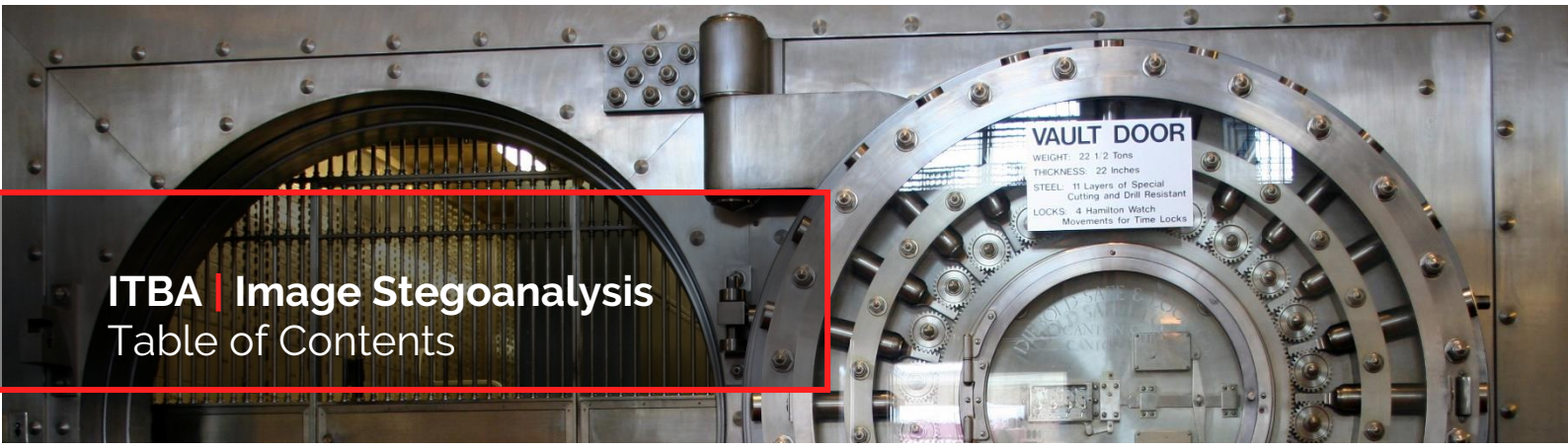
ITBA

Security & Cryptography

Image Stegoanalysis

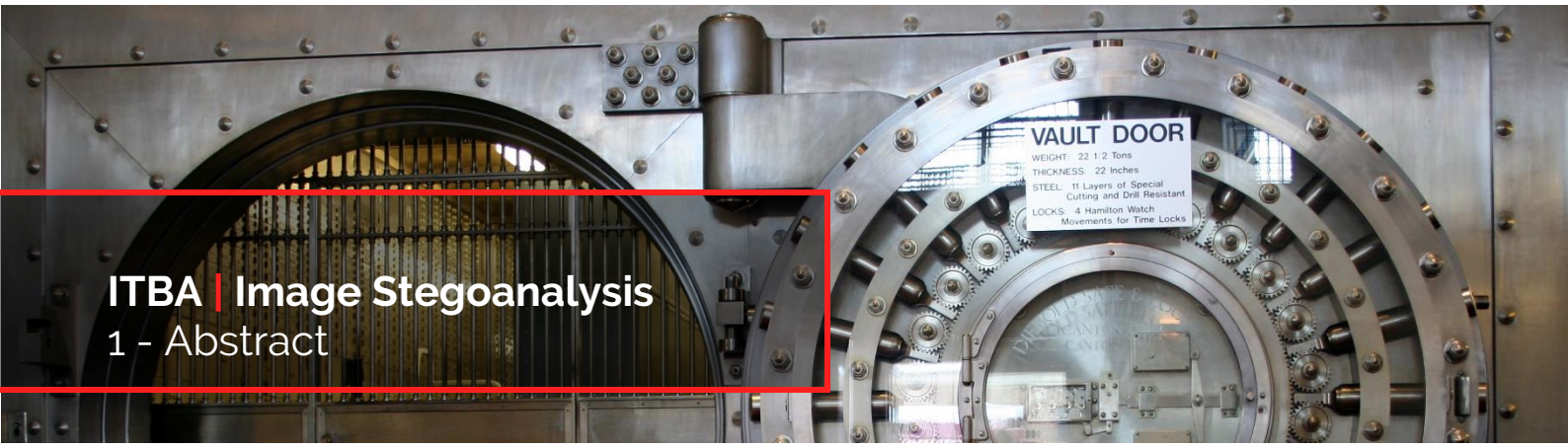
(53.396) Agustín Golmar

June 28, 2018



ITBA | Image Stegoanalysis
Table of Contents

Table of Contents



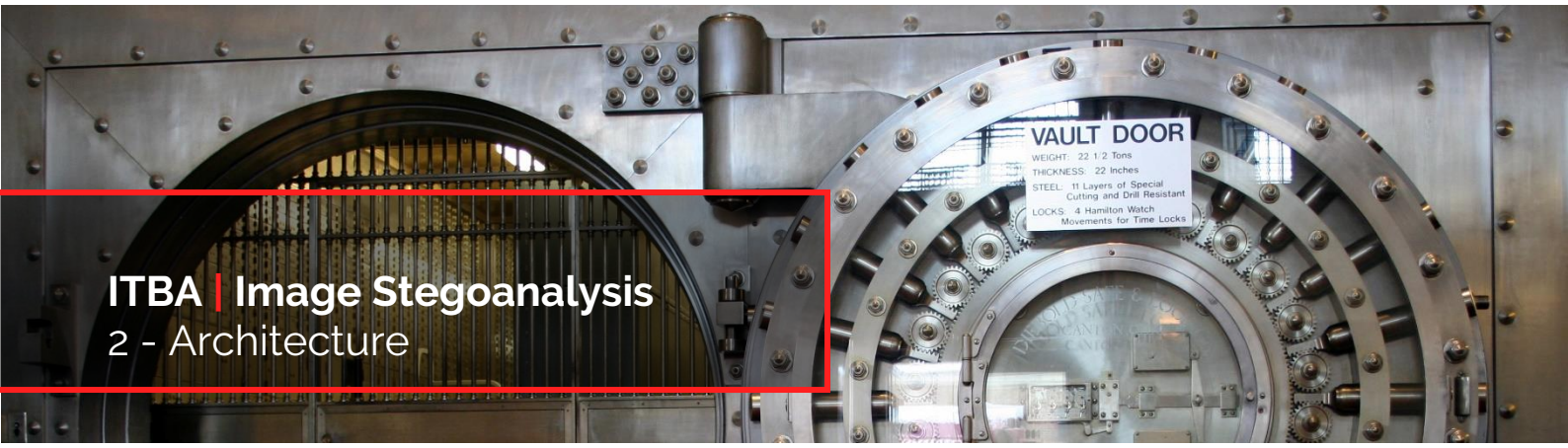
1. Abstract

La esteganografía (del griego `στεγανός`, cuyo significado es *"oculto"*, y `γράφειν`, que significa *"escritura"*), es la práctica que permite ocultar la existencia de un mensaje dentro de otro. Se diferencia de las técnicas criptográficas, ya que estas últimas permiten ofuscar el mensaje en sí mismo a modo de que no pueda extraerse información del *ciphertext* (mensaje cifrado). Sin embargo, es posible combinar ambas herramientas para generar un esquema de seguridad más completo.

En el siguiente informe, se detalla la construcción de una aplicación que provee diferentes métodos esteganográficos básicos, con los cuales es posible ocultar información de cualquier tipo dentro de imágenes, en particular, en aquellas que poseen formato *BMP v3* (*bitmap*). Adicionalmente, la aplicación provee diferentes cifrados configurables con los que se agrega un nivel adicional en el objeto escondido y en la dificultad que un posible atacante encontraría al intentar estegoanalizar dichas imágenes.

Además, la aplicación final será utilizada para analizar un set real de imágenes modificadas bajo los mismos algoritmos propuestos, a modo de verificar el funcionamiento de esta sobre un escenario real.

steganography | cryptography | side-channels | LSB



2. Architecture

En esta sección se describen las características funcionales y no funcionales de la aplicación, junto con los detalles de su implementación (tecnologías, consideraciones, requerimientos, dificultades, etc.).

2.1. Platform

La aplicación fue completamente desarrollada sobre *Java SE 8 Release* y puede ser construida utilizando *Maven 3.5.0* o superior. El código fuente es accesible desde un repositorio público en *GitHub*[1], y cuenta con integración contínua provista por *Travis CI*[2].

Adicionalmente, se utilizó JCE (*Java Cryptography Extension*), bajo la implementación oficial que viene por defecto con JDK 8 (*Java Development Kit*), en lugar de utilizar una dependencia adicional como *BouncyCastle*[3]. Esta implementación provee algunos cifrados adicionales, entre otras funciones, pero los esquemas utilizados en la aplicación son suficientemente estándar y por lo tanto se resuelven automáticamente con la plataforma por defecto (*i.e.*, SunJCE).

Para el parsing de los parámetros de entrada obtenidos por línea de comandos se utilizó *JCommander v1.72*[4], el cual facilita la transformación de dichas opciones en un POJO accesible. Es necesario destacar que dicha librería cuenta con serias limitaciones, a saber:

- No ofrece validaciones cruzadas entre parámetros.
- No ofrece validaciones a nivel de clase (invariantes de clase).
- No ofrece validaciones post-parsing de parámetros.
- Posee complicaciones en cuanto al setting de valores por defecto.

Para la construcción de los algoritmos destinados al procesamiento propiamente dicho, se hizo necesario utilizar la librería *Apache Commons-Lang v3.7*[5], en particular

[1] Disponible en la dirección https://github.com/agustin-golmar/Wolf-in-a-Sheep.
[2] Al que se puede acceder directamente desde el mismo repositorio.
[3] Cuyo sitio oficial es https://www.bouncycastle.org/.
[4] Su documentación oficial de uso se encuentra en http://jcommander.org/.
[5] Disponible en https://commons.apache.org/proper/commons-lang/.

haciendo uso de sus clases mutables, que ofrecen la posibilidad de retener estado variable accesible dentro de una función lambda (estas funciones requieren que los parámetros accedidos sean `final` o efectivamente `final`). De esta forma se evita tanto, una implementación propia de dichas clases, o la utilización de estructuras más complejas (como un array, una lista o una pila).

Finalmente, la licencia utilizada es WTFPL v2.0[6] (*Do What The Fuck You Want Public License*), que permite cualquier tipo de uso y/o modificación de cualquier copia de la aplicación.

La aplicación fue testeada tanto en sistemas Windows como Linux. Esta documentación fue diseñada bajo *Microsoft Office 365 ProPlus*, específicamente con *Microsoft Word*.

2.2. Execution

Se proveen dos modos fundamentales de operación:

- **Embedding**: dado un archivo cualquiera (el archivo que se desea ocultar), y una imagen en formato BMP (de ahora en más denominada *carrier*), se produce una nueva imagen (de ahora en más, el *stego object*), resultado de aplicar algún algoritmo basado en LSB (ver *Sección 2.4*), para embeber el objeto en el carrier. Esta imagen final se percibe prácticamente indistinguible de la original (*i.e.*, el carrier), pero alberga un archivo oculto en su interior.

- **Extraction**: dada una imagen previamente embebida mediante alguno de los mecanismos soportados, se procede a extraer el archivo oculto en su interior. Es importante notar que, si el archivo fue embebido con algún algoritmo no soportado, o incluso, con otra aplicación que no respete el mismo formato de codificación, la aplicación no podrá extraer satisfactoria o completamente el mensaje del stego object.

Como lo indica la *Sección 1*, además de una capa de esteganografiado se provee opcionalmente la posibilidad de encriptar el mensaje antes de embeberlo en el carrier, bajo alguna contraseña (es decir, utilizando un cifrado de clave privada). Los cifrados provistos son:

- **DES**: utiliza bloques de 64 bits y claves del mismo tamaño (aunque la seguridad efectiva provista es equivalente a una clave de 56 bits). No debe utilizarse debido a que este cifrado es inseguro.

- **AES**: se provee en sus tres versiones, esto es, claves de 128, 192 o 256 bits. En todos los casos el tamaño de bloque es de 128 bits.

[6] Originalmente en http://www.wtfpl.net/.

Como todo *block cipher*, es necesario disponer de algún modo de operación que permite cifrar mensajes de longitud superior al tamaño de bloque. Los modos disponibles por la aplicación (tanto para DES como para AES), son:

- **ECB (Electronic Code Book)**: encripta cada bloque por separado, y es el más inseguro de todos. Sin embargo, es uno de los más eficientes. Debido a que el mensaje debe ser múltiplo del tamaño de bloque, es necesario aplicar un proceso de *padding* para completar el mensaje, lo que implica que el mensaje final es más grande, y el espacio efectivo dentro del carrier es menor (aunque no más de 16 bytes, a lo sumo).

- **CBC (Cipher Block Chaining)**: el cifrado se aplica de forma secuencial sobre los bloques y, por ende, es el más ineficiente. Al igual que ECB, requiere padding.

- **OFB (Output Feedback Mode)**: permite transformar el cifrado de bloque en un *stream cipher*, generando un flujo del tamaño del mensaje aplicando repetidamente la primitiva de cifrado como fuente de pseudo-aleatoriedad, y finalizando la operación al igual que el cifrado de *Vernam*. La generación del flujo no es paralelizable, pero hace indispensable la aplicación de un algoritmo de padding para extender el mensaje.

- **CFB (Cipher Feedback Mode)**: al igual que OFB no requiere padding, pero, además, es paralelizable.

- **CTR (Counter Mode)**: es completamente paralelizable. Tanto durante el proceso de encriptación como durante el descifrado. No fue requerido por la cátedra[7], pero se incluyó por completitud y por simplicidad de implementación.

Finalmente, los algoritmos esteganográficos (descriptos en la *Sección 2.4*) disponibles, son:

- **LSB**: en tres versiones, de 1, 4 y 8 bits (ver *Sección 2.4*).

- **LSB Enhanced**: de 1 bit, según se describe en **[2]** (ver *Sección 2.4.1*).

Con todo esto, la ejecución por línea de comandos adquiere la forma:

```
$> java -jar stegobmp.jar <arguments>
```

Donde *<arguments>* representa una lista de los siguientes parámetros:

- **`-extract`**: especifica que el modo fundamental de operación es de extracción, por lo cual se obtendrá el mensaje oculto en un stego object provisto.

[7] *"Criptografía y Seguridad"*, 1er cuatrimestre de 2018. ITBA. C.A.B.A., Argentina.

- **`-embed`**: especifica el modo de incrustación, es decir, se tomará el archivo deseado y se ocultará dentro de una imagen carrier que pueda almacenar dicho mensaje.

- **`-in <filename>`**: en caso de que el modo fundamental sea de incrustación (utilizando la opción `-embed`), esta opción indica el archivo de entrada, es decir, el mensaje a esteganografiar dentro del carrier. El formato del archivo no tiene restricciones, así como también el tamaño de este (aunque claramente el carrier debe ofrecer suficiente espacio para retenerlo).

- **`-p <carrier>`**: la imagen carrier a utilizar, tanto para embeber un archivo en ella, o para extraer un mensaje previamente oculto.

- **`-out <filename>`**: el archivo final de salida. Representa el stego object durante una incrustación o el archivo escondido durante una extracción. En caso de extracción, no es necesario especificar la extensión, debido a que esta se extrae del stego object.

- **`-steg <algorithm>`**: el algoritmo de esteganografiado a utilizar. Se puede especificar LSB1, LSB4, LSB8 o LSBE. El nombre es *case-insensitive*.

- **`-a <cipher>`**: el nombre del cifrado a usar. Los valores disponibles son DES, AES128, AES192 o AES256. Nuevamente, el valor es *case-insensitive*.

- **`-m <mode>`**: el modo de operación del cifrado. Debe ser ECB, CBC, OFB, CFB o CTR. Es *case-insensitive*.

- **`-pass <password>`**: en caso de especificar un cifrado y su modo de operación, es necesario especificar una contraseña bajo la cual encriptar los datos esteganografiados. Esta contraseña será utilizada para derivar la correspondiente clave privada con la cual ocultar (o extraer) el mensaje en el carrier.

Para el modo de extracción, los parámetros **`-p`** y **`-out`** son obligatorios. Para embeber un archivo en el carrier es obligatorio, además, especificar el valor de **`-in`**. En todos los casos es requerido que se especifique el algoritmo de esteganografiado y el modo fundamental de operación.

En caso de utilizar cifrado, se deben especificar tanto **`-a`**, **`-m`** y **`-pass`** para completar el proceso de encriptación o desencriptación correspondiente. Por defecto, si solo se especifica la contraseña, se asume un cifrado AES de 128 bits, en modo CBC.

2.2.1. Considerations

Es importante tener en cuenta las siguientes consideraciones para comprender completamente el funcionamiento interno de la aplicación y reconocer efectivamente las situaciones en las cuales es útil aplicar o utilizar dicho sistema en una situación con determinados requerimientos:

- **Padding**: en caso de utilizar un modo que requiera un proceso de padding, como es el caso específicamente de los modos ECB y CBC, se utiliza el padding *PKCS #5*[8].

- **Key Generation**: para derivar una clave a partir de la contraseña especificada, se aplica un hash SHA-256 sobre la misma, y se utilizan los primeros bits necesarios por el cifrado indicado. El hash se aplica sin *salt* por lo cual la clave obtenida es determinística.

- **IV (Initialization Vector)**: los cifrados bajo los modos CBC, OFB, CFB y CTR requieren un vector de inicialización tanto durante el proceso de encriptación como durante el descifrado. Este vector se deriva de la contraseña junto con la clave de forma determinística al igual que lo hace el algoritmo `EVP_BytesTokey`[9] de *OpenSSL*. En particular, se genera una secuencia de 512 bits, resultado de aplicar las siguientes operaciones:

$$H(x) = SHA_{256}(x) \quad (1)$$

$$\hat{k} = H(p) \quad (2)$$

$$r = k \parallel H(\hat{k} \parallel p) \quad (3)$$

  Donde H representa la función de hash utilizada (*i.e.*, SHA-256 sin *salt*), p la contraseña especificada, $\hat{k}$ la clave derivada a partir de ella, r la secuencia pseudoaleatoria de 512 bits, y $\parallel$ el operador de concatenación de cadenas de bits. Se extraen de la secuencia r los primeros n bits para obtener la clave k efectiva requerida por el cifrado (donde n es el parámetro de seguridad y puede tomar los valores 64 para DES, o 128, 192 y 256 para AES), y de los $512 - n$ bits restantes se obtienen los primeros b bits necesarios para construir el IV, siendo b el tamaño de bloque (donde b es 64 para DES o 128 para AES). La secuencia restante de $512 - n - b$ bits se descarta.

  Es importante notar, que la derivación de IV es determinística y por lo tanto el cifrado de un mensaje bajo la misma clave también lo es, por lo cual este esquema de cifrado es susceptible a CPA (*Chosen-Plaintext Attack*).

  Ya que la secuencia IV es determinística y derivable desde la contraseña, no es necesario embeberla junto con el mensaje esteganografiado.

[8] Definido en su forma más actualizada en https://tools.ietf.org/html/rfc8018.
[9] La documentación completa de dicho algoritmo se encuentra en https://www.openssl.org/docs/man1.1.0/crypto/EVP_BytesToKey.html.

- **Feedback**: para los modos OFB, CFB o CTR, se planteó la necesidad de utilizar un feedback que permita compatibilidad tanto en aplicaciones Java bajo JCE o en C usando OpenSSL. Por ello se decidió utilizar un feedback de 8 bits para CFB y de 128 bits para OFB y CTR.

- **Stronger Keys**: en caso de utilizar el cifrado AES de 192 o 256 bits, es necesario que se encuentre instalada la póliza[10] de extensión de seguridad para JCE. Esta póliza permite utilizar claves de longitud superior a 128 bits, para cualquier cifrado que lo soporte.

- **Bitmap Format**: las imágenes utilizadas como carrier deben ser del tipo *Bitmap v3*, no poseer ningún nivel de compresión y utilizar una profundidad de color de 24 bits. En caso de utilizar una imagen en otro formato, la aplicación arrojará un mensaje de error adecuado. Ver *Sección 2.3* para más información.

- **Row Padding**: el formato de un archivo Bitmap (ver *Sección 2.3*), dispone la información de color de cada píxel en bloques contiguos de w puntos, siendo que la imagen posee una dimensión de $w \times h$ píxeles. Para una profundidad de color de d bits, este bloque posee $d \cdot w$ bits. Si este número no es múltiplo de 32 bits, se aplica un proceso de padding. El algoritmo de esteganografiado utiliza de todas formas esta zona de padding para ocultar datos, en caso de que lo necesite (utilizar la zona de padding permite almacenar mensajes más grandes).

2.3. Bitmap Format

Las imágenes utilizadas como carrier, y los stego object obtenidos al aplicar efectivamente un proceso de esteganografiado, deben respetar el formato Bitmap v3. En general, la estructura[11] de un archivo de este tipo se subdivide en dos bloques principales:

- **Header**: posee un largo de 54 bytes y contiene la metadata del archivo, es decir, los parámetros bajo los cuáles se construye la imagen y se debe visualizar. Se subdivide en 3 sectores denominados BITMAP_FILE_HEADER, BITMAP_INFO_HEADER y RGB_QUAD. En particular, se destacan los siguientes campos:

  - **Signature (*bfType*)**: dos bytes que representan el formato del archivo y siempre valen 0x424D, lo que representa la cadena *BM* en ASCII.

  - **Width (*biWidth*)**: cuatro bytes para el ancho de la imagen, en píxeles.

[10] Para *Java SE 8 Release*, puede obtenerse en http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html.
[11] La estructura completa se encuentra detallada en https://msdn.microsoft.com/en-us/library/dd183392(v=vs.85).aspx.

- **Height (*biHeight*)**: cuatro bytes para el alto de la imagen, en píxeles.

- **Color Depth (*biBitCount*)**: dos bytes para la profundidad de color. Como se esperan 8 bits para cada uno de los canales RGB, este campo debe indicar una profundidad de 24 bits.

- **Compression (*biCompression*)**: cuatro bytes para especificar el modo de compresión. Como las imágenes soportadas no deben estar comprimidas, este campo debe valer cero (*i.e.*, *BI_RGB*).

- **Body**: el cuerpo es formalmente denominado COLOR_INDEX[12] y representa un vector de píxeles, donde cada píxel se compone de 3 componentes (RGB). El tamaño de la secuencia RGB es equivalente a la profundidad de color. Cómo se indicó en la *Sección 2.2.1*, si el tamaño en bits de una fila de píxeles de la imagen no es múltiplo de 4 bytes, se aplica un padding final rellenando con ceros. El tamaño efectivo l en bytes de esta fila viene dado por la expresión:

$$l = 4\left\lfloor \frac{31 + d \cdot w}{32} \right\rfloor \qquad (4)$$

  Donde d es la profundidad de color en bits y w el ancho de la imagen en píxeles. Si la imagen posee una altura h en píxeles, la zona efectiva de esteganografiado es de $l \cdot h$ bytes.

2.4. LSB Algorithm

La aplicación ofrece 4 modos de esteganografiado, todos ellos basados en un algoritmo más general denominado LSB (*Least Significant Bits*). El algoritmo opera de la siguiente forma:

1. Definir un filtro $P : \{0,1\}^8 \longrightarrow \{0,1\}$ de selección de bytes (un predicado).
2. Consumir un byte b_m del mensaje a ocultar.
3. Separar el byte en $p = 8/n$ paquetes, donde n es la cantidad de bits.
4. Consumir p bytes $(c_1, \ldots, c_p)$ del *payload* del carrier que verifiquen $P(c_k) = 1$.
5. Insertar cada paquete en los n bits menos significativos de cada byte c_k.
6. Ejecutar el paso 2 hasta que se consuma el mensaje a ocultar.

Es decir, se fragmenta el mensaje a ocultar en paquetes de n bits, y cada uno de estos paquetes se dispone en los bits menos significativos de cada byte del payload del carrier, donde por *payload* se entiende a la zona efectiva del carrier en la cual el algoritmo de esteganografiado tiene permitido embeber el mensaje.

[12] Ver https://msdn.microsoft.com/en-us/library/dd183391(v=vs.85).aspx.

En esta aplicación, el payload del carrier representa el vector COLOR_INDEX de la imagen bitmap (ver *Sección 2.3*), donde cada byte puede ser, o bien una componente RGB, o bien un byte de padding (en caso de que la imagen lo requiera).

La elección de utilizar los bits menos significativos es vital, *i.e.*, permite reducir la variación percibida del color de cada píxel, siempre que el factor n sea pequeño.

La aplicación provee implementaciones del algoritmo LSB para un parámetro n de 1, 4 y 8 bits. En particular, cuando $n = 8$, los bytes del mensaje a ocultar reemplazan por completo los bytes del carrier, y la imagen es distorsionada significativamente, por lo cual, este modo de operación se incluye solo a modo de prueba y no debería ser utilizado en una situación real.

Para los 3 formatos ofrecidos (LSB1, LSB4 y LSB8), el predicado P se define de la siguiente forma:

$$P(c) = 1 \; \forall c \tag{5}$$

Es decir, se seleccionan todos los bytes disponibles en el payload, sin excepción.

Al aumentar n, la distorsión de la imagen es mayor, pero también el tamaño disponible para ocultar el mensaje. Para obtener la menor distorsión, debe utilizarse LSB con $n = 1$, o mejor aún, una modificación de este modo, denominada *LSB Enhanced*.

2.4.1. LSB Enhanced

Se implementó una versión modificada del algoritmo *LSB Enhanced* definida en **[2]**, la cual implementa el algoritmo LSB1, es decir, LSB con $n = 1$ pero, además, define un predicado diferente:

$$P(c) = \begin{cases} 1, & c = 254 \vee c = 255 \\ 0, & sino \end{cases} \tag{6}$$

Como lo indica la expresión **(6)**, sólo se embebe un byte del mensaje oculto sobre aquellos bytes del payload cuyo valor es 0xFE o 0xFF. Debido a ello, el mensaje puede incrustarse en la imagen con mayor dispersión y no de forma contigua, aunque un claro efecto colateral es la significante (y posible) reducción del espacio disponible para almacenar el mensaje.

Es importante notar que este esquema provee un mecanismo que puede ser decodificado y por lo tanto es funcional. Si el byte c del payload es 0xFE, la modificación del LSB sobre este byte solo puede generar el valor 0xFE o 0xFF. De igual forma, si $c = 255$, el byte esteganografiado solo puede tomar los valores 0xFE o 0xFF. En definitiva, el proceso de decodificación (o extracción) del mensaje oculto consiste en extraer los bits menos significativos de aquellos bytes del stego object que verifiquen el predicado P definido en **(6)**.

2.5. Implementation

El objetivo de esta sección es describir los conceptos mediante los cuales se llevó a cabo la implementación de los algoritmos de esteganografiado y la capa de cifrado.

2.5.1. Overall Structure

El código[13] es accesible desde un repositorio público, bajo el dominio `ar.nadezhda.crypt`, y se subdivide en los siguientes paquetes:

- **`.cipher`**: contiene los cifrados disponibles en todas sus presentaciones disponibles (por longitud en bits de la clave). Gracias a las clases de soporte y al patrón *abstract factory* (ver paquete **`.factory`**), es posible extender la cantidad de cifrados, modos y métodos esteganográficos fácilmente.
    - **`.mode`**: los modos de operación disponibles. Estos modos pueden trabajar sobre cualquiera de los cifrados definidos (DES y AES).
- **`.config`**: dedicado a describir la configuración y validación de entrada haciendo uso de la librería *JCommander* (ver *Sección 2.1*).
- **`.core`**: contiene los algoritmos propios de la aplicación destinados a transformar el flujo de datos. Cada uno de los componentes se detalla en las siguientes secciones.
    - **`.exception`**: las excepciones posiblemente generadas durante la ejecución de los algoritmos de transformación o de configuración. Todas son del tipo *checked*.
    - **`.flow`**: contiene los flujos disponibles. Para la definición de flujo ver la *Sección 2.5.2*.
    - **`.pipe`**: contiene las transformaciones (*a.k.a.* pipes). Para la definición de *pipe* ver la *Sección 2.5.3*.
- **`.factory`**: implementa el patrón *abstract factory* sobre tres fábricas, una para cifrados, una para modos y otra para métodos esteganográficos. La localización de clases y constructores se resuelve por *reflection*, lo que permite desacoplar el agregado de nuevos algoritmos directamente y sin configuración adicional.

[13] https://github.com/agustin-golmar/Wolf-in-a-Sheep

- **`.interfaces`**: todas las interfaces que utiliza la aplicación. Los algoritmos operan sobre interfaces en lugar de clases para mayor flexibilidad.

- **`.steganographer`**: contiene la definición genérica de un algoritmo LSB y sus implementaciones particulares. Este paquete también ofrece localización automática al igual que los cifrados y los modos de operación.

- **`.support`**: provee clases de soporte y funcionalidades complementarias, como el localizador de clases, el generador de claves y IVs, el *timer*, o los mensajes disponibles.

2.5.2. Flow

La aplicación se basa en la idea de transformar un flujo de bytes a través de diferentes etapas intercambiables y reutilizables. Esto permite procesar o describir transformaciones de forma declarativa. Las operaciones intermedias indican como modificar el flujo que las atraviesa, pero son *lazy*, por lo cual no aplican ninguna transformación si el resultado final de dicho proceso nunca es consumido.

Adicionalmente, estas transformaciones pueden agregar o desagregar el flujo modificando el tipo de operaciones que son permitidas sobre el mismo. Cada una de estas operaciones se definen dentro de un *pipe* (ver *Sección 2.5.3*), y las agregaciones a través de interfaces (ver *Sección 2.5.2.3*).

La interfaz `Flow` define solo dos métodos: `isExhausted`, que determina cuando un flujo de bytes puede o no ser consumido, y `consume`, que permite manipular el siguiente byte del flujo a través de un objeto especial, denominado `Drainer`.

En la *Sección 2.6* se expondrán algunas modificaciones que sería favorable implementar en un futuro sobre estos conceptos.

2.5.2.1. Drainer

Un objeto `Drainer` es aquel que expone un único método con el cual se puede consumir (o drenar) un flujo cualquiera. Este objeto se procesará por cada byte, y cada byte procesado será drenado por única vez.

Para que un drenador entienda en qué parte del flujo se encuentra, se transmite adicionalmente la posición que dicho byte ocupa dentro del flujo que se está consumiendo, aunque que con una salvedad: la posición es relativa a la transformación y no a la fuente del flujo original. Esto otorga mayor flexibilidad, ya que un flujo puede, entonces, ser aumentado, ser reducido o permanecer invariante.

Aprovechando el concepto `Drainer`, se hace posible más adelante concatenar transformaciones mediante *pipes*.

2.5.2.2. File Flow

La aplicación debe consumir un flujo de bytes asociado al archivo de entrada, es decir, al mensaje que se desea ocultar (durante el esteganografiado), y otro flujo proveniente del carrier. Ambos flujos provienen de archivos, por lo cual la primera definición de flujo se corresponde con una instancia que permite consumir byte a byte un archivo en disco. Esta instancia queda expresada por la clase `FileFlow`.

Debido a que las operaciones son *lazy*, el archivo se consume byte a byte solo cuando sea necesario, aunque para reducir el número de operaciones se leen grandes sectores (en la implementación se utilizan buffers de 8 Kb.), que se transmiten poco a poco desde memoria.

Inmediatamente, manipular un archivo como flujo permite que la aplicación pueda esteganografiar archivos de cualquier tamaño, ya que no es necesario mantener más que 8 Kb a lo sumo, en memoria.

2.5.2.3. Flow Interfaces

Se utilizan numerosas agregaciones o desagregaciones[14] durante toda la aplicación. En particular, cada una aplica en la etapa de procesamiento que le corresponde, pudiendo o no reaparecer en ciertas partes de la transformación final. Estas incluyen:

- `BoundedFlow`: consumir un flujo de bytes no siempre implica conocer si el flujo es finito o infinito. Un flujo que implementa dicha interfaz posee límite, y ese límite es conocido

- `NamedFlow`: si el flujo posee nombre (*e.g.*, si proviene de un archivo), puede implementar esta interfaz.

- `RegisteredFlow`: este flujo es en realidad una mezcla de los dos anteriores, por lo tanto, posee nombre y límite.

- `BitmapFlow`: si el flujo presenta la estructura de un formato *Bitmap*, puede implementar esta interfaz y ofrecer un abanico de métodos adicionales para obtener propiedades de dicha estructura (ver *Sección 2.3*).

- `FlushableFlow`: si el flujo implementa esta interfaz, debe exponer un método que permita consumir por completo el flujo en una sola llamada, opcionalmente proporcionando un `Drainer` adicional. Es

[14] En alguna etapa, el flujo que ingresa en un *pipe* puede resultar en un flujo de otro tipo. Una **desagregación** implica justamente una transformación hacia un tipo de flujo que expone menos funcionalidad, o que es más simple. Por ejemplo, un flujo que contiene metadatos pierde los métodos asociados a su metadata luego de atravesar una operación de cifrado, ya que ellos dejan de ser accesibles.

equivalente o análogo a las operaciones terminales de las clases `Stream`[15] que provee *Java*.

2.5.2.4. Empty Flow

La clase `EmptyFlow` define un flujo muy particular: el flujo vacío. Debido a que el flujo es vacío no puede consumirse, y por lo tanto cualquier intento de hacerlo produce una excepción del tipo `ExhaustedFlowException`.

Si bien este flujo puede carecer de sentido a primera vista, es de gran utilidad cuando se desea bloquear el flujo de salida de alguna transformación, por ejemplo, al detectar errores de formato, sin intervenir o detener las operaciones normales de transformación. Es decir, ante la presencia de un flujo no-consumible, las sucesivas transformaciones no aplicarán ningún proceso, y la ejecución finalizará inmediatamente.

2.5.3. Pipeline

El concepto de flujo deja de ser útil si no se proveen mecanismos para transformar dichos objetos o para combinar operaciones sobre ellos. Para resolver esta carencia se definió el concepto de *pipeline* como un proceso de transformación de cierto flujo de bytes.

Un *pipeline* es en realidad la unión de uno o más conectores denominados *pipes*, y cada pipe debe ser de tipo *pipelinable*. La interfaz asociada queda expresada en `Pipelinable`, y expone un único método `inject`, que como su nombre lo indica, permite inyectar un flujo en la tubería y aplicar una transformación sobre el mismo. En definitiva, un *pipe* es una función $P : Flow \longrightarrow Flow$ o, dicho de otra forma, un operador unario sobre flujos.

Si los tipos del flujo de salida son compatibles con los del flujo de entrada de otro pipe, estos se pueden conectar, generando un pipeline más complejo, lo que es equivalente a aplicar una composición de funciones.

En las siguientes secciones se definen los tipos de pipe de cada modo fundamental, *i.e.*, para embeber o extraer un mensaje oculto.

2.5.3.1. Embedding Pipeline

El proceso de esteganografiado implica generar dos flujos a partir de dos archivos diferentes (el mensaje a ocultar y el carrier en formato Bitmap), para luego combinar ambos en el stego object final. Adicionalmente, durante el proceso completo se puede optar por incluir una capa de cifrado. Los componentes (pipes) involucrados en dicho proceso son:

[15] https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

- `MetadataPipe`: el mensaje oculto se agrega con 4 bytes que indican su tamaño original más la secuencia ASCII de la extensión del archivo fuente. En la implementación se utiliza UTF-8 que es compatible con ASCII, y *big endian* para codificar el tamaño original. Por lo tanto, este pipe incrementa el tamaño del flujo al agregar información adicional.

- `EncryptedPipe`: su objetivo es cifrar el flujo entrante, que en este caso es generado por `MetadataPipe`. El cifrado opera de forma similar al flujo `FileFlow`: consume un gran bloque de bytes, antes de cifrar y relanzar el flujo cifrado hacia el siguiente pipe. Esto permite encriptar por bloques y no byte a byte. Al igual que `MetadataPipe`, el flujo de salida se agrega con el tamaño final del flujo generado.

- `IdentityPipe`: en caso de que el cifrado no sea requerido, se reemplaza el conector `EncryptedPipe` por el conector identidad, cuyo único objetivo es devolver el mismo flujo que consume (*i.e.*, no aplica ninguna transformación). Es equivalente a la función identidad, aplicada en el dominio de los flujos.

- `Steganographer`: el mensaje oculto se transforma en flujo y se une al carrier. Esta operación difiere del resto de transformaciones debido a que debe operar sobre dos flujos, en lugar de uno. En la *Sección 2.5.3.3* se detalla específicamente el concepto utilizado que permite integrar fácilmente este tipo de procesos en el *pipeline* final.

- `BitmapPipe`: se aplica al flujo de bytes el formato de un archivo Bitmap y se extraen las propiedades principales, citadas en la *Sección 2.3*. Sin embargo, las propiedades no se validan en este conector.

- `BitmapValidationPipe`: la validación del flujo se realiza en este pipe. Este es un ejemplo de transformación que no modifica el flujo de entrada de ninguna forma. Además, demuestra que los pipes pueden utilizarse para otras funciones (en este caso, a modo de filtros, similar a un patrón *intercepting filter*).

- `OutputPipe`: ofrece una operación terminal con la cual volcar el contenido final del flujo en un archivo, generando así el stego object final.

2.5.3.2. Extraction Pipeline

El proceso de extracción reutiliza algunos de los conectores del proceso de esteganografiado, en particular:

- `BitmapPipe`
- `BitmapValidationPipe`
- `Steganographer`
- `OutputPipe`

Sin embargo, esto no basta, ya que las operaciones previas al esteganografiado deben invertirse o desenvolverse (*i.e.*, el cifrado y el agregado de metadatos). De ello se encargan:

- `DecryptedPipe`: realiza el proceso inverso del conector `EncryptedPipe`. Opera de la misma forma y en caso de que la aplicación se ejecute sin proveer un cifrado, el conector se reemplaza por una instancia de `IdentityPipe`.

- `WolfPipe`: aplica la operación inversa del conector `MetadataPipe`, es decir, extrae el tamaño del payload, el payload en sí mismo, y la extensión del archivo con la cual el último conector de la cadena (`OutputPipe`), puede reconstruir el mensaje oculto original.

2.5.3.3. Steganographer

En las secciones anteriores se mencionó el tipo `Steganographer`, pero sin entrar en detalle de su estructura interna.

En particular, un algoritmo de esteganografiado opera en dos modos, al ocultar un mensaje o al extraerlo. Siendo un conector un operador unario sobre flujos de la forma $P : Flow \longrightarrow Flow$, entonces es ideal que un `Steganographer` implemente la interfaz `Pipelinable`, para de esta forma resolver el problema de extracción.

No obstante, el problema de embeber un mensaje dentro del carrier involucra dos flujos de entrada y uno de salida. Es decir, el `Steganographer` debería proveer un operador binario sobre flujos, de la forma:

$$M' : Flow \times Flow \longrightarrow Flow \qquad \textbf{(7)}$$

Por ello, se define la interfaz `Mergeable`, la cual también es implementada por un `Steganographer`. Debido a que un operador binario rompe la estructura de un pipeline, se decidió *currificar* la definición **(7)**, obteniendo:

$$M : Flow \longrightarrow (Flow \longrightarrow Flow) \qquad \textbf{(8)}$$

De esta forma, un tipo `Mergeable` es en realidad una función que recibe un flujo (que en la implementación específica representa el *carrier*), y devuelve un conector preparado para embeber cualquier flujo adicional provisto en el primero. Semánticamente, la expresión **(8)** es en realidad:

$$M : Carrier \longrightarrow (Payload \longrightarrow StegoObject) \qquad \textbf{(9)}$$

Y la expresión para P es:

$$P : StegoObject \longrightarrow Payload \qquad \textbf{(10)}$$

Finalmente, un `Steganographer` queda representado por la tupla $\langle M, P \rangle$, donde M es la función que permite embeber, y P la que permite extraer el mensaje oculto. Nótese que en otros documentos (como en **[3]**), algunos

autores prefieren incluir en la definición formal la clave utilizada para encriptar/desencriptar el mensaje oculto. Sin embargo, como este proceso es opcional y es en realidad una función que transforma solo el payload, no se incluye en **(9)** ni en **(10)**.

2.6. Future Improvements

La aplicación es factible de recibir la siguiente lista de mejoras en el futuro, aunque la lista obviamente no es exhaustiva:

- **Security over CPA**: durante la etapa de cifrado, el vector de inicialización (*IV*) es derivado directamente de la contraseña. En **[1]** se expresa que un cifrado seguro frente a CPA debe generar su vector de inicialización desde una fuente pseudoaleatoria y que, además, este vector debe utilizarse solo una vez. La aplicación falla en este concepto. Como mejora, el vector debería generarse cada vez que se aplique un proceso de esteganografiado, y por lo tanto debería agregarse al payload que se desea ocultar (como metadato).

- **MAC**: un atacante puede modificar el stego object aplicando alguna transformación (ver **[3]**) sobre la imagen (compresión con pérdida, escalado, rotación), y de esta forma modificar el mensaje oculto (o invalidarlo). Si bien, el desarrollo de técnicas esteganográficas que soporten dichos ataques está asociado al *watermarking* y fingerprinting, sería útil incluir una capa adicional de MAC (*Message Authentication Code*), que permite verificar la integridad del mensaje, o bien, un esquema criptográfico que incluya tanto cifrado como autenticación (como en el modo GCM, que provee *authenticated encryption*).

- **Public-Key Cryptography**: sería conveniente ofrecer un esquema de cifrado de clave pública con el cual es posible que dos entidades interactúen bajo un canal encubierto sin intercambiar la clave privada. Este agregado es funcional, es decir, el esquema de clave privada no debería ser reemplazado porque no siempre es posible que dos entidades intercambien la clave pública.

- **Configurable LSBE**: el algoritmo propuesto por Sridevi *et. al.* en **[2]** utiliza sólo los bytes 0xFE y 0xFF para ocultar el mensaje y la implementación se corresponde con este hecho. Una desventaja directa de este esquema es que el espacio disponible depende directamente de la cantidad de bytes con dicho valor en el carrier, y es probable que en un escenario real no se pueda elegir el carrier arbitrariamente. Sin embargo, es posible ofrecer una versión LSBE configurable que permita decidir sobre qué valor[16] ocultar. *E.g.*, seleccionar el valor 0x10, implicaría aplicar LSBE sobre los bytes 0x10 y 0x11.

[16] Los valores seleccionados deben ser un número par cualquiera, y el siguiente, ya que el algoritmo opera modificando el LSB de dicho valor. Es decir, existe arbitrariedad en el valor elegido, siempre que se respete esta precondición.

- **Bitmap Padding**: dado un carrier, la aplicación esteganografía un mensaje utilizando toda la zona efectiva de la tabla COLOR_INDEX en el bitmap (ver *Sección 2.2.1* y *2.3*). debido a que se puede ocultar parte del mensaje dentro del sector de *row padding* (en caso de que exista), sería fácil para un atacante detectar modificaciones en este sector, ya que en general se espera que esta zona se complete con ceros. La futura versión de esta aplicación debería evitar utilizar esta zona, y/o permitir que el usuario decida a través de algún parámetro de configuración adicional.

- **Image Locator**: en **[7]** se propone utilizar un filtro para analizar la imagen y así poder detectar la zona de esteganografiado a modo de reducir significativamente la cantidad de cambios necesarios que el algoritmo LSB debe realizar para ocultar un mensaje. Al reducir la cantidad de cambios en la imagen, se podrían reducir los ataques por análisis estadístico que se observan en **[5]**. Básicamente, se debería detectar el byte de origen que reduce la cantidad de cambios, y agregar esta marca de inicio dentro de la metadata del esteganografiado (sin la marca, no se podría localizar el origen del mensaje).

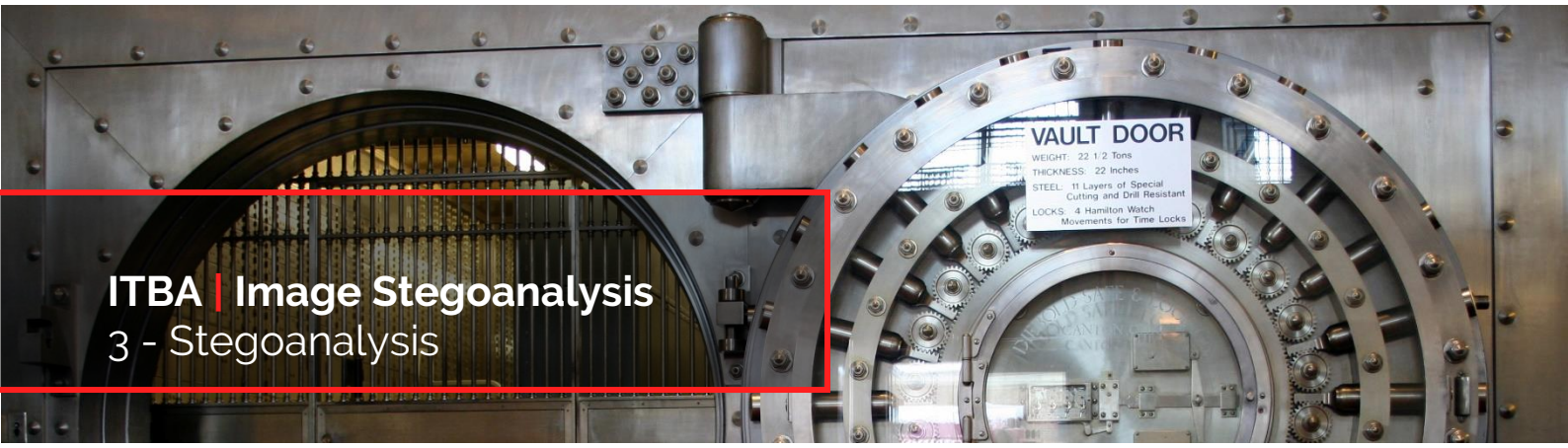
- **Lossy Formats**: actualmente, la aplicación utiliza el formato Bitmap v3 sin compresión como carrier. Sin embargo, transportar mensajes en objetos bitmap implica utilizar archivos que en general son de gran tamaño para altas resoluciones, pero considerablemente más pesados que las imágenes en formatos comprimidos con pérdida, como es el caso de PNG o JPG (ver **[3]** y **[5]**). Esteganografiar imágenes en este tipo de formatos con pérdida no es trivial, pero implicaría una gran ventaja ya que estos formatos son ampliamente utilizados a través de la web y reduciría las sospechas observadas al detectar una *party* enviando múltiples imágenes en BMP.

- **Key Generation**: la clave se genera con un esquema propuesto por OpenSSL (ver *Sección 2.2.1*), sin embargo, la cátedra sugirió utilizar PBKDF2 (del estándar PKCS #5 v2.0), que utiliza un algoritmo diferente durante la derivación. Además, la cantidad de rondas aplicadas actualmente es sólo 1, y este parámetro debería incrementarse.

- **ByteBuffer-driven**: los algoritmos propuestos operan sobre flujos, consumiendo estos, byte a byte. Sería ideal relajar esta condición y ofrecer una interfaz para consumir un buffer completo (*ByteBuffer* en Java), lo que permite reducir la cantidad de llamados a funciones para procesar un pipeline completo.

- **More Pipes**: algunas operaciones fueron empaquetadas dentro de un pipe específico, pero claramente pueden extraerse y generalizarse. Un ejemplo claro de ello es el caso de la agregación o desagregación de información (en particular en `MetadataPipe` o `EncryptedPipe`, donde se agrega el tamaño y la extensión). Estos conceptos podrían abstraerse en conectores especiales y separados, lo que incrementaría la reusabilidad y la separación de intereses (*separation of concerns*). Podría implementarse algún conector de sincronización de flujos, por ejemplo, en el caso del esteganografiado, ya que este algoritmo opera bit a bit (en LSB1), pero el flujo se mueve byte a byte. Esto incrementa la complejidad y obliga a mantener estado.

- **Stateless Pipes**: la última mejora propuesta implica generar conectores que no retengan estado. En particular, si no hay estado, se podría dispensar del uso de clases `Mutable*` (ofrecidas por *Apache Commons-Lang*).



3. Stegoanalysis

En esta sección, la aplicación fue utilizada en un escenario real propuesto por la cátedra que consistía en estegoanalizar un set de imágenes provistas con mensajes ocultos. Se detalla a continuación el análisis y las metodologías aplicadas para extraer los mensajes.

3.1. Stego Objects

La cátedra ofreció cuatro[17] imágenes en formato BMP, sin compresión y con una profundidad de color de 24 bits, es decir, exactamente el formato soportado por la aplicación desarrollada. En la *Tabla 3.1.1* se resumen los nombres, tamaños y dimensiones en píxeles de cada uno de los archivos.

Nombre del Archivo	Tamaño **[bytes]**	Dimensión **[píxeles]**
`kings.bmp`	`6750076`	`1500x1500`
`medianoche1.bmp`	`5376054`	`1120x1600`
`paris.bmp`	`4800054`	`1600x1000`
`sherlock.bmp`	`27528054`	`2480x3700`

Tabla 3.1.1: El nombre de los stego objects y su tamaño, en bytes.

3.2. Attack

Diferentes tipos de ataque han sido descriptos por otros autores (ver **[4]** y **[5]**). En particular, se decidió comenzar por una búsqueda de portadores originales (*Sección 3.2.1*), para luego aplicar una comparación utilizando editores hexadecimales y así reconocer el algoritmo aplicado (*Sección 3.2.2*). Sin embargo, esto no fue suficiente, ya que había mensajes esteganografiados mediante otros métodos, descriptos en las siguientes secciones.

3.2.1. Carrier Detection

Las imágenes se corresponden con posters de películas famosas, y con un paisaje de la ciudad de París. No obstante, era factible pensar que dichas imágenes se obtuvieron desde internet, ya que no son fotos personales.

[17] https://drive.google.com/uc?export=download&id=1h3z8w_ZcTGrVwmXN98RBDL2rCU5LEBfL

Google Images permite realizar búsquedas no solo por *keywords* sino también por dimensión exacta en píxeles. Una búsqueda rápida reveló tres[18] de cuatro imágenes (*Figura 3.2.1.1*). Las imágenes se encontraban en formato JPG como era esperado y si bien poseían la misma dimensión en píxeles, no se correspondían con el tamaño.

Figura 3.2.1.1: Los 3 portadores originales, hallados en internet. De izquierda a derecha se corresponden con los stego objects `medianoche1.bmp`, `paris.bmp` y `sherlock.bmp`.

Al abrir cada imagen con *Paint* y almacenarlas en formato BMP (que Paint genera sin compresión y con una profundidad de color de 24 bits), se obtienen el portador original, ya que la cantidad de bytes es coincidente con la *Tabla 3.1.1*.

Para la imagen no encontrada (`kings.bmp`), se aplicó el mismo proceso de transformación a BMP sobre una imagen[19] de las mismas dimensiones pero que difería solo en un logo de los premios *Academy Awards*, como se observa en la *Figura 3.2.1.2*.

Se notó que esta última imagen difería en algunos pocos bytes (22 bytes, para ser específicos).

---

[18]
`medianoche1.bmp`: https://i.pinimg.com/originals/74/55/56/7455567cedc04de0712c6b8a30a2a01e.jpg
`paris.bmp`: https://handluggageonly.co.uk/wp-content/uploads/2015/10/Paris-Restaurants-with-cool-views-5.jpg
`sherlock.bmp`: https://i.pinimg.com/originals/09/08/80/09088054f98f840eb89e747be6c12088.jpg

[19]
`kings.bmp`: http://4.bp.blogspot.com/-Br_B0jFnGZU/TfKH3bDcoXI/AAAAAAAAA7o/ubDC6xiDAXs/s1600/tksboxart.jpg

Figura 3.2.1.2: El stego object `kings.bmp` a la izquierda y el portador similar hallado a la derecha.

Junto con los portadores originales (o supuestamente originales), se procede a identificar el algoritmo de esteganografiado.

3.2.2. Algorithm Detection

Los algoritmos disponibles son LSB1, LSB4 y LSBE. Si bien es posible aplicar cada uno de forma sistemática sobre cada stego object, es preferible analizar con un editor hexadecimal y aplicar una comparación con las imágenes portadores halladas en la *Sección 3.2.1*.

- `kings.bmp`: en este caso el supuesto portador difería en 22 bytes del stego object. Debido a que la imagen posee dos franjas blancas, fue factible pensar que se había utilizado LSBE (ya que el blanco ofrece una gran oportunidad para codificar, debido a que su valor en RGB es 0xFFFFFF, lo que implica que cada píxel blanco permite ocultar 3 bits del mensaje oculto). La extracción no tuvo efecto.

  Al recorrer las diferencias, se observó sobre el final del stego object el mensaje:

  ```
  la password es ganador
  ```

  Que afortunadamente posee 22 bytes y explica las diferencias de tamaño. La técnica utilizada se denomina adición (según **[5]**), y claramente no debe utilizarse en la práctica porque no codifica ni oculta el mensaje bajo ningún aspecto.

- `medianoche1.bmp`: si se compara con el supuesto portador original, se pueden observar las trazas de los primeros 10 bytes del segmento COLOR_INDEX (byte 54 al 63, inclusive) de ambos archivos:

  ```
  45 3F 38 45 3F 38 4E 46 3F 57
  40 30 30 4D 4D 3B 52 41 3E 5D
  ```

  La traza superior es del supuesto portador original, la inferior del stego object. Se colorean las diferencias con respecto al posible portador. Se observa una marcada preponderancia a divergir en los 4 bits menos

significativos. Se aplica una extracción con LSB4, pero no se obtiene un archivo consistente, por lo cual se continúa con los demás archivos, ya que quizás no sea el algoritmo el que falle, sino que el stego object se puede encontrar cifrado.

- `paris.bmp`: se comparan de igual forma las trazas de 10 bytes:

```
1F 21 4A 10 12 3B 1E 1E 46 01
1E 20 4A 10 12 3A 1E 1E 46 00
```

  De los bytes que difieren, se aplicaron las transformaciones en hexadecimal $F \longrightarrow E$, $1 \longrightarrow 0$ y $B \longrightarrow A$, todas ellas correspondientes a transformaciones LSB de 1 bit, por lo cual se aplica una extracción LSB1, siendo en este caso satisfactoria (ver *Sección 3.2.3*).

- `sherlock.bmp`: al comparar los dos archivos, las primeras trazas de ambos al comienzo del byte 54 se encontraban rellenas con ceros y este patrón se extendía en un gran bloque. Debido a que es necesario esteganografiar el tamaño en primer lugar es completamente improbable que se haya utilizado LSB1 o LSB4 obteniendo dicho resultado.

  Se optó entonces, por aplicar LSBE y se logró una extracción satisfactoria.

3.2.3. Extraction

De los resultados de la *Sección 3.2.2*, se consiguió extraer un archivo PDF del stego object `sherlock.bmp`. El mismo expone las siguientes instrucciones:

```
al .png cambiarle la extension por .zip y descomprimir
```

El archivo PNG se corresponde con el extraído del stego object `paris.bmp`, el cual se refleja en la *Figura 3.2.3.1*.

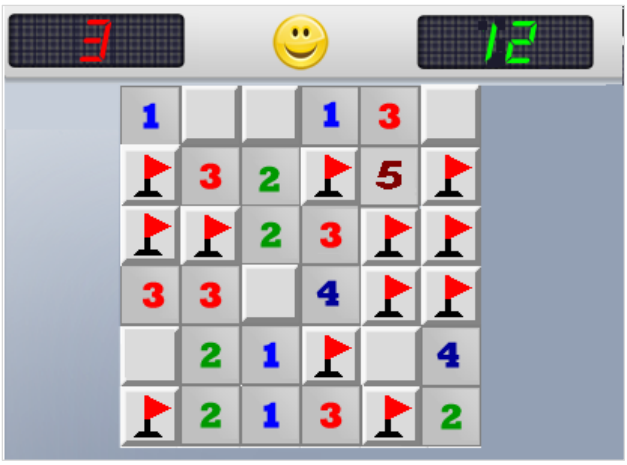


Figura 3.2.3.1: La imagen PNG extraída mediante LSB1 del stego object `paris.bmp`.

Al aplicar las instrucciones del PDF, se obtiene un nuevo archivo denominado `sol12.txt`, con las siguientes instrucciones adicionales:

1. Cada mina es un 1.
2. Cada fila forma una letra.
3. Los ASCII de las letras empiezan todos con 01.
4. Así encontrarás el algoritmo que tiene clave de 192 bits y el modo.
5. La password está en otro archivo.
6. Con algoritmo, modo y password hay un `.wmv` encriptado y oculto.

Siguiendo las instrucciones se asume que la password indicada es `ganador` (extraída de `kings.bmp`), y que el video en formato WMV se encuentra esteganografiado bajo LSB4 en `medianoche1.bmp`.

Para decodificar la imagen PNG y obtener el cifrador, se procede a construir la siguiente matriz:

$$\begin{pmatrix} 0 & 1 & 0 & x & x & 0 & 0 & x \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & x & 0 & 1 & 1 \\ 0 & 1 & x & 0 & 0 & 1 & x & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Esta matriz representa la imagen PNG decodificada junto con los bits iniciales con los cuáles se conforman los 6 caracteres de 8 bits en ASCII. Se coloca un 1 en las celdas que poseen una bandera, un 0 donde hay un número, y una x donde no se pueda determinar si hay o no una mina, es decir, en las celdas vacías. El *scoring* indica que quedan 3 minas, por lo cual 3 de las 6 celdas con x son minas, y las otras son celdas vacías.

De la fila 2, 3 y 6 se obtienen los caracteres directamente: `e`, `s` y `b`, respectivamente. De la primera fila, los caracteres posibles son $\{@, A, H, I, P, Q, X, Y\}$, pero debido a que solo están disponibles los algoritmos AES y DES, solo es posible que la fila 1 se corresponda con el carácter `A`. Por lo tanto, el algoritmo de cifrado es AES.

De la fila 4, los caracteres posibles son $\{C, K\}$, pero no hay ningún modo que comience con K, y por lo tanto la fila 4 se asocia al carácter `C`. La fila 5 permite construir los caracteres $\{D, F, d, f\}$, y dado que con los caracteres de las filas 4 y 6 solo es posible construir el modo CFB, y dado que de las x disponbiles, solo se asignó $x = 1$ en el último bit de la fila 1 y todavía restan dos minas, entonces este carácter debe ser `f`.

En definitiva, la secuencia de caracteres final es `AesCfb` y, por lo tanto, el stego object fue construido con cifrado AES de 192 bits, modo CFB y password `ganador`. Al extraer con LSB4 desde `medianoche1.bmp`, se obtiene el video en formato WMF con éxito.

3.2.4. Final Solver

Para automatizar todo el proceso se construyó un script que permite descargar todo el contenido (*i.e.*, los 4 stego objects), para luego extraer todos los mensajes

ocultos citados en este capítulo. El script solo requiere que se especifique la carpeta en la cual construir la solución:

```
$> ./solver.sh <folder>
```

El script opera en sistemas `*nix`. Es posible que requiera permisos de ejecución antes de ejecutarse, por lo cual es necesario ingresar el comando:

```
$> chmod 500 solver.sh
```

3.3. Issues to Analyze

La cátedra propuso un cuestionario de preguntas obligatorias a resolver dentro de los requerimientos[20]. Las mismas se responden a continuación:

1. **Para la implementación del programa `stegobmp` se pide que la ocultación comience en el primer componente del primer píxel. ¿Sería mejor empezar en otra ubicación? ¿Por qué?**

   Ver *Sección 2.6: Image Locator*. En resumen, sí, es significativamente mejor. Una desventaja no mencionada en la *Sección 2.6* es que, si el mensaje es aproximadamente del mismo tamaño que el espacio disponible carrier, el mensaje no se puede relocalizar demasiado. Este método es muy útil cuando el mensaje es pequeño con respecto al espacio disponible.

2. **¿Qué ventajas podría tener ocultar siempre en una misma componente? Por ejemplo, siempre en el bit menos significativo de la componente azul.**

   En **[7]**, Gupta *et. al.* explican que la componente azul de los canales RGB se ubica en el byte menos significativo y, por ende, cambios en este byte implican menor varianza reflejada en el valor absoluto real del RGB (menos saltos). Sumado al uso de un algoritmo LSB, se utilizarían entonces los bits menos significativos de este byte. El cambio percibido es menor, aunque la cantidad de espacio utilizado para ocultar es equivalente, ya que se utilizan 3 bits sobre el byte azul, no uno.

3. **Esteganografiar un mismo archivo en un BMP con cada uno de los tres algoritmos, y comparar los resultados obtenidos. Hacer un cuadro comparativo de los tres algoritmos estableciendo ventajas y desventajas.**

   Se esteganografió la imagen de la *Figura 3.3.1* dentro de la foto principal[21] del README del repositorio de la aplicación. La foto se encuentra en formato PNG, tiene una dimensión de 220x220 píxeles y pesa exactamente 43104 bytes.

[20] https://github.com/agustin-golmar/Wolf-in-a-Sheep/blob/master/doc/(2018) Requirements I.pdf
[21] https://github.com/agustin-golmar/Wolf-in-a-Sheep/blob/master/res/image/readme-header.bmp

Figura 3.3.1: El mensaje oculto.

Se intentan aplicar los 3 algoritmos disponibles, sin password y por lo tanto sin cifrado, siendo que el carrier posee 1358550 bytes totales y un payload de 1358496 bytes (es decir, sin el *header* de 54 bytes). Posee un *row padding* de 3 bytes debido a que la dimensión de cada fila en bytes no es múltiplo de 4, por lo cual algunos cambios introducidos por el algoritmo LSB no serán percibidos (los que caigan en esta área).

Se pueden obtener las siguientes deducciones de utilizar uno u otro algoritmo:

- **LSB1**: la imagen permanece inalterada a simple vista, inclusive al realizar *zoom* sobre ella. Es el mejor método si el mensaje a ocultar es pequeño.

- **LSB4**: si un mensaje puede ser albergado por un portador bajo LSB1, también lo hará bajo LSB4, ya que la relación de tamaño es exactamente 4, es decir, LSB4 almacena 4 veces más que LSB1. Sin embargo, al aplicar un acercamiento sobre la esquina inferior izquierda de la imagen (la zona donde se comienza a esteganografiar), es posible notar una pixelación o distorsión en la imagen con respecto a la obtenida mediante LSB1 (ver *Figura 3.3.2*).

- **LSBE**: en este caso, la imagen ni siquiera puede insertarse en el archivo, ya que la aplicación expresa que el carrier puede albergar hasta 28789 bytes, pero el mensaje oculto final es de 344904 bytes, 8 veces más que el tamaño de la imagen original (+13 bytes de metadata). Una dificultad del algoritmo LSBE es exactamente la dependencia que hay entre el tamaño disponible y la preponderancia de ciertos valores a aparecer dentro del carrier (en este caso, los valores 0xFE y 0xFF).

La ventaja de LSB1 es notoria con respecto a los demás algoritmos. Ofrece un balance entre el nivel de percepción a simple vista y la cantidad de información que puede ofrecer para albergar mensajes ocultos. LSB4 es un caso más extremo en el que se sacrifica calidad, pero se obtiene suficiente espacio. LSBE funciona a la inversa: la calidad es superior, inclusive frente al algoritmo LSB1, pero el espacio disponible depende directamente de la imagen utilizada.

Figura 3.3.2: La imagen superior se corresponde con LSB1, la inferior con LSB4. Es fácil apreciar una pixelación debajo de la franja roja.

4. **Para la implementación del programa `stegobmp` se pide que la extensión del archivo se oculte después del contenido completo del archivo. ¿Por qué no conviene ponerla al comienzo, después del tamaño de archivo?**

   En **[5]** se detalla un ataque que consiste en hallar "firmas" (*signatures*) o marcas comunes que los algoritmos de esteganografiado dejan al ocultar un mensaje. Teniendo en cuenta que los primeros 4 bytes (sin cifrado) se corresponden con el tamaño del payload y estos dependen del mensaje, podría considerarse que un observador percibiría dichos bytes como parte del mensaje o como información aleatoria. Sin embargo, ubicar la extensión al principio, implicaría que el principio de esta siempre se encontraría ubicado en el mismo byte. La mayoría de las extensiones poseen solo 3 caracteres y la cantidad de formatos que un usuario podría ocultar no representa un gran universo de extensiones posibles (pensando en formatos de imágenes, texto, compresión, audios y videos). Si un atacante tuviera suficientes *stego objects*, podría compararlos y obtener así un patrón de bits sobre la zona de ocultamiento asociada a la extensión. Por lo tanto, la mejor opción es colocarla al final, cifrar el mensaje, o no colocarla en lo absoluto.

5. **Explicar detalladamente el procedimiento realizado para descubrir qué se había ocultado en cada archivo y de qué modo.**

   Ver *Sección 3.2*.

6. **¿Qué se encontró en cada archivo?**

   Ver *Sección 3.2*.

7. **Algunos mensajes ocultos tenían, a su vez, otros mensajes ocultos. Indica cuál era ese mensaje y cómo se había ocultado.**

   Ver *Sección 3.2*.

8. **Uno de los archivos ocultos era una porción de un video, donde se ve ejemplificado una manera de ocultar información. ¿Cuál fue el portador?**

El video es sobre un capítulo de la serie *Bones*[22]. El portador era una imagen de un hombre despidiéndose de su novia, pero detectan que la foto era demasiado grande y que por lo tanto debía ser *carrier* de un archivo adicional.

9. **¿De qué se trató el método de estenografiado que no era LSB? ¿Es un método eficaz? ¿Por qué?**

   En particular, se observan dos métodos que difieren del algoritmo LSB genérico. En primera instancia, el stego object `kings.bmp` tenía un agregado de 22 bytes sobre el final del archivo en ASCII, completamente extraíbles y sin codificación alguna.

   En el segundo caso, la imagen PNG extraída contenía adosado un archivo en formato ZIP. Este método consiste en agregar el contenido de un mensaje al final de otro, aprovechando el hecho de que las aplicaciones que operan sobre dichos formatos hacen caso omiso de la información que no se adapta al formato original. Es decir, un visualizador de imágenes PNG lee el bloque inicial asociado a la imagen en cuestión y descarta el contenido más allá del final de esta. Luego, el descompresor ZIP escanea el archivo hasta encontrar la firma del formato (`50 4B 03 04`, esta firma puede encontrarse en la posición 0x0000A97B), descartando los bytes iniciales (la imagen).

   Este método se denomina *adición* (en ambos casos). En **[3]** se define un posible conjunto de proposiciones que disponen los requerimientos básicos que un esquema esteganográfico debe tener para efectivamente ocultar información. Se enumeran a continuación (sin el inciso asociado a la técnica de *watermarking*):

   1. La integridad del mensaje oculto debe permanecer inalterada luego de ocultar el mismo en un portador.
   2. El stego object debe permanecer inalterado a simpe vista, con respecto al portador original.
   3. Siempre se asume que el atacante sabe que hay un archivo oculto dentro del stego object.

   El método de adición viola la proposición 2, ya que los archivos carrier y stego difieren en tamaño y es fácil percibir esta modificación. Además, el método de adición se hace menos factible cuando se desean agregar grandes cantidades de información, porque el portador crece significativamente.

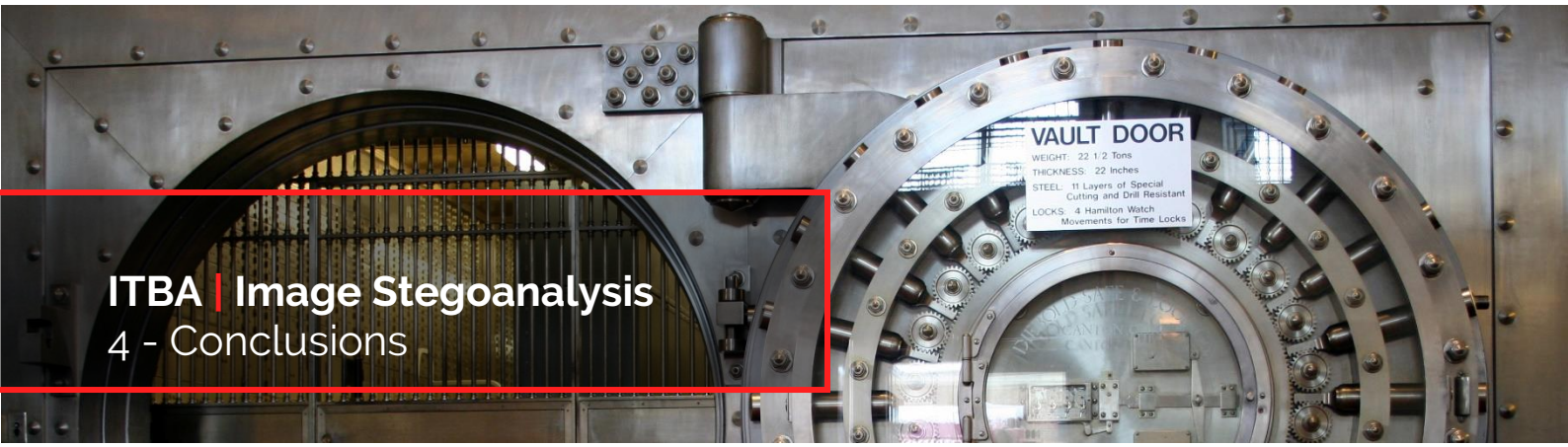
   Además, este método de esteganografiado es bien conocido y puede encontrarse en varios sitios de internet por lo cual es factible pensar que un

[22] https://www.imdb.com/title/tt0460627/

atacante sabe perfectamente como determinar si un archivo posee un mensaje oculto bajo esta metodología.

10. **¿Qué mejoras o futuras extensiones harías al programa `stegobmp`?**

    Ver *Sección 2.6*.



4. Conclusions

Las técnicas esteganográficas proveen un nivel adicional de seguridad al ocultar la existencia del mensaje en sí mismo, pero debe entenderse que no son un reemplazo directo de las técnicas criptográficas de cifrado, sino solo un complemento. Un cifrado provee propiedades matemáticas que no se corresponden bajo ninguna medida con las propiedades que la esteganografía provee.

En particular, durante el análisis realizado en la *Sección 3*, se observó que la obtención del supuesto portador original ofrece una medida casi directa de detección y resolución del método esteganográfico utilizado. Una posible solución a dicho problema se puede inferir de **[5]**. Cárdenas determina que existen 3 tipos de esteganografiado según la forma de embeber el mensaje en el portador:

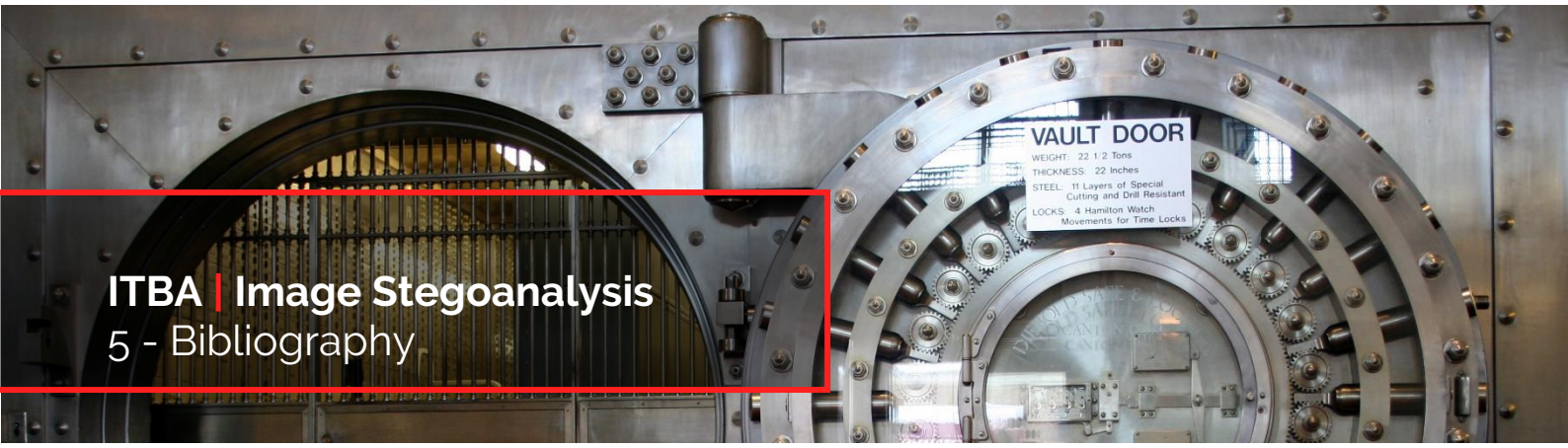
- **Adición**: se agrega el mensaje oculto en el portador, tal cual se observó en la *Sección 3.2.2* y *3.2.3*.
- **Sustitución**: se modifica el portador reemplazando partes de este, al igual que lo hace LSB o alguna técnica similar cómo las provistas en **[4]** y **[6]**.
- **Generación**: se genera un nuevo portador y se utiliza para ocultar el mensaje con alguna técnica (ya sea adicional, o durante la misma generación).

El método de adición debe evitarse, es el más detectable y no codifica ni ofusca la información bajo ningún aspecto. El método de sustitución resuelve el problema del método de adición al codificar la información sin ocupar espacio adicional y sin explicitar el contenido de esta. Sin embargo, es susceptible a ataques por portador conocido (*known-carrier attack*).

Para resolver este último conflicto, el método generativo confecciona un nuevo portador como función del mensaje oculto, y por lo tanto el portador generado es único (si el algoritmo utiliza algún mecanismo probabilístico). Si bien esta técnica no es trivial, se puede aproximar aplicando un método de sustitución sobre un portador generado solo para el transporte (por ejemplo, una foto personal). De esta forma se elimina a posibilidad de que un atacante pueda hallar una copia original a través de internet.

No obstante, en este documento no se hizo hincapié en el hecho de que el stego object podría requerir cierto nivel de resistencia frente a modificaciones de este (escalado, compresión con pérdida, rotación, etc.), aunque se sugirió como mejora futura la inclusión de un algoritmo MAC. Sin embargo, la resistencia de la que se habla no es equivalente a la que provee un MAC. El MAC provee una forma de verificar que el mensaje cambió, pero no permite recuperar el mensaje original de igual forma. Este problema se resuelve utilizando

técnicas de watermarking e idealmente fingerprinting (el *watermarking* es un caso especial de *fingerprinting*). En este caso, un atacante podría modificar el stego object sin poder romper el mensaje oculto y el stego object al mismo tiempo.



5. Bibliography

[1] **"Introduction to Modern Cryptography"**. Jonathan Katz and Yehuda Lindell. *1st Edition. CRC Press. 2007*.

[2] **"Efficient Method of Audio Steganography by Modified LSB Algorithm and Strong Encryption Key with Enhanced Security"**. R. Sridevi, Dr. A. Damodaram and Dr. Svl. Narasimham. *Vol. 5, N° 6, Journal of Theoretical and Applied Information Technology. JNTUCEH, Hyderabad. 30th June, 2009*.

[3] **"Steganography and Digital Watermarking"**. Jonathan Cummins, Patrick Diskin, Samuel Lau and Robert Parlett. *School of Computer Science, The University of Birmingham. 2004*.

[4] **"Análisis de Técnicas Esteganográficas y Estegoanálisis en Canales Encubiertos, Imágenes y Archivos de Sonido"**. Gustavo A. Isaza E., Carlos Alberto Espinosa A. and Sandra M. Ocampo C. *Vector, Vol. 1, N° 1, p29-38. December 2006*.

[5] **"Esteganografía"**. Dr. Roberto Gómez Cárdenas. *Maestría en Seguridad Informática. ITESM-CEM. Tecnológico de Monterrey*.

[6] **"Exploring Steganography: Seeing the Unseen"**. Neil F. Johnson and Sushil Jajodia. *George Mason University. February, 1998*.

[7] **"Enhanced Least Significant Bit Algorithm for Image Steganography"**. Shilpa Gupta, Geeta Gujral and Neha Aggarwal. *IJCEM International Journal of Computational Engineering & Management. Vol. 15, Issue 4. July, 2012*.