anpassen, funktioniert der Nachweis auch für F5:

$$P(Y=1) = (1-\frac{\alpha}{2})P(X=1) + \frac{\alpha}{2}P(X=2) \tag{22}$$

$$P(Y=2) = (1-\frac{\alpha}{2})P(X=2) + \frac{\alpha}{2}P(X=3) \tag{23}$$

$$P(Y=3) = (1-\frac{\alpha}{2})P(X=3) + \frac{\alpha}{2}P(X=4) \tag{24}$$

Wir bilden die Differenz der Gleichungen 22 und 23 bzw. 23 und 24 und erhalten Gleichung 25 bzw. 26.

$$P(Y=1) - P(Y=2) = \left(1-\frac{\alpha}{2}\right)\big(P(X=1) - P(X=2)\big) + \frac{\alpha}{2}P(X=3) \tag{25}$$

$$P(Y=2) - P(Y=3) = \left(1-\frac{\alpha}{2}\right)\big(P(X=2) - P(X=3)\big) + \frac{\alpha}{2}P(X=4) \tag{26}$$

Durch die erste charakteristische Eigenschaft (Ungleichung 5) wissen wir, dass die rechten Seiten der Gleichungen 25 und 26 positiv sind; somit liefern uns die linken Seiten die erste charakteristische Eigenschaft von Y:

$$P(Y=1) > P(Y=2) > P(Y=3) \tag{27}$$

Aus den charakteristischen Eigenschaften von X (vgl. Ungleichungen 5 und 6)

$$P(X=1) - P(X=2) > P(X=2) - P(X=3)$$
$$P(X=3) > P(X=4)$$

folgt, dass die rechte Seite von Gleichung 25 größer als die von Gleichung 26 ist. Der Vergleich der linken Seiten führt uns zur zweiten charakteristischen Eigenschaft von Y:

$$P(Y=1) - P(Y=2) > P(Y=2) - P(Y=3) \tag{28}$$

Analog können wir diese charakteristischen Eigenschaften auch für weitere von F5 modifizierte Werte zeigen.

[4] F4 kann als der Spezialfall $\alpha = 1$ aufgefasst werden.

```
 1 F5Random random = new F5Random(password.getBytes());
 2 Permutation permutation = new Permutation(coeff.length, random);
 3 int k = determineCodeParameter(coeff, embeddedData.available());
 4 n = (1<<k)-1;
 5 if (n > 1) { // verwende Matrix-Kodierung (1, n, k)
 6     int kBitsToEmbed; int extractedBit; int hash; int s;
 7     int[] codeWord = new int[n]; int startOfN=0; int endOfN=0;
 8 embeddingLoop:
 9     for (;;) { // Endlosschleife
12         if (embeddedData.available()==0)
14             break; // Nachrichtenende, verlasse Endlosschleife
15         kBitsToEmbed = embeddedData.readBits(k);
17         do { // bette k Bits ein
18             j = startOfN;
21             for (i=0; i<n; j++) { // n-stelliges Kodewort füllen
22                 if (j>=coeff.length) // Kapazität erschöpft
24                     break embeddingLoop; // beende Endlosschleife
26                 shuffledIndex = permutation.getShuffled(j);
27                 if (shuffledIndex%64 == 0) continue; // skip DC
28                 if (coeff[shuffledIndex] == 0) continue; // skip 0
29                 codeWord[i++]=shuffledIndex;
30             }
31             endOfN = j; // merke Kodewort-Ende
32             hash = 0;
33             for (i=0; i<n; i++) {
34                 if (coeff[codeWord[i]] > 0)
35                     extractedBit = coeff[codeWord[i]]&1;
36                 else
37                     extractedBit = 1-(coeff[codeWord[i]]&1);
38                 if (extractedBit == 1)
39                     hash ^= i+1;
40             }
41             s = hash ^ kBitsToEmbed;
42             if (s==0) break; // keine Änderung nötig
44             if (coeff[codeWord[--s]]>0) // dekrementiere Betrag
45                 coeff[codeWord[s]]--;
46             else
47                 coeff[codeWord[s]]++;
48         } while (coeff[codeWord[s]]==0); // wiederhole bei Schwund
49         startOfN = endOfN; // weiter mit neuen Koeffizienten
50     }
51 } else ... // ohne Matrixkodierung
```

Abbildung 13. Java-Quelltext der Einbettungsfunktion von F5 (vereinfacht)

6.4 Implementierung

Der Algorithmus F5 hat die folgende Grobstruktur (die Zeilennummern beziehen sich auf den in Abbildung 13 angegebenen Quelltext):

1. Starte die JPEG-Kompression. Halte nach der Quantisierung der Koeffizienten.
2. Initialisiere einen kryptographisch starken Zufallszahlengenerator mit dem vom Passwort abgeleiteten Schlüssel. (Zeile 1)
3. Instanziiere die Permutation (zwei Parameter: Zufallsgenerator und Anzahl aller JPEG-Koeffizienten[5]). (Zeile 2)
4. Bestimme den Parameter k aus der Kapazität C des Trägermediums und der Länge der einzubettenden Nachricht. (Zeile 3)
5. Berechne die Kodewortlänge $n = 2^k - 1$. (Zeile 4)
6. Bette die geheime Nachricht mit $(1, n, k)$-Matrixkodierung ein. (Zeilen 5–51)
   (a) Fülle einen n-stelligen Puffer mit den Indizes von Null verschiedener JPEG-Koeffizienten. (Zeilen 18–31)
   (b) Bilde den k-stelligen Hashwert des Puffers nach Gleichung 19. (Zeilen 32–40)
   (c) Addiere die nächsten k Bits der Nachricht bitweise (xor) zum Hashwert. (Zeile 41)
   (d) Falls die Summe 0 ist, wird der Puffer nicht verändert. Ansonsten gibt die um 1 verringerte Summe s den Index $0 \ldots (n-1)$ im Puffer an, dessen Element an dieser Stelle betragsmäßig um 1 verringert wird. (Der Puffer entspricht nun $\boldsymbol{a'}$ in Gleichung 21.) (Zeilen 44–47)
   (e) Teste, ob Schwund aufgetreten ist, d. h. ob beim Einbetten der Wert Null entstanden ist. Wenn Schwund aufgetreten ist, dann bereinige den Puffer, d. h. beseitige die 0 durch Wiederholung von Schritt (6a). Wenn kein Schwund aufgetreten ist, dann lies neue Werte in den Puffer. In Zeile 49 wird der Lesebeginn `startOfN` hinter das Ende `endOfN` des alten Pufferinhalts gelegt. Falls noch einzubettende Daten vorhanden sind (Zeile 12) wird mit Schritt (6a) fortgesetzt.
7. Setze die JPEG-Komprimierung fort (Huffman-Kodierung usw.).

7 Schlussfolgerung und Ausblick

Viele steganographische Algorithmen bieten eine hohe Kapazität für versteckte Nachrichten, sind aber durch visuelle und statistische Angriffe

[5] Die JPEG-Koeffizienten mit dem Wert 0 sind hier inbegriffen, obwohl sie nicht steganographisch verwendet werden.

leicht nachweisbar. Einige Programme widerstehen diesen Angriffen, bieten jedoch nur eine sehr geringe Kapazität. Der Algorithmus F4 vereint beide Vorzüge: Er ist resistent gegenüber visuellen und statistischen Angriffen und bietet gleichzeitig eine sehr hohe Kapazität.

Zu den Vorzügen von F4 fügt F5 noch eine erhöhte Einbettungseffizienz und die permutative Spreizung hinzu, was für geringere und gleichmäßigere Änderungen sorgt. F5 bietet einen steganographischen Anteil von über 13 % der JPEG-Dateigröße (siehe Tabelle 3). Bitte fassen Sie dieses Ergebnis als freundliche Provokation für Sicherheitsanalytiker auf. Durch die Veröffentlichung des Algorithmus [7] erhofft sich der Autor erhöhtes Vertrauen oder interessante Angriffe. Sollten Sie letzteres befürchten, so sei darauf hingewiesen, dass sich die Einbettungsrate bei F5 durch Verwendung kürzerer Nachrichten oder größerer Trägermedien verringern und somit die Stichprobe des Angreifers beliebig „verwässern“ lässt.

Tabelle 3. Vergleich von verschiedenen mit F5 erzeugten Dateien

Dateiname	Dateigröße (Bytes)	eingebettete Nachricht (Bytes)	Verhältnis von Nachricht zu Steganogrammgröße	Einbettungseffizienz	JPEG-Qualität
expo.bmp	1 562 030	0	(Trägermedium)	—	—
expo80.jpg	129 879	0	—	—	80 %
ministeg.jpg	129 760	213	0,2 %	3,8	80 %
maxisteg.jpg	115 685	15 480	13,4 %	1,5	80 %
expo75.jpg	114 712	0	—	—	75 %

Literatur

1. Ron Crandall: Some Notes on Steganography. Gesendet an die „Steganography Mailing List“, 1998. http://os.inf.tu-dresden.de/~westfeld/crandall.pdf
2. Andy C. Hung: PVRG-JPEG Codec 1.1, Stanford University, 1993. http://archiv.leo.org/pub/comp/os/unix/graphics/jpeg/PVRG
3. Fabien Petitcolas: MP3Stego, 1998. http://www.cl.cam.ac.uk/~fapp2/steganography/mp3stego
4. Derek Upham: Jsteg, 1997. z. B. http://www.tiac.net/users/korejwa/jsteg.htm
5. Andreas Westfeld: Angriffe auf steganographische Systeme, in Rainer Baumgart, Kai Rannenberg, Dieter Wähner, Gerhard Weck (Hrsg.): Verlässliche Informationssysteme (IT-Sicherheit an der Schwelle des neuen Jahrtausends), DuD-Fachbeiträge, Vieweg Braunschweig, 1999. S. 263–286.

6. Andreas Westfeld, Gritta Wolf: Steganography in a Video Conferencing System, in David Aucsmith (Hrsg.): Information Hiding, LNCS 1525, Springer-Verlag Berlin Heidelberg 1998. S. 32–47.
7. Andreas Westfeld: The Steganographic Algorithm F5, 1999. http://wwwrn.inf.tu-dresden.de/~westfeld/f5.html
8. Andreas Westfeld: Unsichtbare Botschaften. In c't Magazin für Computertechnik 9/2001. S. 170–181.
9. Jan Zöllner, Hannes Federrath, Andreas Pfitzmann, Andreas Westfeld, Guntram Wicke, Gritta Wolf: Über die Modellierung steganographischer Systeme, in Günter Müller, Kai Rannenberg, Manfred Reitenspieß, Helmut Stiegler (Hrsg.): Verlässliche IT-Systeme (Zwischen Key Escrow und elektronischem Geld), DuD-Fachbeiträge, Vieweg Braunschweig, 1997. S. 211–223.