



סטגנוגרפיה מתקדמת - חלק א': תמונות דיגיטליות

מאת עידן קובליק

הקדמה

סטגנוגרפיה היא אחת השיטות הוותיקות ביותר להסתרת מידע, והיא ממשיכה להיות רלוונטית גם היום - בתמונות, בקבצי שמע ואפילו בתעבורת רשת. במאמר זה נתמקד בעולם התמונות הדיגיטליות: נבין כיצד תמונה מיוצגת במחשב, כיצד ניתן להטמיע בה מידע מבלי שהדבר יורגש, אילו אלגוריתמים משמשים לכך, וכיצד ניתן לתקוף ולזהות הטמעה כזו באמצעות ניתוח סטטיסטי. נעבור בין תמונות PNG לתמונות JPEG, ונראה מדוע כל פורמט דורש גישה שונה לחלוטין.

מאמר זה הינו המאמר הפותח סדרת מאמרים שאפרסם בנושא סטגנוגרפיה דיגיטלית.

מהי סטגנוגרפיה

סטגנוגרפיה היא שיטה להסתרת מידע בתוך אובייקט אחר, כך שעצם קיומו של המידע הנסתר אינו גלוי למתבונן. ניתן להטמיע מידע בתוך אובייקטים דיגיטליים או פיזיים, כגון תמונות, קבצי שמע, סרטונים, מסמכים ואף תקשורת ברשת. בניגוד להצפנה, שבה המידע הופך לבלתי קריא אך ברור שקיים מידע מוגן, בסטגנוגרפיה המטרה היא להסתיר את עצם קיומו של המידע. לדוגמה, ניתן להטמיע הודעה בתוך תמונה על ידי שינוי עדין של הנתונים המרכיבים אותה, כך שהתמונה תיראה רגילה לחלוטין למי שפותח אותה, אך תוכנה ייעודית תוכל לחלץ ממנה את המידע שהוסתר.

כיצד זה עובד?

תהליך הסטגנוגרפיה מורכב בדרך כלל משני שלבים עיקריים:

- **הטמעה** - הכנסת המידע הסודי לתוך אובייקט שנראה תמים, הנקרא Carrier.
 - **חילוץ** - שימוש במפתח או באלגוריתם המתאים כדי להוציא את המידע שהוטמע.
- ככל שהמידע מוסתר בצורה טובה יותר, כך קשה יותר לזהות שהאובייקט מכיל מידע נוסף, תוך שמירה על איכות האובייקט המקורי.



מה ההבדל בין קריפטוגרפיה לבין סטגנוגרפיה?

קריפטוגרפיה וסטגנוגרפיה הן שתי שיטות המשמשות להגנה על מידע, אך הן עושות זאת בדרכים שונות.

בקריפטוגרפיה, המידע עצמו מוצפן והופך לטקסט בלתי קריא ללא המפתח המתאים. עם זאת, ניתן לראות שקיים מידע מוצפן. לדוגמה, אם נשלח קובץ מוצפן, מי שיירט אותו יוכל לדעת שמדובר בנתונים מוצפנים, גם אם לא יוכל לקרוא אותם.

בסטגנוגרפיה, לעומת זאת, המטרה היא להסתיר את עצם קיומו של המידע. המידע מוטמע בתוך אובייקט אחר, כגון תמונה, קובץ שמע או סרטון, כך שלרוב לא ניתן להבחין בכך שהאובייקט מכיל מידע נסתר.

לדוגמה, בקריפטוגרפיה ניתן להצפין את המילה HELLO למחרוזת שגראית אקראית. בסטגנוגרפיה ניתן להטמיע את אותה מילה בתוך תמונה, כך שמי שיצפה בתמונה יראה תמונה רגילה לחלוטין ולא יידע שמוסתרת בה הודעה.

לכן, ניתן לסכם את ההבדל כך:

- קריפטוגרפיה מסתירה את **תוכן** המידע.
- סטגנוגרפיה מסתירה את **עצם קיומו** של המידע.

ניתן גם לשלב בין שתי השיטות. במקרה כזה, תחילה מצפינים את המידע באמצעות קריפטוגרפיה, ולאחר מכן מטמיעים את המידע המוצפן בתוך אובייקט באמצעות סטגנוגרפיה. כך גם אם מישהו יגלה את המידע הנסתר, הוא עדיין יצטרך לפענח אותו כדי להבין את תוכנו.

כיצד ניתן להסתיר מידע בתוך תמונות?

כדי להבין כיצד ניתן להסתיר מידע בתוך תמונה, יש להבין תחילה כיצד תמונה דיגיטלית מיוצגת במחשב.

תמונה דיגיטלית מורכבת ממספר גדול של נקודות הנקראות פיקסלים (Pixels). כל פיקסל מכיל ערכים מספריים המתארים את הצבע שלו. במודל הצבע הנפוץ RGB, כל פיקסל מורכב משלושה ערוצים (Channels):

- R - ערוץ האדום (Red)
- G - ערוץ הירוק (Green)
- B - ערוץ הכחול (Blue)

בדרך כלל, כל ערוץ משתמש ב-8 ביטים, ולכן הוא יכול לקבל ערך בטווח:

$$0 \leq C \leq 255$$

כאשר C הוא הערך של אחד מערוצי הצבע. לכן, פיקסל RGB אחד דורש:

$$bits = 8 \cdot 3 = 24$$

ומכיוון שכל אחד משלושת הערוצים יכול לקבל 2^8 ערכים שונים, אזי מספר הצבעים האפשריים עבור פיקסל RGB הוא:

$$2^8 \times 2^8 \times 2^8 = 2^{24}$$



[תמונה שעברה הליך הטמעה של מידע נסתר (תצליחו למצוא מהו?)]

ערוץ Alpha

בנוסף לערוצי R , G ו- B , תמונות מסוימות מכילות ערוץ נוסף הנקרא Alpha, המסומן באות A , שגם טווח הערכים שלו הוא:

$$0 \leq A \leq 255$$

כאשר 0 מייצג שקיפות מלאה ו-255 מייצג אטימות מלאה.

ואם נעשה את המתמטיקה, זה אומר שעכשיו כל פיקסל דורש:

$$8 \times 4 = 32 \text{ bits}$$

שימו לב! מומלץ לא להשתמש בערוץ זה בזמן שימוש בסטגנוגרפיה, כיוון שערוץ זה יוצר הבדל פורנזי ניכר ולכן ניתן יהיה להבחין אם תמונה עברה הטמעת מידע כלשהו בתוכה.



תקיפת תוכנות באמצעות תמונות סטגנוגרפיות

עד כה התמקדנו באופן שבו ניתן להסתיר מידע בתוך תמונה. אולם תמונה דיגיטלית אינה רק אוסף של פיקסלים - היא גם קובץ שנקרא ומעובד על ידי תוכנות שונות. עובדה זו מאפשרת לתוקף להשתמש בתמונה עצמה כווקטור תקיפה.

לדוגמה, ניתן ליצור תמונה שתיראה תקינה לחלוטין למשתמש, אך תכיל בתוכה מידע או מבנה שינסה לנצל חולשה בתוכנה שאחראית על פתיחת התמונה. חולשות כאלה עשויות להימצא במפענח (Decoder), במנתח הקובץ (Parser) או בספריית עיבוד התמונות שבה משתמשת התוכנה.

לדוגמה, הקוד הבא מדגים באופן פשטני כיצד חוסר בבדיקת גבולות עלול ליצור חולשת Buffer Overflow:

```
#include <stdio.h>
#include <string.h>

void load_image(const unsigned char *data, size_t size) {
    char buffer[16];

    /* Vulnerable: no check that the data fits into buffer */
    memcpy(buffer, data, size);

    printf("Image loaded\n");
}

int main(void) {
    unsigned char image_data[64] = {0};
    load_image(image_data, sizeof(image_data));

    return 0;
}
```

הבעיה נמצאת בכך שהפונקציה מקבלת כמות מידע גדולה יותר מהשטח שהוקצה ל-buffer. בתוכנת עיבוד תמונות אמיתית, מצב דומה עלול להתרחש בעת עיבוד שדות, מטא-דאטה או מבנים פנימיים של קובץ תמונה. לכן, תוקף יכול ליצור קובץ תמונה מיוחד אשר מצד אחד נראה כמו תמונה רגילה, ומצד שני מכיל קלט שנבנה במיוחד כדי להפעיל את החולשה בתוכנה המעבדת אותו.

בהקשר של סטגנוגרפיה, הדבר מעניין במיוחד: ניתן להשתמש בתמונה כ-Carrier עבור מידע נסתר, כאשר התמונה עצמה משמשת גם כקלט לתוכנה פגיעה. כלומר, התקיפה אינה חייבת להתבסס על שבירת מנגנון הסטגנוגרפיה או על חילוץ המידע הנסתר - היא יכולה להתבסס על התוכנה שמקבלת ומעבדת את התמונה.

חשוב להבחין בין שני סוגי התקיפה:

- **תקיפה של הסטגנוגרפיה** - ניסיון לזהות את קיום המידע הנסתר או לחלץ אותו מהתמונה.
 - **תקיפה באמצעות התמונה** - ניצול חולשה בתוכנה או בספרייה שמעבדת את התמונה.
- הבחנה זו מראה שתמונה סטגנוגרפית יכולה לשמש לא רק כאמצעי להסתרת מידע, אלא גם כווקטור להעברת קלט זדוני לתוכנות אחרות.



Least Significant Bit (LSB) Replacement

אחת השיטות הפשוטות והנפוצות ביותר בסטגנוגרפיה של תמונות היא שיטת ה-Least Significant Bit, או בקיצור LSB. הרעיון מבוסס על העובדה שהביט האחרון של ערך צבע משפיע מעט מאוד על הצבע המוצג. לדוגמה, נבחן את הערכים הבאים:

$$11001010_2 = 202$$

$$11001011_2 = 203$$

למרות שהערכים שונים בביט אחד, ההבדל ביניהם הוא 1 בלבד. לכן, שינוי הביט האחרון של ערוץ צבע בדרך כלל אינו מורגש בעין האנושית.

במקום להשתמש בביט האחרון כדי לייצג את צבע הפיקסל, ניתן להשתמש בו כדי לאחסן מידע. לדוגמה, נניח שהערכים של שלושת ערוצי הצבע של פיקסל מסוים הם:

$$R = 11001010$$

$$G = 10110101$$

$$B = 01101100$$

אם הביטים שאנו רוצים להסתיר הם 1,0,1, נוכל לשנות את הביט האחרון של כל ערוץ:

$$11001010 \rightarrow 11001011$$

$$10110101 \rightarrow 10110100$$

$$01101100 \rightarrow 01101101$$

כך קיבלנו את הביטים:

$$101$$

הפיקסל השתנה מעט מאוד, אך כעת הוא מכיל מידע שהאלגוריתם שלנו יכול לקרוא.

במקרה של RGB, ניתן להשתמש בביט האחרון של כל אחד משלושת הערוצים. כלומר, כל פיקסל יכול להכיל עד:

$$3 \text{ bits}$$

לכן, תמונה בגודל 1920×1080 פיקסלים יכולה להכיל באופן תאורטי:

$$1920 \times 1080 \times 3 = 6,220,800 \text{ bits}$$

עם זאת, קיבולת זו היא קיבולת תאורטית בלבד. בפועל, ככל שמכניסים יותר מידע ומשנים יותר פיקסלים, כך גדל הסיכוי שהשינויים יהיו ניתנים לזיהוי באמצעות ניתוח סטטיסטי או פורנזי.



כיצד יודעים איפה נמצא המידע?

לאחר שהחלטנו להשתמש ב-LSB, עדיין קיימת בעיה: כיצד תוכנת החילוץ תדע היכן לחפש את המידע? אין דרך אחת נכונה לעשות זאת, וניתן לתכנן את פורמט ההטמעה במספר דרכים.

שיטה ראשונה - כתיבה רציפה מהפיקסל הראשון

השיטה הפשוטה ביותר היא להתחיל מהפיקסל הראשון בתמונה ולכתוב את המידע באופן רציף. לדוגמה, אם ההודעה היא:

TEST

ניתן להמיר אותה לביטים:

$$T = 01010100$$

$$E = 01000101$$

$$S = 01010011$$

$$T = 01010100$$

לאחר מכן מתחילים מהפיקסל הראשון ומכניסים את הביטים של ההודעה לתוך ה-LSB של ערוצי הצבע. החילוץ מתבצע באותה צורה: מתחילים מהפיקסל הראשון, קוראים את ה-LSB של הערוצים, ומרכיבים מחדש את הבייטים.

היתרון של השיטה הוא הפשטות שלה. אין צורך במידע נוסף כדי לדעת היכן להתחיל.

החסרון הוא שהמבנה צפוי מאוד. מי שיודע שהאלגוריתם משתמש ב-LSB ושהמידע מתחיל מהפיקסל הראשון, יכול לנסות לחלץ אותו בקלות יחסית.

שיטה שנייה - Header בתחילת התמונה

אפשרות מתקדמת יותר היא להקדיש חלק קטן מהמידע בתחילת התמונה ל-Header. ה-Header יכול להכיל מידע שיסייר לתוכנת החילוץ כיצד למצוא את המידע האמיתי.

לדוגמה, ניתן להגדיר Header המכיל:

- אורך המידע.
- סוג המידע.
- האלגוריתם שבו נעשה שימוש.
- מספר הביטים המשמשים מכל ערוץ.
- מיקום תחילת המידע.
- מפתח או מזהה המשמשים לבחירת מיקום המידע.

לדוגמה, המבנה יכול להיראות כך:

[HEADER][PAYLOAD]

כאשר ה-Header נמצא בתחילת רצף המידע וה-Payload הוא המידע הסודי עצמו. כך תוכנת החילוץ יכולה לקרוא תחילה את ה-Header, להבין כיצד התמונה נבנתה, ולאחר מכן לחלץ את ה-Payload בהתאם להוראות. לדוגמה:

[LENGTH = 1000][METHOD = LSB][DATA = ...]

שיטה שלישית - פיזור המידע באמצעות מפתח

במקום לכתוב את המידע בפיקסלים רצופים, ניתן לפזר אותו על פני התמונה. לדוגמה, במקום:

$P_0, P_1, P_2, P_3, P_4, \dots$

ניתן להשתמש בסדר:

$P_{42}, P_{917}, P_{153}, P_{2048}, \dots$

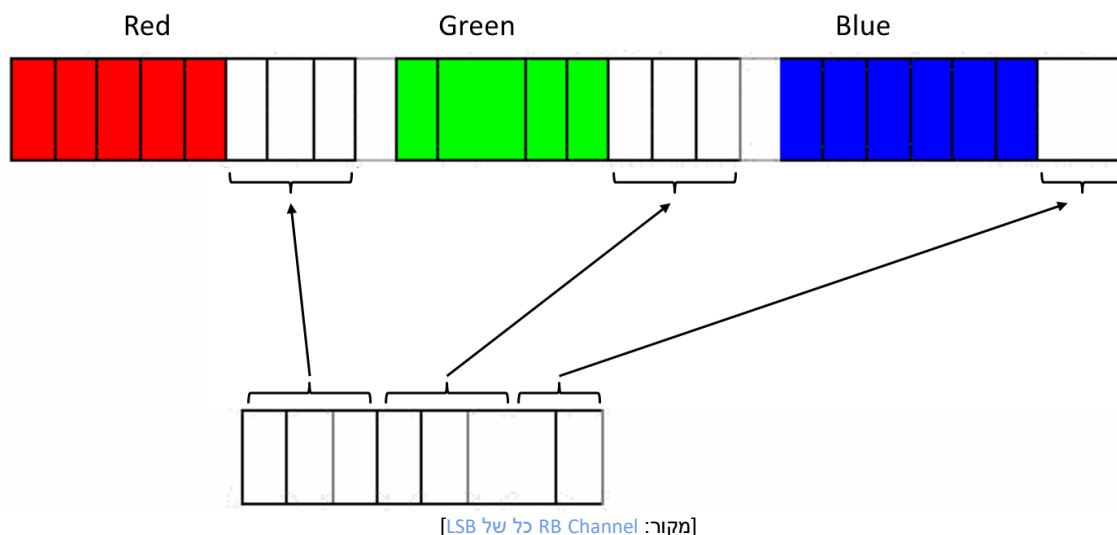
הסדר יכול להיקבע באמצעות מחולל פסאודו-אקראי (PRNG) שמקבל מפתח. במקרה כזה, מי שאין ברשותו את המפתח אינו יודע באילו פיקסלים לחפש את המידע. לדוגמה:

$KEY \rightarrow PRNG \rightarrow [42, 917, 153, 2048, \dots]$

תוכנת ההטמעה והתוכנה שמחלצת את המידע משתמשות באותו מפתח ולכן מייצרות את אותו סדר פיקסלים. שיטה זו מקשה משמעותית על חילוץ המידע כאשר לא יודעים את המפתח.

כמובן שאפשר ואף רצוי לשלב בין השיטות, כדי להחביא את המידע הנסתר כמה שיותר טוב.

שימו לב! כדאי להשתמש ב-LSB בעיקר בתמונות PNG, מכיוון ש-PNG הוא פורמט ללא איבוד מידע (Lossless), ולכן שינוי ביטי ה-LSB נשמר לאחר שמירת התמונה.



LSB Matching

שיטה נוספת המבוססת על ביט ה-LSB היא LSB Matching. בניגוד ל-LSB Replacement, שבה פשוט מחליפים את ביט ה-LSB של ערוץ הצבע בביט שאנו רוצים להטמיע, ב-LSB Matching משנים את ערך הפיקסל ב-1 כאשר ה-LSB הנוכחי אינו מתאים למידע שאנו רוצים להטמיע.

לדוגמה, נניח שערך ערוץ הצבע הוא:

$$100 = 01100100$$

והביט שאנו רוצים להטמיע הוא 1. ב-LSB Replacement פשוט נחליף את הביט האחרון:

$$01100100 \rightarrow 01100101$$

ולכן הערך משתנה מ-100 ל-101. ב-LSB Matching, אם ה-LSB אינו מתאים, ניתן להגדיל או להקטין את הערך באופן אקראי:

$$100 \rightarrow 101 \quad \text{או} \quad 100 \rightarrow 99$$

בשני המקרים ה-LSB הופך ל-1, אך השינוי בערך המקורי הוא קטן בלבד. אם, לעומת זאת, הערך המקורי כבר מכיל את הביט הרצוי, אין צורך לבצע שום שינוי. לדוגמה:

$$01100101 = 101$$

אם הביט שאנו רוצים להטמיע הוא 1, הערך כבר מתאים ולכן נשאיר אותו ללא שינוי.

מה ההבדל בין LSB Replacement ל-LSB Matching?

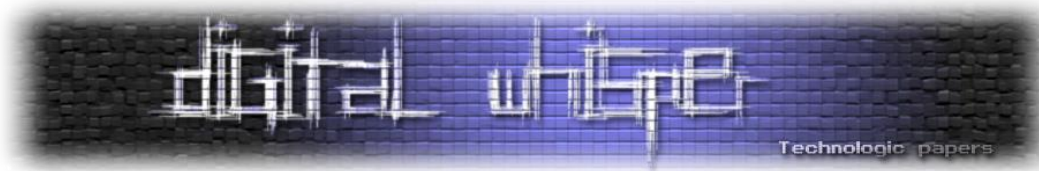
ב-LSB Replacement, הביט האחרון מוחלף ישירות בביט המידע:

$$x \rightarrow x'$$

כאשר הערך החדש נקבע כך שה-LSB שלו יהיה הביט הרצוי. ב-LSB Matching, כאשר יש צורך בשינוי, הערך מוגדל או מוקטן ב-1:

$$x \rightarrow x + 1 \quad \text{או} \quad x \rightarrow x - 1$$

הבחירה בין +1 לבין -1 יכולה להיעשות באופן פסאודו-אקראי. לכן, בעוד שב-LSB Replacement השינוי בערך הפיקסל נקבע ישירות לפי הביט שאנו מטמיעים, ב-LSB Matching השינוי נעשה בצורה פחות דטרמיניסטית.



תקיפת אלגוריתם LSB

ישנן שיטות שונות שבהן אנו תוקפים או מנתחים תמונה המשתמשת באלגוריתם LSB. כאן נתמקד בשיטת שיטות מרכזיות:

שיטת תקיפה	אלגוריתם
Chi-Square	LSB Replacement
Histogram of Differences	LSB Matching

Chi-Square

הרעיון מאחורי התקיפה פשוט מאוד. ב-LSB Replacement, כאשר אנחנו מחליפים את ה-LSB, אנחנו גורמים לזוגות של ערכי צבע עוקבים להיות דומים יותר מבחינה סטטיסטית. לדוגמה:

$$100 = 01100100 \quad 101 = 01100101$$

נבדלים רק ב-LSB. לכן ניתן לחלק את ערכי הצבע לזוגות באופן הבא:

$$(0,1), (2,3), (4,5), \dots, (254,255)$$

אלה נקראים Pairs of Values - PoVs.

למה זה קורה?

נניח שבמקור היו לנו:

$$100, 100, 100, 100, 101$$

כלומר:

$$H(100) = 4, \quad H(101) = 1$$

כאשר מטמיעים מידע באמצעות LSB Replacement, חלק מה-100 יכולים להפוך ל-101 וחלק מה-101 יכולים להישאר 101. כאשר כמות גדולה של מידע מוטמעת, ההבדל בין שני הערכים הולך ומצטמצם. באופן אידיאלי, אם הביט המוטמע אקראי, אזי שני הערכים בזוג נוטים להתקרב:

$$H(100) \approx H(101)$$

זה בדיוק מה ש-Chi-Square מנסה למדוד.



איך החישוב עובד?

נניח זוג כלשהו:

$$(2k, 2k + 1)$$

נסמן את מספר ההופעות שנמדד בפועל בתור:

$$O_0 = H(2k), \quad O_1 = H(2k + 1)$$

אם אין מידע נסתר, אין סיבה לצפות בהכרח שהם יהיו שווים. אבל אם קיימת הטמעת LSB משמעותית, אנחנו מצפים בקירוב:

$$E_0 = E_1 = \frac{O_0 + O_1}{2}$$

כאשר E הוא מספר ההופעות הצפוי לכל ערך בזוג. עכשיו משתמשים במבחן χ^2 :

$$\chi^2 = \sum_i \frac{(O_i - E_i)^2}{2}$$

ועבור זוג יחיד נקבל:

$$\chi^2 = \frac{(O_0 - E)^2}{2} + \frac{(O_1 - E)^2}{2}$$

דוגמה:

$$O_0 = 800, \quad O_1 = 200$$

לכן:

$$E = \frac{800 + 200}{2} = 500$$

$$\chi^2 = \frac{(800 - 500)^2}{2} + \frac{(200 - 500)^2}{2} = 180 + 180 = 360$$

קיבלנו ערך גדול, כלומר ההתפלגות שנצפתה רחוקה מאוד מההתפלגות הצפויה.

אבל חשוב שתזכרו! ערך χ^2 לבדו אינו "אחוז סטגנוגרפיה".

למה זה עובד דווקא על LSB Replacement?

ניקח זוג כלשהו:

(100,101)

ואילו בבינארי:

100 = 01100100

101 = 01100101

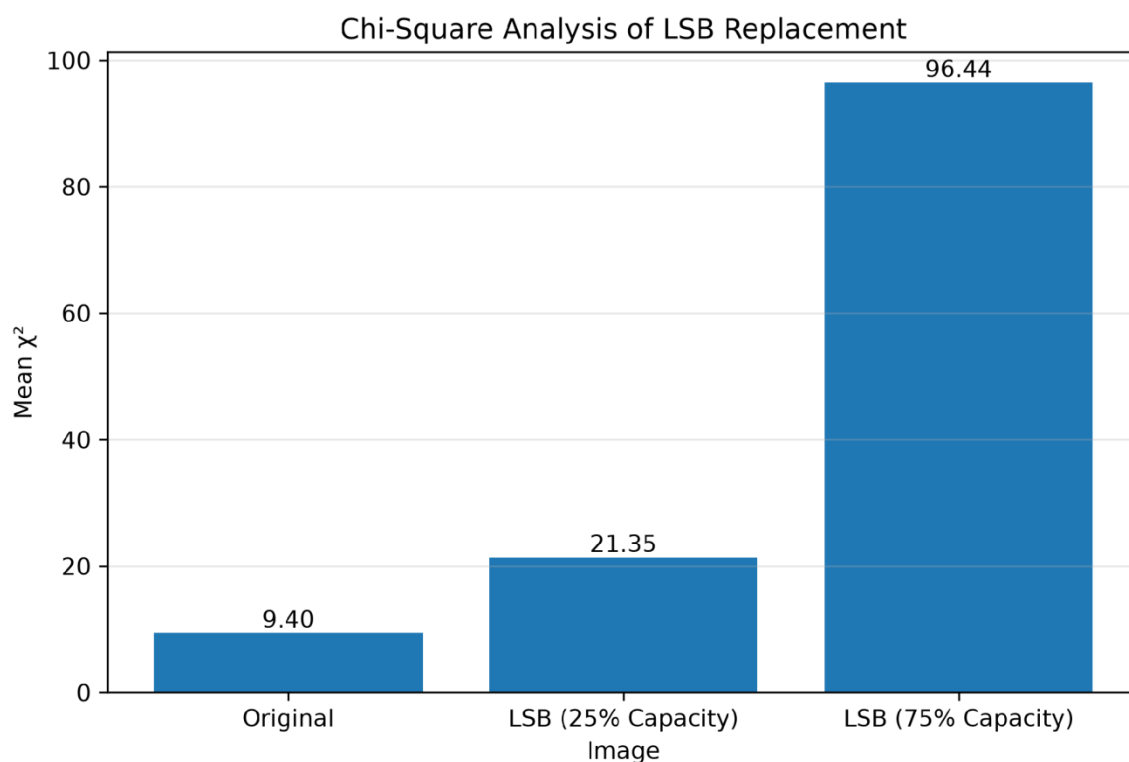
ב-LSB Replacement, אם אנו רוצים להכניס 1, הערך חייב להיות 101, ואם אנו רוצים להכניס 0, הערך חייב להיות 100. כלומר ההטמעה למעשה מעבירה ערכים בין הזוגות:

$100 \Leftrightarrow 101$

כאשר כמות גדולה של ביטים אקראיים מוחלפת, שני הערכים נוטים להתקרב בכמות ההופעות שלהם. לכן:

$$H(100) \approx H(101)$$

זה יוצר "טביעת אצבע" סטטיסטית שאפשר לחפש.





Histogram of Differences (HOD)

Chi-Square אינו הכלי המתאים ביותר ל-LSB Matching. הסיבה לכך היא ש-LSB Matching לא פשוט אומר:

$$x \rightarrow x'$$

אלא:

$$x \rightarrow x + 1 \quad \text{או} \quad x \rightarrow x - 1$$

באופן אקראי. כתוצאה מכך הוא נמנע מחלק מהאנומליות שהופכות את LSB Replacement לקל יחסית לזיהוי.

ואיך החישוב שלו עובד?

במקום להסתכל על:

$$H(x)$$

כלומר כמה פעמים כל ערך צבע מופיע, נסתכל על ההבדלים בין פיקסלים שכנים. נניח שיש לנו:

$$P_1, P_2, P_3, \dots$$

נחשב כי:

$$\Delta_i = P_{i+1} - P_i$$

לדוגמה:

$$[100, 102, 101, 105, 104]$$

נקבל:

$$[2, -1, 4, -1]$$

ומכאן נסיק כי הנוסחה להיסטוגרמה תהיה:

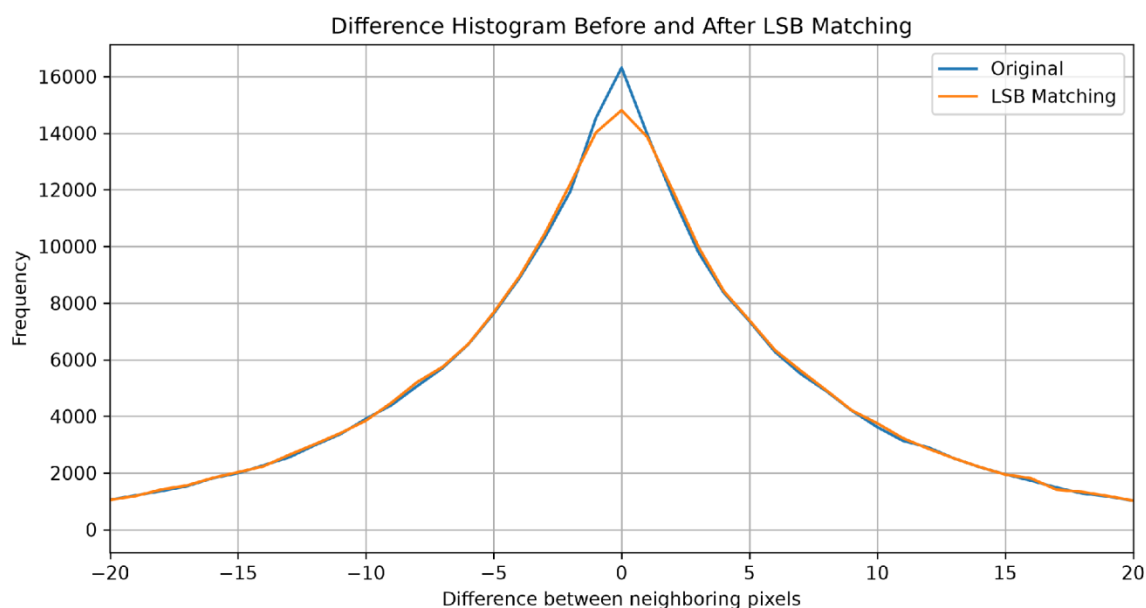
$$H_D(d) = |\{i: \Delta_i = d\}|$$

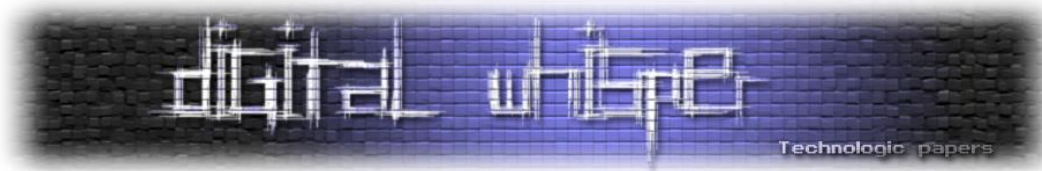
למה זה עובד?

בתמונות טבעיות, פיקסלים שכנים בדרך כלל דומים. לכן הרבה מההפרשים נמצאים בסביבות:

$$d = 0 \quad \text{או} \quad d = \pm 1, \pm 2, \pm 3$$

כאשר משתמשים ב-LSB Matching ומבצעים הרבה שינויים של ± 1 , אנחנו מוסיפים למעשה "רעש" קטן לערכי הפיקסלים. הרעש הזה משנה את ההפרשים בין פיקסלים שכנים, ולכן ההיסטוגרמה של ההפרשים נעשית חלקה יותר. זהו בדיוק אחד הרעיונות המרכזיים שהוצעו לזיהוי LSB Matching.





תמונות JPEG

תמונות מסוג JPEG הן סיפור שונה לגמרי מתמונות PNG. בעוד ש-PNG משתמש בדחיסה Lossless, כלומר לאחר הדחיסה ניתן לשחזר את המידע המקורי בדיוק, JPEG משתמש בדחיסה Lossy. בדחיסה זו חלק מהמידע בתמונה מושלך בכוונה כדי להקטין משמעותית את גודל הקובץ.

באופן מפושט, תהליך יצירת קובץ JPEG נראה כך:

פיקסלים $\leftarrow$ חלוקה לבלוקים של 8×8 $\leftarrow$ DCT $\leftarrow$ קוונטיזציה $\leftarrow$ קידוד $\leftarrow$ קובץ JPEG

לכן, בשיטת LSB, שבה משנים את הביט האחרון של ערכי הפיקסלים, ניתן להטמיע מידע ב-PNG ולשמור אותו גם לאחר שמירת התמונה. לעומת זאת, ב-JPEG הפיקסלים אינם נשמרים ישירות כפי שהם. התמונה עוברת תהליך דחיסה הכולל, בין היתר, המרה ל-DCT (Discrete Cosine Transform).

כתוצאה מכך, אם נשנה את ה-LSB של פיקסל בתמונה ולאחר מכן נשמור אותה כ-JPEG, הדחיסה עלולה לשנות את ערך הפיקסל ולמעשה להרוס את המידע שהוטמע.

לכן, ב-JPEG לא נהוג להטמיע מידע ישירות ב-LSB של הפיקסלים. במקום זאת, ניתן להטמיע את המידע במקדמי ה-DCT של התמונה, אשר נמצאים בשלב פנימי של תהליך הדחיסה. קיימות שיטות שונות לשינוי מקדמים אלה באופן שמאפשר להטמיע מידע תוך ניסיון לשמור על איכות התמונה ולצמצם את הסיכוי שההטמעה תהיה ניתנת לזיהוי.

חלוקת התמונה לבלוקים

כפי שצוין, JPEG מחלק את התמונה לבלוקים בגודל 8×8 פיקסלים. לדוגמה, בלוק יכול להיראות כך:

$$\begin{array}{c} x \\ \rightarrow \\ \downarrow y \end{array} \begin{pmatrix} -76 & -73 & -67 & -62 & -58 & -67 & -64 & -55 \\ -65 & -69 & -73 & -38 & -19 & -43 & -59 & -56 \\ -66 & -69 & -60 & -15 & 16 & -24 & -62 & -55 \\ -65 & -70 & -57 & -6 & 26 & -22 & -58 & -59 \\ -61 & -67 & -60 & -24 & -2 & -40 & -60 & -58 \\ -49 & -63 & -68 & -58 & -51 & -60 & -70 & -53 \\ -43 & -57 & -64 & -69 & -73 & -67 & -63 & -45 \\ -41 & -49 & -59 & -60 & -63 & -52 & -50 & -34 \end{pmatrix}$$

כל בלוק כזה מעובד בנפרד.

ולמה משתמשים ב-DCT?

ה-DCT ממיר את הייצוג של התמונה מייצוג במרחב הפיקסלים לייצוג במרחב התדרים. במקום לשאול "מה הערך של כל פיקסל?", אנחנו מקבלים מידע כגון "עד כמה הבלוק מכיל שינוי איטי או מהיר בעוצמת הבהירות?". כך ניתן לתאר את אותו בלוק באמצעות 64 מקדמים בלבד.

הטמעת מידע

המקדם הראשון, $C(0,0)$, נקרא DC ומתאר את הבהירות הממוצעת של הבלוק. שאר 63 המקדמים נקראים AC והם מתארים שינויים ותדרים שונים בתמונה. מקדמים בפניה השמאלית-עליונה מייצגים תדרים נמוכים, וככל שמתרחקים ממנה התדר עולה.

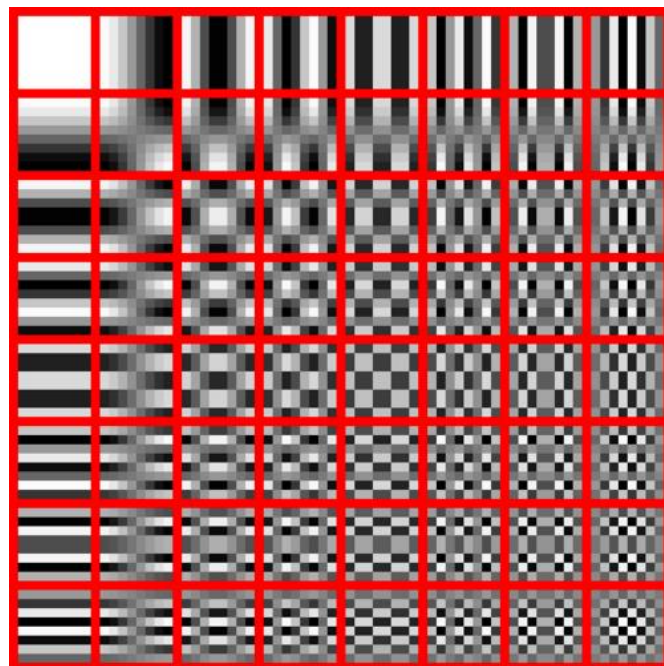
במקום לשנות את ה-LSB של הפיקסלים כמו ב-PNG, ב-JPEG ניתן לשנות מעט מקדמי AC כדי לייצג את הביטים של המידע הסודי. לדוגמה, בשיטה פשוטה אפשר להשתמש בזוגיות:

$$\text{זוגי} - \text{אי מקדם} \rightarrow 1, \quad \text{זוגי מקדם} \rightarrow 0$$

אם רוצים להטמיע 1 ויש לנו מקדם 4, נשנה אותו ל-5, וכך:

$$5 \bmod 2 = 1$$

השינוי קטן ולכן בדרך כלל אינו מורגש בתמונה:



[מקור: חלוקת תמונה לבלוקים של 8×8 על ידי ה-DCT]



תקיפת תמונות JPEG

ב-JPEG, התקיפה מתבצעת בעיקר על מקדמי ה-DCT, ולא ישירות על ערכי הפיקסלים. לאחר חילוך מקדמי ה-DCT מכל בלוק 8×8 , ניתן לבחון את ההתפלגות הסטטיסטית שלהם.

נסמן את אוסף מקדמי ה-AC ב- C . ניתן לחשב, לדוגמה, את מספר המקדמים השווים לערך מסוים:

$$H(k) = \#\{c \in C : c = k\}$$

כאשר $H(k)$ הוא מספר הפעמים שהמקדם k מופיע בתמונה. לאחר מכן ניתן להציג את $H(k)$ באמצעות היסטוגרמה ולחפש שינויים בהתפלגות.

בדיקת זוגיות

אם שיטת הסטגנוגרפיה משתמשת בזוגיות של המקדמים כדי להטמיע ביטים, ניתן לבדוק את מספר המקדמים הזוגיים והאי-זוגיים. נסמן:

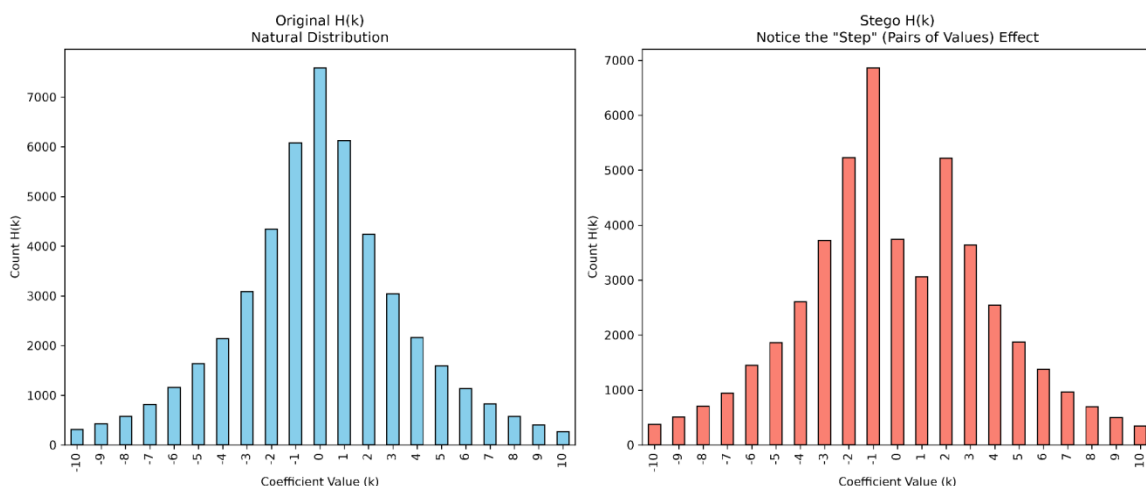
$$N_0 = \#\{c \in C : c \bmod 2 = 0\}$$

$$N_1 = \#\{c \in C : c \bmod 2 \neq 0\}$$

ניתן להגדיר את היחס:

$$R = \frac{N_1}{N_0 + N_1}$$

בתמונה שלא עברה הטמעה, ההתפלגות נובעת מהתוכן הטבעי של התמונה ומהתהליך הסטטיסטי של JPEG. כאשר מטמיעים מידע באמצעות שינוי זוגיות, היחס וההתפלגות עלולים להשתנות.





סיכום

במאמר זה ראינו כיצד תמונה דיגיטלית מיוצגת במחשב, וכיצד ניתן לנצל את המבנה הזה כדי להטמיע בתוכה מידע נסתר. התחלנו מ-LSB Replacement - השיטה הפשוטה והנפוצה ביותר, המשכנו ל-LSB Matching שנמנעת מחלק מהאנומליות הסטטיסטיות שלה, וראינו כיצד כל אחת מהן נתקפת: Chi-Square עבור הראשונה, והיסטוגרמת הפרשים עבור השנייה. לבסוף ראינו מדוע JPEG מחייב גישה אחרת לגמרי, שבה ההטמעה מתבצעת במקדמי ה-DCT ולא בערכי הפיקסלים.

המסקנה המרכזית היא שסטגנוגרפיה אינה עוסקת רק בהסתרה, אלא במאזן שבין כמות המידע המוטמע לבין העקבות הסטטיסטיות שהוא מותיר. ככל שמטמיעים יותר מידע, כך גדל הסיכוי שניתוח פורנזי יגלה זאת. בחלק הבא נמשיך ונעמיק בשיטות הטמעה מתקדמות יותר ובכלים המשמשים לזיהוי.

על המחבר

עידן קובליק, מפתח תוכנה, מתעניין ב-Low Level ועולם הסייבר. בנוסף אני אוהב לפתח קוד open source שאנשים אחרים יכולים להשתמש בו ולתרום לו.

Github: <https://github.com/IdanKoblik>

מקורות מידע

1. [A Review on Steganography Techniques - IJSCE, Vol. 3, Issue 5](#)
2. [Steganography - Wikipedia](#)
3. [JPEG - Wikipedia](#)
4. [PNG - Wikipedia](#)

5. [סטגנוגרפיה בשני שקל](#)

6. [Chi squared - Wikipedia](#)