Steganography & Encryption - The Hiding of Data

William Banks – 2000927@uad.ac.uk

Introduction to Security – CMP110

BSc Ethical Hacking Year 1

2020/21

Abstract

Steganography and encryption are two sides of the same coin. They have a common purpose: the concealment of data, but the way they go about achieving it is quite different. Encryption seeks to make data unreadable, while steganography seeks to hide it from being read. This paper will give a brief introduction to steganography and encryption in the modern world, and introduce and explain in depth a common technique for steganography involving hiding data in the pixels of an image, and a strong modern encryption algorithm standardised by NIST, US - the Advanced Encryption Standard (AES). It will also discuss and explain how they might be implemented, including a look at a dedicated AES instruction set, and provide an implementation in C++ of those types of steganography and encryption working together.

Contents

1. Introduction

1.1. Background

Steganography and encryption operate in very different ways in order to perform their function.

Encryption is the transformation of data into an alternate unintelligible form in an attempt to hide it's content from unauthorised parties (Cloudflare, no date). The process of encryption is the transformation of plaintext into ciphertext via an encryption algorithm. Note that neither the plaintext nor ciphertext have to actually be *text*, they can be any form of data. These algorithms typically use a key (which can be a sequence of characters or numbers) to encrypt and decrypt - Data that has been encrypted cannot be decrypted without the key, in a strong encryption algorithm. Some algorithms use the same key to encrypt as they do to decrypt, they are called symmetric-key algorithms. Others use a different key to decrypt as they do to encrypt, and so are called asymmetric-key algorithms, or public key algorithms because of how they are usually used.

Steganography is the hiding of data in plain sight, in a way that it's not obvious there's any data hidden - it aims to hide the presence of data rather than just making it unreadable. An example of this is encoding a message by creating a fake message - the *cover* message - where the first character of each word in the fake message is the steganographically hidden message.

> *"Waitrose encrypts Arthur's recyclables, except DES is so crackable omitting Vs embarrassingly. Retail enterprises decrypt files like ELF executables and transform, omnipotent neovim critic empires"*

The above message contains a hidden message: "We are discovered flee at once". With steganography, if it is known how the message is hidden, it can be read - Because of this, steganography is often not the best choice for security. However, combined with encryption, it can be very powerful indeed.

1.2. Aim

The project undertaken alongside this report aims to create a cross-platform steganography tool that is also capable of strong AES encryption. The tool should be able to perform LSB image steganography utilising powerful compression to shrink the embedded data, it should be able to embed and extract with 100% accuracy. In this report I will also discuss how encryption and steganography could potentially be used to disguise malware and other potentially malicious data, and go over some possible countermeasures.

2. Overview of Techniques

This section shall explain some steganography and encryption techniques, along with prerequisite knowledge. Basic knowledge of binary, hexadecimal and bitwise operations is assumed.

2.1. Steganography

The example hidden message in the example in the 1.1. Background section is a basic form of null cipher, a type of cipher that hides a message in amongst many *nulls*, characters that are simply noise, they don't have any significance except being there for the actual message to hide amongst. Hiding data amongst irrelevant noise is a very prevalent and widely used idea in steganography - indeed, some claim that is what steganography *is* (Stanger, 2020)

Steganography is not only used for covert communication, however - It can be used for the purpose of identification, annotation and copyright (Bender *et al.*, 1996), such as in the case of embedding watermarks to assert ownership, authenticity or to encode DRM rules (Krzyzanowski, 2020)

One common steganography method is LSB image steganography - Encoding data in the Least Significant Bit (in a byte, the bit that has the lowest value) of the pixels of a raster image (Singh, Singh, Singh, 2015). This is a lossless technique, but requires that the image is not compressed or edited in any way, as that would destroy or corrupt the data.

LSB image steganography is what will be focused on in this report, and is the kind implemented in the steganography tool "Hideit" developed alongside this report.

2.1.1. Raster Images

Raster images, also known as bitmap images, bitmaps for short, come in a variety of formats, but all of them store image data the same way: As a sequence of numbers representing a grid of pixels. A pixel can be thought of as a single "unit" of an image - It is a square (although not in all cases (Pirazzi, no date) but for the purposes of this report it's irrelevant) block of a single colour. How the colour is represented depends on the format, some common formats are 1-bit monochrome, 8-bit greyscale, 16-bit RGB, 24-bit RGB and 32-bit ARGB. Each format is comprised of a number of "channels", each channel representing a component of colour.

- In 1-bit monochrome, there is just 1 channel, taking up just 1 bit. That bit is either 0 (black) or 1 (white). This format is impossible to do LSB steganography on without perceptibly changing the image.
- In 8-bit greyscale, there is just 1 channel, the grey channel. It uses 1 byte and so ranges from 0 (black) to 255 (white).
- In 16-bit RGB, there are 3 channels, one for red, one for green and one for blue. Each channel takes up 5 bits, and ranges from 0 (black) to 31 (pure red, green or blue respectively).
- In 24-bit RGB, it's the same as 16-bit RGB, except each channel takes up a full byte, and so ranges from 0 to 255.
- In 32-bit ARGB, it's the same as 24-bit RGB except there's an extra channel - alpha. This determines the transparency of the colour, ranging from 0 (100% transparent) to 255 (0% transparent).

Figure 1: Colour selection dialog of popular pixel art graphics editor Aseprite

Example: A 24-bit colour with RGB values 55, 148 and 110 is represented by the binary numbers:

Red: 55 - 00110111
Green: 148 - 10010100

Blue: 110 - 01101110
These are placed consecutively in memory:

0	0	1	1	0	1	1	1	1	0	0	1	0	1	0	0	0	1	1	0	1	1	1	0

And can be accessed individually, as bytes, to operate on.

Colours can also be represented in hexadecimal - 0x37946e for this particular shade of blue-green.

Raster images store pixels, or rather their bytes, consecutively as well, in a long list, or array, indexed from 0 to n, where n is the image width * image height * bytes per pixel.

Pixel	0			1			2		
0	0	1	2	3	4	5	6	7	8
1	9	10	11	12	13	14	15	16	17
2	18	19	20	21	22	23	24	25	26

2.1.2. Perceptibility

If a colour has enough bits per channel, it is possible to change the LSB and not perceptibly change the colour.

Take the 2 24-bit RGB colours on the right. The colour on the left is 0x45283c, the other is 0x44293d - The LSB of each channel is different between them, and even with the colours right next to each other it's difficult to tell there's even 2 distinct colours there. Not to mention that on larger images with many millions of pixels, such a slight change in colour between an image and that same image with the LSB of each channel changed is unlikely to be noticeable.

Take the two images above. The image on the right is a copy of the one on the left, but has had it's LSBs inverted. The difference is almost completely imperceptible. And when using LSB steganography to hide data in an image, each channel is highly unlikely to always have it's LSB flipped for every pixel, as that would require the LSBs of the data to hide and the cover image to hide it in lining up exactly. Not to mention that when hiding data in a cover image, the original will almost certainly not accompany it.

2.1.3. Bit Manipulation in C++

In C++, to write to a specific bit, bitwise operators need to be used - In particular, bit shift (<<, >>), AND (&), OR (|) and NOT (~).

For example, to write 1 to bit 4 of the byte in the below table, or to "set it", the following operations are done:

Bit index	7	6	5	4	3	2	1	0
Byte	0	1	1	0	0	0	1	0

Create a bitmask by shifting 1 to the left 4 places (4 is the bit index):

Bit index	7	6	5	4	3	2	1	0
Bitmask	0	0	0	1	0	0	0	0

Now there are 2 different things to do depending on whether the bit is being set to 1 ("set") or 0 ("unset"). Here it's being set, so the next step is to OR the bitmask with the byte, which will produce the result in the table below:

Bit index	7	6	5	4	3	2	1	0
Byte	0	1	1	1	0	0	1	0

Bit 4 has been set.

Doing that in C++ would look like this:

```
byte = byte | (1 << bitIndex);
```

Where bitIndex is 4 for this case.

To do the reverse of that now and unset it, the following operations must be carried out:

Create an inverse bitmask by shifting 1 to the left 4 places and NOTing it:

Bit index	7	6	5	4	3	2	1	0
Bitmask	1	1	1	0	1	1	1	1

Then, AND the bitmask with the byte:

Bit index	7	6	5	4	3	2	1	0
Byte	0	1	1	0	0	0	1	0

And bit 4 has been unset.

In C++ this would be:

```
byte = byte & ~(1 << bitIndex);
```

Where bitIndex is 4 for this case.

Reading a bit is not a dissimilar process.

For example, to read bit 1 from the byte in the above examples:

Shift the byte to the right 1 place:

Bit index	7	6	5	4	3	2	1	0
Byte	0	0	1	1	0	0	0	1

Then simply AND it with 1 and the value of bit 1 is produced, in this case 1:

Bit index	7	6	5	4	3	2	1	0
Byte	0	0	0	0	0	0	0	1

In C++, this process would be:

```
bool result = (byte >> bitIndex) & 1;
```

Where bitIndex is 1 and result comes out as true if the bit was set and false if it wasn't. For this example of course it'd come out as true.

2.2. Encryption

The encryption algorithm focused on in this report is the Advanced Encryption Standard (AES), a subset of Rijndael, a symmetric-key encryption algorithm developed by Belgian cryptographers Joan Daemen and Vincent Rijmen. Technically speaking, AES isn't an algorithm, it's a standard, but for all intents and purposes it can be referred to as one (as it is in the official standard (FIPS, 2001), which I shall be referring to extensively throughout the following sections) and will be in this report.

Rijndael is a block cipher, with variable block length and variable key length. The only difference between AES and Rijndael is the range of supported values for the block length and the key length (Daemen, Rijmen, 2020).

Block ciphers operate on each block independently, they perform the same transformations on each block. A block is simply a subsection of the full amount of data to be encrypted/decrypted. So say 32 bytes were to be encrypted using a block cipher, first the 32 bytes would be split into separate blocks to be operated on - The size of the blocks depends on the cipher. With Rijndael, the block size can be any multiple of 32 bits between 128 and 256, but AES fixes it to 128 bits.

AES does allow different key sizes, though it's not as flexible as Rijndael in the ones it does allow. The key size can either be 128, 192 or 256 - AES using these key lengths is denoted AES-128, AES-192 and AES-256 respectively. The version implemented in Hideit is AES-128.

The AES algorithm has two main parts - The key expansion and the actual encryption, referred to as the cipher. The cipher describes the operations on each block, while the key expansion describes the generation of a set of keys, called round keys, from the input key, to be used in the cipher. The generated set of round keys is called the key schedule.

In the cipher, there are a number of rounds that are performed on the input block. Each round uses one of the round keys, so there needs to be as many round keys as there are rounds, plus one extra one for reasons covered in the 2.2.3. Cipher section. The number of rounds performed is specified in the AES specification to be dependent on the key length, the table below describes the relation:

	Key Length	Block Size	Number of Rounds
AES-128	128	128	10
AES-192	192	128	12

AES-256	256	128	14

Figure 2: Key-Block-Round combinations table (FIPS, 2001)

The way the cipher is used is called the mode of operation, and different modes of operation produce different results, and some are considered more secure than others.

The sections following 2.2.1. Modes of Operation will explain how the AES algorithm works, specifically in the context of AES-128, using the CTR mode of operation.

2.2.1. Modes of Operation

Back in 2001 when AES was created, 5 modes of operation were standardised: ECB (Electronic Code Book), CBC (Cipher Block Chaining), CFB (Cipher FeedBack), OFB (Output FeedBack) and CTR (Counter) (Blazhevski *et al*, 2013).

Only the ECB and CTR modes shall be explained here, as they are the most common, and the CTR mode is thought to be the one that provides the strongest encryption, and is subsequently the one that will be focused on throughout the rest of the sections.

AES with a specific mode of operation is denoted by putting the mode of operation after AES, with a separating slash - AES using CTR mode of operation is AES/CTR.

2.2.1.1. ECB Mode

ECB mode is considered the simplest mode of operation, but also the least secure.

In ECB mode, given an array of bytes to be encrypted and a key, the round keys are computed (via key expansion) and then each block of the input array of bytes is encrypted with those round keys.

This leads to some problems however. First, the input array might not be an exact multiple of the block size, meaning padding must be done, and the real size must be stored somehow.

Secondly, a security problem: If two blocks are identical, then those two blocks when encrypted will still be identical. This can give information about the plaintext to attackers, and so ECB mode is generally not considered secure (Edney, Arbaugh, 2003).

2.2.1.2. CTR Mode

CTR mode is not much more complicated than ECB mode, but is far more secure.

First, a 128-bit counter is initialised to some value based off of the key.

Then, the input array of bytes is broken down into blocks, much like in ECB. Now it operates quite differently to ECB in that it isn't the actual blocks of the input that get encrypted, it's the counter that gets encrypted.

For each block in the input array of bytes:

- The counter is encrypted using AES
- The current block is XORed with the encrypted counter
- The counter is incremented

This means that, since for each block the counter is different, 2 identical blocks are never encrypted to be identical afterwards, and since it's not actually the input blocks that are getting encrypted, the input doesn't need to be padded if it doesn't fit into a multiple of 16 bytes, mitigating the problems with ECB.

Yet another advantage of counter mode is that it is reversible, as XOR is reversible - Double encrypting an array of bytes with the same key will produce exactly the initial array. Because of this, only AES encryption needs to be used/implemented, which will be reflected in this report and the tool developed alongside - I won't discuss the decryption process using AES, as it is unnecessary.

2.2.2. Key Expansion

The AES key expansion describes the generation, or *expansion*, of round keys from the 128-bit input key, referred to as the cipher key in this section.

Each round key is made up of 4 32-bit words, w_0 to w_3 - To a total of 128 bits. The first round key, rk_0, is simply initialised to the value of the cipher key.

The following round keys, rk_1 to rk_{10}, are derived as follows:

For each of the round keys rk_1 *to* rk_{10}, words w_1, w_2 and w_3 are computed as the sum of the corresponding word in the previous round key and the preceding word in the current round key (Edney, Arbaugh, 2003). The sum, along with all arithmetic in AES, is Finite Field Arithmetic, which won't be explained in detail here but to find out more you can read this chapter of this book or this paper. In Finite Field Arithmetic, addition is the same as the XOR bitwise operation.

Example calculation: $rk_7{:}w_2 = rk_7{:}w_1 \text{ XOR } rk_6{:}w_2$

Where $rk_m{:}w_n$ refers to the nth word in key m. So a general form of the above calculation would look like:

$rk_m{:}w_n = rk_m{:}w_{(n-1)} \text{ XOR } rk_{(m-1)}{:}w_n$

Word w_0 is a little different to compute, and requires a value called the round constant, or $rcon[i]$, which comes from the calculation $rcon[i] = 2^i$ in the finite field $\text{GF}(2^8)$ (refer to previously mentioned publications for information on this). The values for this from 1-10 are below:

i	1	2	3	4	5	6	7	8	9	10
$rcon[i]$	1	2	4	8	16	32	64	128	27	108

AES-128 uses up to $rcon[10]$, AES using different key lengths will use different amounts of $rcon[i]$.

Computing w_0 also requires a rotation Rotr8 (rotate right 8 bits) function that operates on the bytes of a word, defined as:

$\text{Rotr8}([a_0, a_1, a_2, a_3]) = [a_3, a_0, a_1, a_2]$

Where a_n is the nth byte in a word. (Gueron, 2010)

And then finally, one more function, SubWord, this operates on the bytes of a word again, it performs a substitution on each byte, replacing it with the value in the lookup table which can be found at 2.2.3.1. SubBytes() Figure 3.

$\text{SubWord}([[a_0, a_1], [b_0, b_1], [c_0, c_1], [d_0, d_1]]) = [\text{sbox}(a_0, a_1), \text{sbox}(b_0, b_1), \text{sbox}(c_0, c_1), \text{sbox}(d_0, d_1)]$

Where $[a_0, a_1]$ is byte a split into nibbles, and $\text{sbox}(i, j)$ is the value in the s-box at row i and column j.

The calculation of w_0 is as follows:

$rk_m{:}w_0 = rk_{(m-1)}{:}w_0 \text{ XOR } \text{SubWord}(\text{Rotr8}(rk_{(m-1)}{:}w_3)) \text{ XOR } rcon[m]$

The round keys can either all be generated before performing any encryption with the cipher, or each one can be generated just in time for its respective round in the cipher, for every block. The former is more computationally efficient, while the latter is more space efficient.

2.2.3. Cipher

The cipher operates independently on each 128-bit block of the input (the counter as this is talking about AES/CTR) making use of the 11 128-bit round keys, applying the same transformations to each block. Since this section is talking on a per-block basis, the block being operated on will be referred to as the state.

It performs transformations in rounds, 128-bit AES uses 10 rounds.

Before the first round, the first round key is added to the state - This is the reason for 11 round keys and 10 rounds. As mentioned previously, all arithmetic operations in AES take place in the finite field $GF(2^8)$, so addition is synonymous with the XOR bitwise operation.

After the initial addition of the round key, some functions need to be defined for the individual transformations: SubBytes(), ShiftRows(), MixColumns() and AddRoundKey().

For the first 9 rounds of the cipher, these transformations are applied in this order:

- SubBytes()
- ShiftRows()
- MixColumns()
- AddRoundKey()

The last round is a little different, excluding the MixColumns() transformation:

- SubBytes()
- ShiftRows()
- AddRoundKey()

The following sections define these transformations.

2.2.3.1. SubBytes()

The SubBytes() transformation is a byte-wise substitution, using a substitution table (the Rijndael s-box). The s-box is constructed using the multiplicative inverse in the finite field $GF(2^8)$, with element 0 being mapped to itself, and an affine transformation:

$$\begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

(Daemen, Rijmen 2020)

Although interesting, it's not necessary to know how the s-box is constructed however, as it is a constant so can be used precalculated. The precalculated s-box is below:

	00	**01**	**02**	**03**	**04**	**05**	**06**	**07**	**08**	**09**	**0a**	**0b**	**0c**	**0d**	**0e**	**0f**
00	63	7c	77	7b	f2	6b	6f	c5	30	1	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	4	c7	23	c3	18	96	5	9a	7	12	80	e2	eb	27	b2	75
40	9	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	0	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	2	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	6	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	8
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	3	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 3: The Rijndael Substitution Box (FIPS, 2001)

When substituting byte a using the above table, the most significant nibble (4 bits) determines the column, and the least significant nibble determines the row. So if *a* is constructed from nibbles a_0 and a_1, most and least significant respectively, and the value in the s-box at row i and column j is $\mathrm{sbox}(i, j)$ the substitution is as follows:

$a = \text{sbox}(a_0, a_1)$

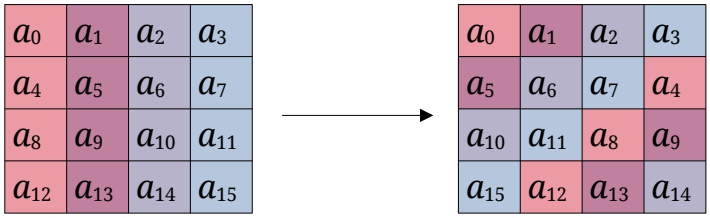
This substitution is performed individually on each byte of the state.

2.2.3.2. ShiftRows()

This transformation is applied to each row of the state, where the state can be put into a table, and the nth byte of the state is given by a_n:

a_0	a_1	a_2	a_3
a_4	a_5	a_6	a_7
a_8	a_9	a_{10}	a_{11}
a_{12}	a_{13}	a_{14}	a_{15}

The transformation rotates (cyclically shifts) each row to the left by the row number, starting at 0. So for example, the first row is left unchanged, while the 3rd row is rotated to the left by 2 places.



2.2.3.3. MixColumns()

This transformation operates on each column individually, putting the state into the table like in the ShiftRows() transformation, but this time labelling the bytes by their position in the table:

$a_{0,0}$	$a_{1,0}$	$a_{2,0}$	$a_{3,0}$
$a_{0,1}$	$a_{1,1}$	$a_{2,1}$	$a_{3,1}$
$a_{0,2}$	$a_{1,2}$	$a_{2,2}$	$a_{3,2}$
$a_{0,3}$	$a_{1,3}$	$a_{2,3}$	$a_{3,3}$

This transformation is a matrix multiplication:

$$\begin{bmatrix} a_{n,0} \\ a_{n,1} \\ a_{n,2} \\ a_{n,3} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} a_{n,0} \\ a_{n,1} \\ a_{n,2} \\ a_{n,3} \end{bmatrix}$$

(FIPS, 2001)

Where n is the number of the column being operated upon. Using the normal matrix multiplication method, this expands to:

$a_{n,0} = 2 * a_{n,0} + 3 * a_{n,1} + 1 * a_{n,2} + 1 * a_{n,3}$

$a_{n,1} = 1 * a_{n,0} + 2 * a_{n,1} + 3 * a_{n,2} + 1 * a_{n,3}$

$a_{n,2} = 1 * a_{n,0} + 1 * a_{n,1} + 2 * a_{n,2} + 3 * a_{n,3}$

$a_{n,3} = 3 * a_{n,0} + 1 * a_{n,1} + 1 * a_{n,2} + 2 * a_{n,3}$

Where all the operations are in the finite field $\mathrm{GF}(2^8)$. Addition is XOR, but multiplication is a little bit more complicated. This report won't go into the details, which can be found here, but multiplication in $\mathrm{GF}(2^8)$ can be calculated using a series of bit shifts and XORs.

The process for calculating $a * b$ is as follows:

- Start with an accumulator, c, initialised to 0.
- Take each bit of b in turn, starting with the most significant bit:
  - If the most significant bit of c is 1, then XOR c with the bounding irreducible polynomial of $\mathrm{GF}(2^8)$, which is 00011011 or 0x1b.
  - Multiply c by 2 (shift left).
  - If the current bit in b is 1, XOR c with a.
- The final value of c is the answer.

(Edney, Arbaugh, 2003)

2.2.3.4. AddRoundKey()

This is the simplest transformation, and is identical to the one done before the first round. It is the simple addition of the round key to the state, which in finite field arithmetic is the same as XOR.

2. Procedure

2.1. Overview of Procedure

Along with this report, a steganography & encryption tool was created, named Hideit.

The tool has 2 main parts, the LSB image steganography part and the AES-128 encryption algorithm implementation. The tool was created in C++ using the GNU Compiler Collection (GCC) C++ compiler, and although only tested on x86_64 Linux, it was designed to be compatible between platforms that support GCC, but unfortunately isn't between architectures, because of the raw assembly implementation of the AES - Saying that, it should work on 32-bit x86.

2.2. LSB Image Steganography Implementation

The implementation of the LSB steganography part of the tool Hideit uses 2 classes named BitWriter and BitReader. The purpose of these classes is to have an easy to use way to read and write single bits, and have objects to keep track of what exact bit in an array of bytes is currently being read/written.

The classes are very similar, they both have several variables to keep track of progress through an array of bytes m_Bytes:

```
uint8_t* m_Bytes             // The array of bytes to work with
uint64_t m_NumBits             // The number of bits of m_Bytes that
should be used for reading/writing
uint8_t m_BitsPerByte         // The number of bits to read/write per
byte
bool m_StartAtLSB         // Whether to start reading/writing at the LSB
of every byte
bool m_Done              // Whether m_NumBits has been read/written
uint8_t m_BitIndex          // The index of the bit to read/write next in
the current byte
uint64_t m_OverallBitIndex     // The overall index of the bit in the
```

```
whole of the m_Bytes array
```

What differs of course is their functions. The BitReader has a function "ReadNextBit" which uses the method of reading a bit discussed in the section 2.1.3. Bit Manipulation in C++, while the BitWriter uses the 2 methods for writing a bit in it's function "WriteNextBit" discussed in that same section - These methods both modify several of the above variables to point to the next bit, whichever that may be.

The code for these functions is below:

```
bool BitReader::ReadNextBit() {
    uint8_t byte = m_Bytes[m_ByteIndex];
    uint8_t bitIndex = m_StartAtLSB ? m_BitIndex : 7 -
m_BitIndex;

    bool value = (byte >> bitIndex) & 0x1;

    m_BitIndex++;
    if(m_BitIndex >= m_BitsPerByte) {
      m_BitIndex = 0;
      m_ByteIndex++;
    }
    m_OverallBitIndex++;
    if(m_OverallBitIndex >= m_NumBits) {
      m_Done = true;
    }

    return value;
}
```

```
void BitWriter::WriteNextBit(bool value) {
```

```
    uint8_t& byte = m_Bytes[m_ByteIndex];
    uint8_t bitIndex = m_StartAtLSB ? m_BitIndex : 7 -
m_BitIndex;

    if(value) {
      byte = byte | (0x1 << bitIndex);
    } else {
      byte = byte & (~(0x1 << bitIndex));
    }

    m_BitIndex++;
    if(m_BitIndex >= m_BitsPerByte) {
      m_BitIndex = 0;
      m_ByteIndex++;
    }
    m_OverallBitIndex++;
    if(m_OverallBitIndex >= m_NumBits) {
      m_Done = true;
    }
}
```

Full source code for these 2 classes is in Appendix A.

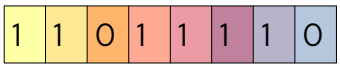
These classes are designed to be flexible enough to be able to read sequentially from a byte array while writing only to a certain amount of LSBs in another - And in fact, that is exactly how they are used. The tool provides an option for how many LSBs in the input image should be written to, and that value goes straight into the BitWriter's m_BitsPerByte variable.

The code for reading all the bytes from an input array data (implemented as a std::vector from the C++ standard library) and writing them to the 1st 2 LSBs (starting from the LSB) of each byte in input array cover is below:

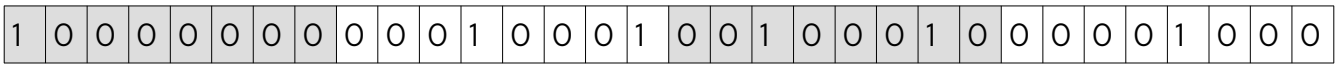
```
// Reading from data
```

```
BitReader reader(data.data(), data.size() * 8ul);

// Writing to cover
BitWriter writer(cover.data(), cover.size() * 8ul, 2, true);

// Use the BitWriter and BitReader to read from data and write to cover
while(!(reader.Done() || writer.Done())) {
    writer.WriteNextBit(reader.ReadNextBit());
}
```

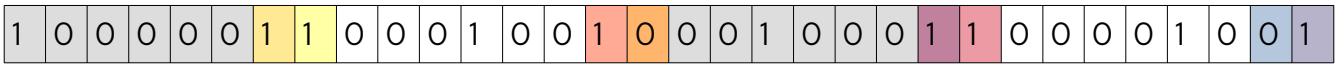
With the input arrays data:



and cover:



this results in cover becoming the following:



If the input cover is an array of bytes representing pixels of an image, then this can be used (and is used in Hideit) to write data to the LSBs of each pixel component - This requires that the pixel components all take up exactly a byte, which is enforced in Hideit by converting the image to a format where this is the case, if the image is not already in such a format. This functionality comes from the image reading library that Hideit uses, named stb_image after the author Sean T. Barrett, which when reading in images converts them to such 8-bit channel formats. stb_image and stb_image_write (the image writing library used) can both be found here: https://github.com/nothings/stb

2.3. AES-128 Implementation

The encryption part of Hideit implements AES-128 using the CTR (Counter) mode of operation.

The function used to encrypt/decrypt is called AES128, and as arguments it takes an array of bytes to be encrypted/decrypted (Using the CTR mode of operation, they are the same thing) and an array of bytes that is the key. The input array that is to be encrypted/decrypted will be referred to as the state array.

2.3.1. Cipher Key Generation

The first thing that is done is to modify the key using a Cryptographically Secure PseudoRandom Number Generator (CSPRNG). This is done to remove the cipher key used for encryption from the input key. The CSPRNG is seeded using the input key, and then random numbers are generated and written over the input key to generate a new key, based off of the input key. This is similar to hashing the input key in that it is now impossible to go from the cipher key to the input key, however it is less secure than a good hashing algorithm such as SHA-2, as there is not guaranteed to be a unique output for every input (or seed).

Hideit uses a fast, secure and unbiased CSPRNG called ISAAC for this (Jenkins, 1993-1996), which is implemented by wrapping the C code provided by Jenkins in a class called Rand. Code for this class can be found in Appendix B.

The code used to generate the cipher key from the input key using ISAAC is below:

```
void AESEncryption::ProcessKey128(std::vector<uint8_t>& key, Rand*& rand) {
    // Copy the key into a buffer to be used as the CSPRNG seed
    std::vector<uint32_t> randSeed;
    randSeed.resize(RANDSIZ);
    uint32_t seedBytes = RANDSIZ * sizeof(uint32_t);
    size_t copyAmt = key.size() > seedBytes ? seedBytes : key.size(); //
Make sure that what's copied doesn't go over the limit the size of the seed
(in bytes) - don't want any segfaults
    memcpy(&randSeed.front(), &key.front(), copyAmt);
    rand = new Rand(randSeed);

    key.resize(16);

    // Write random numbers into the key array
    for(uint8_t i = 0; i < key.size(); i += sizeof(uint32_t)) {
```

```
        uint32_t n = rand->NextInt();
        key[i] = n & 0xff;
        key[i + 1] = (n >> 8) & 0xff;
        key[i + 2] = (n >> 16) & 0xff;
        key[i + 3] = (n >> 24) & 0xff;
    }
}
```

After the cipher key has been generated, a key schedule must be generated. This is done as described in 2.2.2. Key Expansion, translated to C++ code, which can be found, along with related function SubWord, in Appendix C.

2.3.2. Operation

After the key schedule has been generated, a counter is initialised to a value generated by the CSPRNG. and the state array is lengthened to a multiple of 16 bytes, padding with 0 - It's initial length is saved into a variable to be restored later.

Here, if the length of the state array is greater than 0xffff, or 65535 bytes, then a multithreaded approach is taken to increase speed:

- The counter is copied into an array the same length as the state array.
- The counter array is iterated through, incrementing counter each block. For example, if counter started as 1, the next block would be 2, then 3, and so on.
- A number of threads are started (the number depends on the CPU). Easiest to explain with an example here, if 2 threads are started, then the first thread operates on the 1st block of the counter and state arrays, then the 3rd, then 5th, while the second thread operates on the 2nd, then the 4th, then the 6th.
    - Each thread calls a function to encrypt the current counter array block using AES.
    - Then, the current state array block is XORed with that encrypted counter block.

Otherwise, a single-threaded approach is taken, as for small amounts of data to encrypt the overhead of starting threads and creating an array to contain the counter array will mean single-threaded is faster.

- The state and counter arrays are iterated through:
    - A function is called to encrypt the current counter array block using AES.
    - The current state block is XORed with the encrypted counter block.
    - The counter is incremented.

Afterwards, the length of the state array is restored back to the saved initial length.

The actual AES cipher implementation is described in the following sections.

2.3.2. AES Cipher Implementation

The majority of x86/x86_64 CPUs nowadays support an instruction set extension called AES-NI, or Advanced Encryption Standard New Instructions, first proposed by Intel (Gueron, 2010) but also supported by AMD.

These new instructions enable hardware-accelerated AES that is faster and more secure against side-channel attacks (attacks based on the implementation of an encryption algorithm rather than the algorithm itself) than pure software implementations, along with being easier to use (with a little knowledge of assembly - Or use of intrinsic functions).

There are 6 added instructions: AESENC, AESENCLAST, AESDEC, AESDECLAST, AESKEYGENASSIST and ASEDIMC.

The two used in Hideit are AESENC and AESENCLAST. The rest, save AESKEYGENASSIST are relevant only for AES decryption, which because of the CTR mode of operation the implementation does not need to do. AESKEYGENASSIST will be used in later versions of Hideit.

AESENC <source>, <destination> performs one round of encryption on destination using source as the round key.

AESENCLAST <source>, <destination> performs the final round, as that differs slightly from the rest.

These functions operate using the 128-bit wide XMM registers.

Using GCC, it's possible to embed inline assembly in C++ code - this allows modification of C++ variables using assembly.

2.3.2.1. GCC Inline Assembly

GCC uses AT&T/UNIX syntax for assembly - This is quite different to Intel syntax, the most relevant being the source-destination operand ordering - Intel syntax uses "instruction destination source" while AT&T syntax uses "instruction source destination".

Other differences are:

- Register naming - In AT&T syntax register names are prefixed with an '%'. So rax becomes %rax. In inline assembly however registers need *two* '%', rax becoming %%rax
- Immediate operands - Or constants (including static C/C++ variables) are prefixed with an '$'. Hexadecimal constants are also prefixed with an 0x, so the hexadecimal number "fe" in AT&T syntax is $0xfe
- Operand size: In AT&T syntax the size of the memory operands is determined from the last character of the instruction name - Suffixes of 'b', 'w', 'l' and 'q' are used for 8-bit, 16-bit, 32-bit and 64-bit sizes.
- And others, but those are the major ones.

(Sandeep, 2003)

GCC uses the asm keyword to declare inline assembly code. There are 2 versions, basic and extended.

Basic is very basic. Simply:

```
asm("assembly code");
```

Extended is much more useful because it helps inform GCC of changes made to registers, and allows input and output of C/C++ variables. The basic form is here:

```
asm (
    "assembly code line 1;"
    "assembly code line 2;"
    : // List of outputs
    : // List of inputs
    : // List of clobbered registers
```

```
);
```

Note that it is possible to have multiple lines of assembly with basic as well - Each line in both basic and extended must end in either ';' or a newline and a tab, '\n\t'.

The list of outputs is a list of variables that can then be accessed in the assembly (in write-only mode). The list of inputs is the same, except they are read-only. If a read-write functionality is desired the same variable can be put in both lists.

Each input/output has a contraint. This determines where the variable will be stored. Examples of constraints are "m" for memory, and "r" for registers. Constraints in the output list are prefixed with an "=".

```
asm (
    "movl $5, %0;"
    "movq %3, %1;"
    : "=m"(ovar1), "=r"(ovar2)     // List of outputs
    : "r"(ivar1)                 // List of inputs
    : "%clobbered_register" // List of clobbered registers
);
```

The clobber list is the list of registers that are "clobbered" in the assembly. A clobbered register is one that has it's value modified. If the value of a register is changed in the assembly and GCC isn't informed of this by putting it in the clobber list, then this could lead to errors, as GCC will expect the value of that register to have not changed.

In the above example, %n refers to the nth operand - the inputs and outputs - counting the outputs first. So %0 refers to ovar1, %1 refers to ovar2, and %3 refers to ivar1.

2.3.2.2. Using the AES Instructions

The aes instructions operate on 128-bit values, using the XMM registers. AT&T syntax doesn't have an operand size specifier for 128-bit values, as only certain instructions support such sizes on a 32 or 64-bit computer.

MOVDQA <source>, <destination> is one such instruction - It moves (copies) source to destination.

XORPS <source>, <destination> is another, this does 128-bit XOR of source and destination, writing result to destination. This is of course used for the initial adding of the key to the block.

Now with the inline assembly and those instructions, the encryption of block b with round keys 0-10 (which are stored in a std::vector) can be done with the following code:

```
asm (
    // Load the block into XMM1
    "movdqa %0, %%xmm1;"
    // Move the first key into XMM0
    "movdqa %1, %%xmm0;"
    // Encrypt the block with each of the round keys
    "xorps %%xmm0, %%xmm1;"
    "aesenc %2, %%xmm1;"
    "aesenc %3, %%xmm1;"
    "aesenc %4, %%xmm1;"
    "aesenc %5, %%xmm1;"
    "aesenc %6, %%xmm1;"
    "aesenc %7, %%xmm1;"
    "aesenc %8, %%xmm1;"
    "aesenc %9, %%xmm1;"
    "aesenc %10, %%xmm1;"
    "aesenclast %11, %%xmm1;"
    // Finally move the result back into the block
    "movdqa %%xmm1, %0;"
    : "=m"(b)
    : "m"(roundKeys.at(0)), "m"(roundKeys.at(1)), "m"(roundKeys.at(2)),
"m"(roundKeys.at(3)), "m"(roundKeys.at(4)), "m"(roundKeys.at(5)),
      "m"(roundKeys.at(6)), "m"(roundKeys.at(7)), "m"(roundKeys.at(8)),
"m"(roundKeys.at(9)), "m"(roundKeys.at(10))
```

```
    : "%xmm0", // The register to store the first round key
      "%xmm1" // The register to hold the block
);
```

XORPS requires both it's operands to be in XMM registers, which is the reason for loading XMM0 with the first round key before doing XORPS.

3. Results

3.1. Hideit

3.1.1. About & Source Code

Hideit is a fully functional steganography and encryption tool, that is capable of encrypting any sort of data with a password and then embedding that encrypted data in the LSBs of the pixels of an image.

The full source code of Hideit is included in the submission of this report in a .zip file, along with the CMakeLists.txt files to be used with the meta-build-system CMake. It must be compiled with GCC (on Windows, MinGW can be used), but has not been tested on Windows, so may not compile there.

3.1.2. Functionality & Example Usage

Hideit is a simple command-line tool, it takes several arguments when it's run that determine what the tool does. Running with incorrect or no arguments shows the help, as shown below:

```
$ hideit
Usage: hideit <operation> [-h] [--version] [-df <datafile>] [-sf <stegofile>] [-cf <coverfile>] [-bpb <bpb>] [-c] [-p <passkey>]

Arguments:
        operation        - The operation to perform, must be either embed or extract
                embed   - Embed dataFile in coverFile, saving output as stegoFile
                extract - Extract data from stegoFile, saving output as dataFile


Options:
        -h, --help       - Displays this help
        --version        - Displays the program version
        -df, --datafile <datafile>       - Allows you to specify the datafile - If not specified, will take data from stdin (for embedding) and output to stdout (for extracting)
        -sf, --stegofile <stegofile>     - Allows you to specify the stegofile
        -cf, --coverfile <coverfile>     - Allows you to specify the coverfile
        -bpb, --bits-per-byte <bpb>      - Allows you to specify the number of bits of data stored in each byte (1-8) - More bpb is more noticeable
        -c, --compress   - If present, the data is compressed before being embedded
        -p, --passkey <passkey>  - If present, AES-128 encryption is used to encrypt and decrypt the data with the given passkey
```

Example usage, embedding "Ghost.gif" into "Sunset.png" with output "Spooky.png" using compression and AES-128 encryption with passkey "haunting":

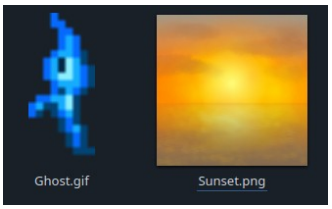


Figure 4: These images are both my own work

```
$ hideit embed -cf Sunset.png -df Ghost.gif -sf Spooky.png -c -p haunting
```

And the result:

On the left, the original Sunset.png. On the right, Spooky.png, with the compressed and encrypted Ghost.png hidden inside it - There is no visible difference.

And the extraction:

```
$ hideit extract -sf Spooky.png -df ExtractedPestilence.gif -p
haunting
```

Running diff between the files Ghost.gif and ExtractedPestilence.gif gives no output - The files are identical.

```
$ diff Ghost.gif ExtractedPestilence.gif
```

4. Discussion

4.1. Aims Met?

As can be seen from the results, Hideit is capable of LSB image steganography using compression with 100% accuracy, and is capable of encryption using AES-128 (effort has been made to ensure this is as strong as possible) and compression using LZO (Oberhumer, 2017) - Although the compression isn't as powerful as was aimed for, it has better performance than the algorithm that was initially going to be used, LZMA (Pavlov, no date) - Use cases will differ of course, so this isn't necessarily true, but speed is often less important than compression rate - With higher compression, more data can be embedded into a single cover, which is especially useful if there is a lot of data to be embedded. The deciding factor of which algorithm used was the ease of integration of the associated libraries though - LZO was found to be much easier to integrate, particularly using CMake.

Hideit was intended to be cross-platform, but hasn't achieved that as well as hoped. Currently, the tool works on x86_64 Linux, it should also work on 32-bit x86, and will probably work on MacOS - Windows is quite different though and it hasn't been tested on there.

All in all, the project has achieved most of what it set out to.

4.2. Malicious Uses of Steganography & Encryption

As with any tool, steganography and encryption can be used for some less than agreeable things.

For one, covert communication isn't by definition malicious, but certainly can be and most malicious communication won't exactly be out in the open. Nevertheless there are many legitimate uses for it.

However, one thing steganography can be used for is the disguising of malware by embedding it in something innocent, like an image of say a kitten. This is essentially a trojan. Embedding malware in this way would allow one to get it through a firewall, as according to the firewall the image is completely innocent - it may scan the image but it has no definitive way of knowing whether there is data hidden inside the image, unless it knows about LSB image steganography and tries to extract the data and analyse it - However even simple compression can work to thwart this method. It is possible it will try to extract the data using various compression algorithms however, which is where encryption comes in. A hacker may embed some encrypted malware into an image, to be somehow downloaded onto a target machine, extracted and executed. An example of image steganography being used in this way is LokiBot in 2019 - It infected systems by using VBS macros in microsoft excel to execute powershell to download malicious code hidden inside an image file, evading firewalls and antivirus (Ang, Mendoza, Yaneza, 2019).

4.2.1. Countermeasures

It is very difficult to block malware hidden in this way, as there is usually no definitive way of knowing whether something like an image does have hidden data - Even if the embedded malware isn't compressed or encrypted, it's very computationally intensive to scan and analyse every image - So most firewalls/antivirus won't do that.

While there may not usually be any definitive way of knowing if there is hidden data in a file, there is a whole field dedicated to development of techniques to try and determine - Steganalysis. Steganalysis is to steganography what cryptanalysis is to cryptography.

One method that could help determine whether data has been hidden in an image using LSB image steganography, is a type of visual analysis where the image is split into bit planes - A bit plane constists of a single bit for each pixel in the image (Or it could be done slightly differently, say a bit for each channel in each pixel). If there is any unusual display in the least significant bit planes, that could give a strong indication that there is data hidden in the LSBs.

This could be automated by use of statistical analysis, which uses statistics like chi-squared, standard deviation, and averages to compare parts of the file and measure deviances from the expected data.

Detecting data hidden in other ways like as alterations to the format of a file, such as appending data after the end of file marker in an image file, is structural analysis. Such sorts of steganography can be noticed by analysing the file manually in a hex editor such as HxD or Okteta, and several automated methods have also been developed.

(M4JPEG Project, no date)

One quite drastic countermeasure to LSB image steganography is the encoding of all images with a lossy compression algorithm, like JPEG. Lossy compression destroys/corrupts hidden data while still keeping images looking similar to how they did previously - So it is a viable method if images not being the exact same doesn't matter.

4.3. Future Work

Future work that would be carried out given more time and resources would include the implementation of AES-192 and AES-256 as well as AES-128 in Hideit - Also, implementations of those in pure C++, for when the inline assembly or the necessary instructions aren't supported. Also, ensuring that Hideit works on Windows and 32-bit x86, and the introduction of more powerful compression algorithms such as LZMA.

Aside from the continuation of the development of Hideit, other fascinating work that could be carried out is the research and implementation of some steganalysis techniques, such as the bit-plane statistical analysis mentioned previously.

References

- Ang, M.C., Mendoza, E., Yaneza, J. (2019) *LokiBot Gains New Persistence Mechanism, Steganography.* Available at: https://www.trendmicro.com/en_us/research/19/h/lokibot-gains-new-persistence-mechanism-uses-steganography-to-hide-its-tracks.html (Accessed: 13/05/2021)
- Bender, W., Gruhl, D., Morimoto, N., Lu, A. (1996) *Techniques for data hiding.* Available at: https://web.archive.org/web/20200611050549/https://pdfs.semanticscholar.org/8c82/c93dfc7d3672e58efd982a23791a8a419053.pdf (Accessed: 11/05/2021)
- Blazhevski, D., Bozhinovski, A., Stojchevska, B., Pachovski, V. (2013) *Modes of Operation of the AES Algorithm.* Available at: https://www.researchgate.net/publication/236656798_MODES_OF_OPERATION_OF_THE_AES_ALGORITHM (Accessed: 12/05/2021)
- Cloudflare (no date) *What is encryption?* Available at: https://www.cloudflare.com/en-gb/learning/ssl/what-is-encryption/ (Accessed: 11/05/2021)
- Daemen, J., Rijmen, V. (2020) *The Design of Rijndael - The Advanced Encryption Standard (Second Edition)* Available at: https://link.springer.com/book/10.1007/978-3-662-60769-5 (Accessed: 11/05/2021)
- Edney, J., Arbaugh, W. (2003) *Real 802.11 Security: Wi-Fi Protected Access and 802.11i.* Available at: https://etutorials.org/Networking/802.11+security.+wi-fi+protected+access+and+802.11i/ (Accessed: 18/04/2021)
- Federal Information Processing Standards (FIPS) (2001) *Announcing the Advanced Encryption Standard (AES)* Available at: https://csrc.nist.gov/csrc/media/publications/fips/197/final/documents/fips-197.pdf (Accessed: 09/05/2021)
- Gueron, S. (2010) *Intel® Advanced Encryption Standard (AES) New Instructions Set.* Available at: https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf (Accessed: 20/04/2021)
- Jenkins, R. (1993-1996) *ISAAC and RC4.* Available at: https://burtleburtle.net/bob/rand/isaac.html (Accessed: 16/04/2021)

- Johnson, N.F. and Jajodia, S. (1998) *Exploring steganography: Seeing the unseen.* Available at: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4655281 (Accessed: 10/05/2021)
- Krzyzanowski, P. (2020) *Steganography & Watermarking.* Available at: https://www.cs.rutgers.edu/~pxk/419/notes/steganography.html (Accessed: 11/05/2021)
- M4JPEG Project (no date) *Steganalysis: How to Detect Steganography.* https://digitnet.github.io/m4jpeg/about-steganography/how-to-detect-steganography.htm (Accessed: 13/05/2021)
- Oberhumer, M.F.X.J. (2017) *LZO real-time data compression library.* Available at: https://www.oberhumer.com/opensource/lzo/ (Accessed: 12/04/2021)
- Pavlov, I. (no date) *LZMA SDK.* Available at: https://www.7-zip.org/sdk.html (Accessed: 12/04/2021)
- Pirazzi, C. (no date) *Square and non-square pixels.* Available at: https://lurkertech.com/lg/pixelaspect/ (Accessed: 10/05/2021)
- Sandeep, S. (2003) *GCC-Inline-Assembly-HOWTO.* Available at: https://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html (Accessed: 20/04/2021)
- Singh, A., Singh, J., Singh, H. (2015) *Steganography in Images Using LSB Technique.* Available at: https://www.ijltet.org/wp-content/uploads/2015/02/60.pdf (Accessed: 11/05/2021)
- Stanger, J. (2020) *What Is Steganography?* Available at: https://www.comptia.org/blog/what-is-steganography (Accessed: 12/05/2021)
- Toumazis, A. (2009) *Steganography.* Available at: https://www.cl.cam.ac.uk/teaching/0910/R08/work/essay-at443-steganography.pdf (Accessed: 10/05/2021)

Appendices

Appendix A

BitIO.h is the header file containing the declarations of the classes BitReader and BitWriter along with their members. BitIO.cpp is the file containing the definitions of the members of BitReader and BitWriter.

BitIO.h

```
#ifndef BITIO_H
#define BITIO_H

#include <cstdint>

struct BitReader {
    const uint8_t* m_Bytes;
    uint64_t m_NumBits;
    uint8_t m_BitsPerByte;
    bool m_StartAtLSB;
    bool m_Done;

    BitReader(const uint8_t* bytes, uint64_t numBits, uint8_t
bitsPerByte = 8, bool startAtLSB = false);

    bool ReadNextBit();
    bool Done() { return m_Done; }

    uint32_t m_ByteIndex;
    uint8_t m_BitIndex;
    uint64_t m_OverallBitIndex;
};
```

```
struct BitWriter {
    uint8_t* m_Bytes;
    uint64_t m_NumBits;
    uint8_t m_BitsPerByte;
    bool m_StartAtLSB;
    bool m_Done;

    BitWriter(uint8_t* bytes, uint64_t numBits, uint8_t bitsPerByte = 8,
bool startAtLSB = false);

    void WriteNextBit(bool value);
    bool Done() { return m_Done; }

    uint32_t m_ByteIndex;
    uint8_t m_BitIndex;
    uint64_t m_OverallBitIndex;
};

#endif // BITIO_H
```

BitIO.cpp

```
#include "BitIO.h"

BitReader::BitReader(const uint8_t* bytes, uint64_t numBits, uint8_t
bitsPerByte, bool startAtLSB) : m_Bytes(bytes), m_NumBits(numBits),

m_BitsPerByte(bitsPerByte), m_StartAtLSB(startAtLSB),

m_Done(false), m_ByteIndex(0), m_BitIndex(0), m_OverallBitIndex(0) {
```

```
}

bool BitReader::ReadNextBit() {
    uint8_t byte = m_Bytes[m_ByteIndex];
    uint8_t bitIndex = m_StartAtLSB ? m_BitIndex : 7 - m_BitIndex;

    bool value = (byte >> bitIndex) & 0x1;

    m_BitIndex++;
    if(m_BitIndex >= m_BitsPerByte) {
      m_BitIndex = 0;
      m_ByteIndex++;
    }
    m_OverallBitIndex++;
    if(m_OverallBitIndex >= m_NumBits) {
      m_Done = true;
    }

    return value;
}

BitWriter::BitWriter(uint8_t* bytes, uint64_t numBits, uint8_t
bitsPerByte, bool startAtLSB) : m_Bytes(bytes), m_NumBits(numBits),

m_BitsPerByte(bitsPerByte), m_StartAtLSB(startAtLSB),

m_Done(false), m_ByteIndex(0), m_BitIndex(0), m_OverallBitIndex(0) {
}

void BitWriter::WriteNextBit(bool value) {
    uint8_t& byte = m_Bytes[m_ByteIndex];
```

```
    uint8_t bitIndex = m_StartAtLSB ? m_BitIndex : 7 - m_BitIndex;

    if(value) {
      byte = byte | (0x1 << bitIndex);
    } else {
      byte = byte & (~(0x1 << bitIndex));
    }

    m_BitIndex++;
    if(m_BitIndex >= m_BitsPerByte) {
      m_BitIndex = 0;
      m_ByteIndex++;
    }
    m_OverallBitIndex++;
    if(m_OverallBitIndex >= m_NumBits) {
      m_Done = true;
    }
}
```

Appendix B

This is the implementation of the C++ wrapper around the C code for ISAAC CSPRNG, split across the header Rand.h and the source Rand.cpp.

Rand.h

```
#ifndef RAND_H
#define RAND_H

#include <cstdint>
#include <vector>
```

```
// This has all been adapted from the C and C# implementations of ISAAC from https://www.burtleburtle.net/bob/rand

#define RANDSIZL   (8)
#define RANDSIZ    (1 << RANDSIZL)

struct RandContext
{
    uint32_t randcnt;            // Count through the results in randrsl[]
    uint32_t randrsl[RANDSIZ];    // Results - Randomness
    uint32_t randmem[RANDSIZ];    // Memory - The internal state
    uint32_t randa;                // Accumulator
    uint32_t randb;                // Last result
    uint32_t randc;                // Counter, guarantees cycle is at least 2^^40
};

/// Implementation of ISAAC (Indirection, Shift, Accumulate, Add, and Count) CSPRNG (Cryptographically Secure PseudoRandom Number Generator)
class Rand {
private:
    RandContext ctx;

    void Init(bool flag);
    void Isaac();

public:
    Rand();
    Rand(const std::vector<uint32_t>& seed);
    ~Rand();
```

```
    void SeedRand(const std::vector<uint32_t>& seed);
    uint32_t NextInt();
};

#endif // RAND_H
```

Rand.cpp

```
#include "Rand.h"
#include <cstring>

#define ind(mm,x)  (*(uint32_t *)((uint8_t *)(mm) + ((x) & ((RANDSIZ-
1)<<2))))
#define rngstep(mix,a,b,mm,m,m2,r,x) \
{ \
    x = *m;  \
    a = (a^(mix)) + *(m2++); \
    *(m++) = y = ind(mm,x) + a + b; \
    *(r++) = b = ind(mm,y>>RANDSIZL) + x; \
}

#define mix(a,b,c,d,e,f,g,h) \
{ \
    a^=b<<11; d+=a; b+=c; \
    b^=c>>2;  e+=b; c+=d; \
    c^=d<<8;  f+=c; d+=e; \
    d^=e>>16; g+=d; e+=f; \
    e^=f<<10; h+=e; f+=g; \
    f^=g>>4;  a+=f; g+=h; \
    g^=h<<8;  b+=g; h+=a; \
```

```
    h^=a>>9;  c+=h; a+=b; \
}

Rand::Rand() {
    ctx = RandContext();
    Init(false);
}

Rand::Rand(const std::vector<uint32_t>& seed) {
    ctx = RandContext();
    SeedRand(seed);
}

void Rand::SeedRand(const std::vector<uint32_t>& seed) {
    uint32_t amtData = seed.size() > 256u ? 256u : seed.size();
    memcpy(ctx.randrsl, &seed.front(), amtData);
    Init(true);
}

uint32_t Rand::NextInt() {
    if(ctx.randcnt-- == 0) {
      Isaac();
      ctx.randcnt = RANDSIZ - 1;
    }
    return ctx.randrsl[ctx.randcnt];
}

Rand::~Rand() {
}

void Rand::Init(bool flag) {
```

```
  int i;
  uint32_t a,b,c,d,e,f,g,h;
  uint32_t *m,*r;
  ctx.randa = ctx.randb = ctx.randc = 0;
  m=ctx.randmem;
  r=ctx.randrsl;
  a=b=c=d=e=f=g=h=0x9e3779b9;  /* the golden ratio */

  for (i=0; i<4; ++i)          /* scramble it */
  {
    mix(a,b,c,d,e,f,g,h);
  }

  if (flag)
  {
    /* initialize using the contents of r[] as the seed */
    for (i=0; i<RANDSIZ; i+=8)
    {
        a+=r[i  ]; b+=r[i+1]; c+=r[i+2]; d+=r[i+3];
        e+=r[i+4]; f+=r[i+5]; g+=r[i+6]; h+=r[i+7];
        mix(a,b,c,d,e,f,g,h);
        m[i  ]=a; m[i+1]=b; m[i+2]=c; m[i+3]=d;
        m[i+4]=e; m[i+5]=f; m[i+6]=g; m[i+7]=h;
    }
    /* do a second pass to make all of the seed affect all of m */
    for (i=0; i<RANDSIZ; i+=8)
    {
        a+=m[i  ]; b+=m[i+1]; c+=m[i+2]; d+=m[i+3];
        e+=m[i+4]; f+=m[i+5]; g+=m[i+6]; h+=m[i+7];
        mix(a,b,c,d,e,f,g,h);
```

```
            m[i  ]=a; m[i+1]=b; m[i+2]=c; m[i+3]=d;
            m[i+4]=e; m[i+5]=f; m[i+6]=g; m[i+7]=h;
        }
    }
    else
    {
        /* fill in m[] with messy stuff */
        for (i=0; i<RANDSIZ; i+=8)
        {
            mix(a,b,c,d,e,f,g,h);
            m[i  ]=a; m[i+1]=b; m[i+2]=c; m[i+3]=d;
            m[i+4]=e; m[i+5]=f; m[i+6]=g; m[i+7]=h;
        }
    }

    Isaac();            /* fill in the first set of results */
    ctx.randcnt=RANDSIZ;  /* prepare to use the first set of results */
}

void Rand::Isaac() {
    uint32_t a,b,x,y,*m,*mm,*m2,*r,*mend;
    mm=ctx.randmem; r=ctx.randrsl;
    a = ctx.randa; b = ctx.randb + (++ctx.randc);
    for (m = mm, mend = m2 = m+(RANDSIZ/2); m<mend; )
    {
        rngstep( a<<13, a, b, mm, m, m2, r, x);
        rngstep( a>>6 , a, b, mm, m, m2, r, x);
        rngstep( a<<2 , a, b, mm, m, m2, r, x);
        rngstep( a>>16, a, b, mm, m, m2, r, x);
    }
```

```
    for (m2 = mm; m2<mend; )
    {
      rngstep( a<<13, a, b, mm, m, m2, r, x);
      rngstep( a>>6 , a, b, mm, m, m2, r, x);
      rngstep( a<<2 , a, b, mm, m, m2, r, x);
      rngstep( a>>16, a, b, mm, m, m2, r, x);
    }
    ctx.randb = b; ctx.randa = a;
}
```

Appendix C

These functions are static functions inside a class called AESEncryption. sbox[] is an array containing all the values in the table Figure 3 in section 2.2.3.1. SubBytes(). Looking up value a with $\text{sbox}(a_0, a_1)$ (where a_n is the nth nibble in a starting from the most significant) in the table is the same as using a as the index into sbox[]. For example, sbox[0x5f] is identical to $\text{sbox}(0\text{x}5, 0\text{xf})$.

```
std::vector<RoundKey> AESEncryption::GenRoundKeys128(const
std::vector<uint8_t>& key) {
    /// Round constants. Rcon[0] is unused
    static constexpr uint8_t c_Rcon128[11] = { 0, 1, 2, 4, 8, 16, 32,
64, 128, 27, 54 };

    std::vector<RoundKey> roundKeys;
    roundKeys.resize(11);

    // Copy bytes into the first round key - This is the value of it
    memcpy((uint8_t*)&roundKeys.front(), &key.front(), key.size());

    // Calculate all the round keys
    // Each round key contains words w0, w1, w2, w3
```

```
    // Each of the round keys from 1 to 10 (0 is initialised to the value of the key):
    //   w0 is the sum of: w0 from previous round key, s-box substitution of w3 from previous round key rotated right by 8, and a value from a special table called Rcon
    //   w1, w2, w3 are the sum of the corresponding word in the previous round key and the preceding word in the current round key
    // Note that this is using Finite Field Arithmetic, so addition is actually XOR (^)
    // This method: https://csrc.nist.gov/csrc/media/publications/fips/197/final/documents/fips-197.pdf
    // Finite Field Arithmetic: https://etutorials.org/Networking/802.11+security.+wi-fi+protected+access+and+802.11i/Appendixes/Appendix+A.+Overview+of+the+AES+Block+Cipher/Finite+Field+Arithmetic/
    for(uint8_t i = 1; i < roundKeys.size(); i++) {
      roundKeys.at(i).w0 = roundKeys.at(i - 1).w0 ^ SubWord(std::rotr(roundKeys.at(i - 1).w3, 8)) ^ c_Rcon128[i];
      roundKeys.at(i).w1 = roundKeys.at(i - 1).w1 ^ roundKeys.at(i).w0;
      roundKeys.at(i).w2 = roundKeys.at(i - 1).w2 ^ roundKeys.at(i).w1;
      roundKeys.at(i).w3 = roundKeys.at(i - 1).w3 ^ roundKeys.at(i).w2;
    }

    return roundKeys;
}
```

```
uint32_t AESEncryption::SubWord(uint32_t word) {
    uint8_t b0 = sbox[(word >> 24) & 0xff];
    uint8_t b1 = sbox[(word >> 16) & 0xff];
```

```
    uint8_t b2 = sbox[(word >> 8) & 0xff];
    uint8_t b3 = sbox[word & 0xff];
    return (b0 << 24) | (b1 << 16) | (b2 << 8) | b3;
}
```