Hector Codes - SoNIC Steganography

Hector A. Tosado Jimenez

August 7, 2011

1 Introduction

asdIn our world we tend to rely too much on a variety of networks in order to have a functional society. May it be railroads, highways, financial networks, telecomm networks, businesses, school systems or the internet; we just simply depend too much on their functionality.

Different networks interact with and build on each other. Social networks are built upon information networks which are built upon communication networks which in turn are built on physical networks. It's because of all these types of networks that there is a need to optimize and look for other uses that can be achieved with them.

Steganography is the art and science of writing hidden messages in such a way that no one, apart from the sender and intended recipient, suspects the existence of the message, a form of security through obscurity[1]. When compared to normal cryptography, where a encrypted message would arouse suspicion in some way, steganography depends on complete secrecy on the message being sent. Most of contemporary, widely available implementations of steganographic systems are dedicated to the multimedia applications hidden data is distributed in sound files, images and movies. More specifically, and for our purposes there is a type of steganography referred to as Network Steganography, where instead of using digitalize media, like the ones mentioned above, it uses communication protocols' control elements and their basic intrinsic functions. It is further described as a type of steganography where "the concealment occurs within data whose inherent ephemerality makes the hidden payload nearly impossible to detect, let alone thwart". An example is Voice Over IP(VoIP) where data is transferred through an going internet conversation. The longer the communicators talk, the longer the secret message they can send [2].

The design of this project is to enable the sending of a covert message between 2 communication devices utilizing steganographic techniques. These will allow a message to be sent without a higher level application or a router knowing. This type of network steganography will be achieved using Hector Codes, which the author implemented during a summer internship program, and SoNIC.

2 Background

2.1 Network Stack

In order to be able to send a covert message a good understanding of how computers communicate over a network is needed. The Open Systems Interconnection (OSI) model, is the sub-division of a communication network into 7 layers as seen in Figure 1. These layers provides services to the layers above and requests services from the layers below. The physical layer (PHY) is the lowest, or first layer of the OSI model. This layer generates and detects signal so as to transmit and receive data over a network medium. The second layer, the Data Link Layer, provides reliable, efficient communication between end hosts connected by a single communication channel. This layer has a sublevel called Media Access Control (MAC) that determines how data on a network meant for a specific computer reaches it and how a computer can transmit data. The third layer is the Network layer that provides the means of communicating between hosts that are in different nodes. The fourth layer is the Transport layer that is in charge of detecting loss of packets, bit duplication and the transparent transfer of data between end systems. The fifth, sixth and seventh layer are in charge of terminating existing data connections, interpreting data to be used in the application layer and representing the data to the end-user, respectively.

The PHY layer is in charge of how a message is generated and sent but it consists of 3 more sub layers that work more specifically with the managing of the data as illustrated in Figure 1. The Physical Coding Sublayer (PCS), PMA (Physical Medium Attachment), and PMD (Physical Medium Dependent), the PCS is in charge of encoding every 64-bit block into a 66-bit block when transmitting, by adding a two bit sync header and scrambling the 64-bit data. After that is done a 66-bit block is transmitted over a physical medium and received again using the PMA to recover the bits from electrical signals and having the PCS decode 66-bit blocks. This type of encoding is done after a packet is received from the IP or Network layer and sent across the MAC layer into the PHY layer. The MAC layer is then in charge of adding Ethernet header and computing a 32 bit CRC (Cyclic Redundancy Check) value over the entire received Ethernet Frame. A minimum of 12 idle character are sent between frames by the MAC which are needed for correct analysis of the bitstreams. There is a Reconciliation Layer between the MAC and the PHY layers that continuosly generate idle packets if there is no data character to send The PCS is also in charge of then retrieving the 66-bit block sent, descramble and decode the data that was encoded and scrambled. After this is done the data is then sent to the top layers that may need it.

2.2 SoNIC

One of the biggest limitations of the PHY layer is that its implementation is normally done in hardware which limits the amount of analysis that could be

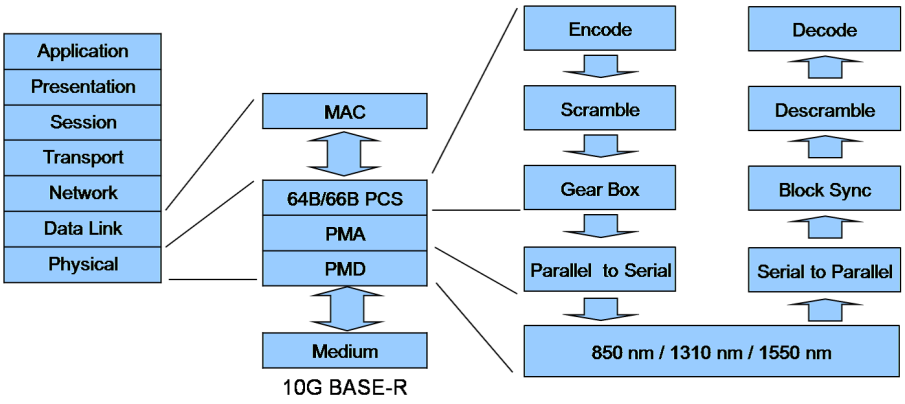

Figure 1: 10G Network Stack

done to the 10G data bitstream. This is where the Software Network Interface Card (SoNIC) comes in, SoNIC is an effort to build a real-time network adaptor that gives a user precise control over the entire 10G network stack [3]. It divides what is sent in software and how it's sent in hardware, more specifically allows the PCS and all the top layers to be implemented in software and the rest of the PHY layer in hardware. This manipulation in software allows the user to visualize and study the PHY layer more closely because of being able to alter the information that is being stored inside the data packets. It provides the developer with a simple yet powerful programming interface to the PHY and MAC layers.

2.3 10G Ethernet Standard

In order to be able to use SoNIC and the Hector Codes the 10G Ethernet Standard plays a big part on understanding where to actually insert the hidden message character bits in the bitstream. When sending data across an ethernet cable, the packets of information that are sent need to follow a specific format. Whenever an ethernet frame is received,the PCS layer encodes every 64 bit data of the Ethernet frame into a 66 bit block. Each block has to be correctly configured with the correct block Type to be able to identify it as the beginning or end of a frame or as a data only block. The data block consists of the '01' sync header and 64-bit scrambled data. The mixed data/control block has the '10' sync header, 8-bit Type field and 56-bit payload, example of these Types are the /S/ (Start of an Ethernet frame), /T/ (End of an Ethernet frame) and /E/ (Idle frame) for simplicity. Similarly the standard defines many control characters, although we only need /I/ (=0x00) for this design which is used as an idle character between packets. A control character is 7-bit long, and a data character is 8-bit long [4].

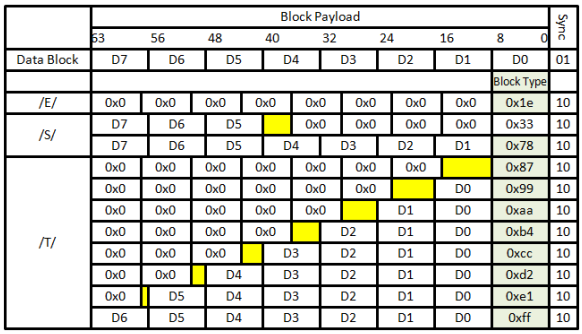

Figure 2: 10G Ethernet Standard - Block Format

In 10G network when two end hosts are connected with an ethernet cable, both ends will continously generate a continous bitstream regardless of the existence of packets. This means, that both of them will insert idle characters into the bitstream until a packet containing data arrives. The IEEE 802.3 standard states that a valid ethernet bitstream will have twelve idle characters between characters.

/S/ indicates the start of a new Ethernet frame in the 66b block. /S/ appears at the first octet(8-bit) and has two values depending on the position of the first octet of an Ethernet frame. Similarly, /T/ indicates the end of an Ethernet frame and has eight different values depending on the number of valid data octets in the block. For invalid octets you can simply fill them with /I/s. /E/ is filled with eight idle characters and is used for interpacket gap between two Ethernet frames, as well as to comply with the 12 minimum idles characters needed. These block Types and structure is shown in Figure 2.

3 Design

Before heading into more details a brief overview can be seen in Figure 5. This figure shows how two end host computers or nodes are connected together through an ethernet cable. The two computers have SoNIC installed and would allow User Y to send a hidden message to User Z without any of the end hosts knowing. This message would be send and interpreted using SoNIC.

Therefore, following the PCS manipulation that is allowed by the SoNIC, work can be done to the encode and scramble as well as the decode and descramble processes in order to insert a hidden message. The hidden message will be formatted using the Hector's Code in Table 1; each letter will match with the corresponding ASCII character in order to have its correct 6-bit representation, this is done by ignoring the 7th bit in the ASCII character. The way our hidden message is going to be encoded is by utilizing the thin rectangles spaces that appear in the /S/ 0x33 block type and in the /T/ block types in Figure 4. These thin rectangles are expressed to be 0's in Clause 49.2.4.3 of the

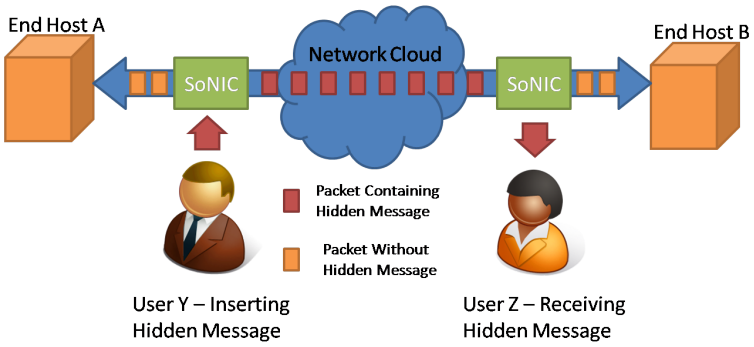

Figure 3: Visual of Hector Codes

IEEE STD 802.3-2008. The message is going to be introduced into the encoder before it gets scrambled. In the same way, it will get descrambled before it is decoded so the message would not be lost, this can be seen in the representation in Figure 3. We assume that we have constant communication between two nodes, in other words the two end nodes are constantly sending data packets in a continuous interval in order to be able to send a full hidden message through the empty spaces (thin rectangles in Figure 4).

3.1 Code for PCS

This part describes the variables and the changes that are going to be done to encode and decode.

1. Variables for encode():

   - *uint64_t h_bits:* has the 6 bits of the Hector Code character that is currently being embedded in *encode*().
   - *uint64_t th_bits:* temporary h_bits variable that contains the bits that are currently being embeded into the encode process in the available rectangle spaces.
   - *int h_counter:* counter for how many bits have already been sent of a particular character. $0 \leq h\_counter \leq 6$.
   - *int h_length:* maintains the amount of bits that have been sent from h_code.
   - *char h_input[4096]:* char array that stores hidden message thats input by the user.
   - *unsigned char* h_code:* complete data bits of encrypted message with start and end of message bits included.
   - *static int h_total_bits:* size of the hidden message that is going to be sent.

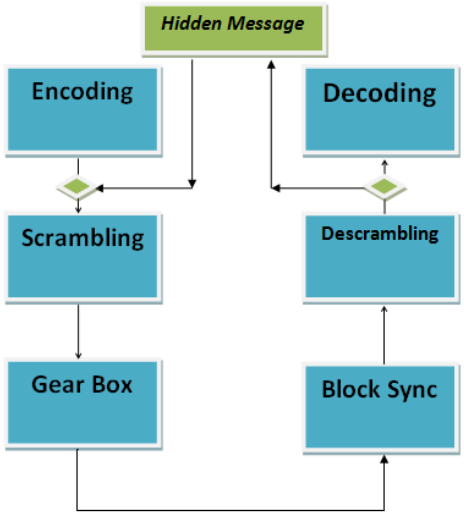

Figure 4: /S/ and /T/ block format with rectangle spaces

$C_0\ C_1\ C_2\ C_3/S_4\ D_5\ D_6\ D_7$	10	0x33	C_0	C_1	C_2	C_3		D_5	D_6	D_7
$T_0\ C_1\ C_2\ C_3/C_4\ C_5\ C_6\ C_7$	10	0x87		C_1	C_2	C_3	C_4	C_5	C_6	C_7
$D_0\ T_1\ C_2\ C_3/C_4\ C_5\ C_6\ C_7$	10	0x99	D_0		C_2	C_3	C_4	C_5	C_6	C_7
$D_0\ D_1\ T_2\ C_3/C_4\ C_5\ C_6\ C_7$	10	0xaa	D_0	D_1		C_3	C_4	C_5	C_6	C_7
$D_0\ D_1\ D_2\ T_3/C_4\ C_5\ C_6\ C_7$	10	0xb4	D_0	D_1	D_2		C_4	C_5	C_6	C_7
$D_0\ D_1\ D_2\ D_3/T_4\ C_5\ C_6\ C_7$	10	0xcc	D_0	D_1	D_2	D_3		C_5	C_6	C_7
$D_0\ D_1\ D_2\ D_3/D_4\ T_5\ C_6\ C_7$	10	0xd2	D_0	D_1	D_2	D_3	D_4		C_6	C_7
$D_0\ D_1\ D_2\ D_3/D_4\ D_5\ T_6\ C_7$	10	0xe1	D_0	D_1	D_2	D_3	D_4	D_5		C_7
$D_0\ D_1\ D_2\ D_3/D_4\ D_5\ D_6\ T_7$	10	0xff	D_0	D_1	D_2	D_3	D_4	D_5	D_6	

Figure 5: /S/ and /T/ block format with rectangle spaces

2. Variables for decode():

   - *unsigned char h_hector:* hidden message is stored here as is it being extracted from the decoding. This message is still 1's and 0's.
   - *static int h_message:* 1 or 0, true or false. Flags if a message is being received in *decode*().
   - *char h_buffer[4096], *h_pointer:* maintain the message being received in *decode*(). h_buffer is filled up with all the bits being retrieved from *decode*(), saving in each byte size 6 bits that correspond to a specific hector code. h_pointer always keeps pointing to the current 6 bits being introduced to h_buffer, it's used to check for the END_HiddenMessage character.
   - *int h_count:* maintains the counts for the *decode*() on the amount of bits of a character that has been already received.
   - *int h_length:* in *decode*() works as a counter for everytime that h_length%6 == 0, checks if the bit sequence received is the END_HiddenMessage.
   - *uint64_t h_bits:* handles the descrambled output and adjusts the bitstream as to retrieve the bits that are hidden.
   - *uint64_t h_code:* maintains the bits that are currently being received, will be added to h_buffer after 6 bits of a character is completed.

3. encode(...) One thing to note about this design is that the original code from SoNIC is not touched. The implementation relies on code built in on already existing code that will, give an input message from user, start encoding the hidden message into the packets. When an input message is received the corresponding variables are set and initialized in order to continue with inserting the hidden message into the stream.

```
     printf("Message? : ");

fgets(h_input, 4096, stdin);

        if(strlen(h_input) != 0)
        {

h_code = h_encode(h_input);
h_total_bits = strlen((char *) h_code) * 6;
h_counter = 0;
h_bits = *(uint64_t*)h_code;//Casts the first 8 bits on the h_code to h_bits.
h_bits <<= 58;//Clears the 7 and 8th bits.
h_bits >>= 58;
h_length = 0;
```

```
        }
```

After this is done, and if a message is received, bit manipulation is done to th_bits allowing the correct amount of bits to be inserted into the amount of rectangular spaces that are available. The bit manipulation done depends on how many bits are left of a character of the hidden message to be sent and how many available spaces there are in the to embed the hector code bits in the block type. For example, the following code adds the character bits into the 0x33 block type. This part uses conditional statements on the h_counter to know how the 4 rectangular spaces that are available in the block type are going to be fitted. If $h\_counter < 2$ 4 bits of a hidden message characters are embeded into the payload, or if the $h\_counter \geq 2$, then your going to embed the left over bits of one character and then look for the following character in h_code.

```
     if(h_counter < 2 && h_counter >= 0 && h_length < h_total_bits)
{
temph_bits = h_bits;
temph_bits <<= 56 + 4 - h_counter;
temph_bits >>= 56 + 4 - h_counter + h_counter;
cur <<= 4;
cur = cur|temph_bits;
‘h_counter += 4;
h_length += 4;
}

else
  if(h_counter < 6 && h_counter >= 2 && h_length < h_total_bits)
{
temph_bits = h_bits;
temph_bits >>= h_counter;
cur <<= 4;
cur = cur | temph_bits;
h_length += 6 - h_counter;

if(h_length < h_total_bits)
{
h_code += 1;
h_bits = *(uint64_t*)h_code;
h_bits <<= 58;
h_bits >>= 58;
temph_bits = h_bits;
temph_bits <<= 56 + (6 - h_counter) + 4;
temph_bits >>= 56 + (6 - h_counter) + 4 - (6 - h_counter);
```

```
cur = cur | temph_bits;
h_counter = h_counter - 2;
h_length += h_counter;
}
}
}
if(h_message == 1)
cur <<= 36;
else
cur <<=40;
```

4. decode(...)

   In this section the descrambled variable is received as it normally would. However, in order to be able to check whether a hidden message is included in the blocks payload, a test is done to the a block types corresponding rectangular spaces. A checkis done to find out whether the spaces contain the start of the START_HiddenMessage character. This would flag *h_message* as 1 and start the retrieving of the hidden message across the whole decoding process.

```
if(h_message == 0 && h_complete == 0)
{

h_bits = descrambled << 24;
            h_bits = h_bits >> 60;

if(h_bits == 0xe)
{
h_message = 1;
h_count = 4;
h_code = h_bits;
h_length = 4;
}
...
```

   To continue with the past example, on how the encoding of a hidden message worked in the 0x33 block type, lets see how its done in the decode. After *h_message* is changed to 1, conditional statements are used to verify the amount of h_count (It is worth mentioning that h_count, in contrast to h_counter, maintains the amount of bits that have been received of a specific hidden message character). If $h\_count < 2$, then descrambled is shifted to point to the 4 rectangular spaces of the 0x33 block type and

added into h_code. The variable h_code is in charge of receiving the hidden message character bits until 6 are received. If $h\_count \geq 2$ then the bits remaining of the character that is currently being received are added to h_code, h_code is then added to a char array that will contain all the retrieved characters that are retrieved from the decode process and the remaining bits that werent used to finish the previous character are added to a new h_code. In this part a test is done in order to check if a END_HiddenMessage character was the last character received from the decoding process. If true, h_message is set to 0 and h_complete set to 1 finishing all hidden message retrieving algorithms, allowing for the rest of the decode to run as normal. The function *h_decode* is then called printing out the hidden message.

```
if(h_message == 1)
{
if(h_count < 2)
{
h_bits = descrambled << 24;
h_bits >>= 60 - h_count;
h_code = h_bits|h_code;
h_ length += 4;
h_count += 4;
}
else
{
if( h_count >= 2)
{
h_bits = descrambled << (24 + (h_count - 2));
h_bits = h_bits >> (60 + (h_count -2) - h_count);
h_code = h_bits | h_code;
h_length += (6 - h_count);
* h_pointer ++= h_code;

if(h_length % 6 == 0)
{
if(h_code == 0x3f)
{
h_message = 0;
h_length = 0;
h_complete = 1;
printf("Detected message is : %s\n", h_decode(h_buffer));
}

}
if(h_message == 1)
{
```

```
h_bits = descrambled << 24;
h_bits = h_bits >> (60 + (6 - h_count));
h_code = h_bits;
if(h_count == 2)
h_code = 0x0;
h_count = (h_count - 2);
h_length += h_count;

}
}

}
}
```

3.2 Implemented code for Hector Codes

1. h_encode(): Function called after receiving hidden message input. Interprets the string character by character adding into another string the Hector Code representation. This string is returned.

```
unsigned char* h_encode(char *input)
{
char * h = malloc(4096);
char c;

strcat(h, "~");

while ((c = *input++) != '\0') {
                switch (c)
                {

   case 'A':
                        strcat(h, "@");
                        break;
                ...
                ...

                case '^':
                        strcat(h, "{");
                        break;

case '!':
strcat(h, "|");
break;
}
```

```
}
strcat(h, "?");//End of message bits
printf("The current hector code is: %s\n", h);
return (unsigned char*)h;
}
```

2. h_decode - Function that is called after hidden message is completely received. Characters are interpreted as hexadecimal numbers and changed to their corresponding Hector Code letter or number. These character are added to another string and then returned.

```
char* h_decode(char hector[])//char * hector[4096]???
{
char* h = malloc(4096);
int i = 0;
uint64_t h_char = (uint64_t)hector[i];

while(h_char != 0x3f)
{
  switch(h_char)
{

case 0x00:
strcat(h, "A");
break;

case 0x3c:
      strcat(h, "!");
      break;

    case 0x3d:
      strcat(h, "");
      break;

}
i++;
h_char = (uint64_t)hector[i];
}
return h;
}
```

References

[1] Patrick Philippe Meier, "Steganography 2.0: Digital Resistance against Repressive Regimes", 2009 http://irevolution.net/2009/06/05/steganography-2-0-digital-resistance-against-repressive-regimes.

[2] Lubacz,Jozeph; Mazurczyk,Wojciech; Szczypiorski, Krysztof "Vice Over IP: The VoIP Steganography Threat" 2010 http://spectrum.ieee.org/telecom/internet/vice-over-ip-the-voip-steganography-threat/

[3] Ki Suh, Lee; Wang, Han; Weatherspoon, Hakim. "Real-time High Precision Network Analysis with SoNIC" Computer Science Department, Cornell University. 2010.

[4] IEEE 802.3-2008-Section 4, "Physical Coding Sublayer (PCS)", p 262

A 6 Bits

During the design, the amount of bits that were going to be used to encrypt each letter or symbol had to be chosen. Different amount of bits were tried, at first 4 bits, then 5 and ending up using 6 bit because it was the most efficient while still not having to reach the 7 bits that ASCII code uses. When initially starting the project one of the most important parts of sending a hidden or covert message would be trying to keep the number of bits used as low as possible. 4 bits were tested at the beginning to try and keep the message being sent to only letters using a very similar representation to that of the Morse Code were the "*dots*" and "*dashes*" were 0 and 1, respectively. For letters that only had a dot and dash combination that didn't complete 4 bits, 0's were added to fill the spots. However, this only allowed for 16 unique characters having more than 6 letters that overlapped with each other, as would be expected. 5 bits had a similar problem because, although it could fit all the letters, it wouldn't be able to fit numbers or symbol characters. Being able to use numbers and/or symbol character in a hidden message would make it much more practical. This was then the reason 6 bits were used, these bits allowed for numbers, the most common used symbols and all the capitol letters to be represented in the 6 bits. Some symbols that are not popularly used were removed and the minuscule letters are not really needed to be able to understand a covert message. The last four 6 bit representations were reserved for special cases and to be able to flag the start and the end of a hidden message. These flags will notify the program when to read the corresponding "empty" spaces for hidden message bits and when to just continue running the algorithm normally.

Table 1: Complete Hector Codes

Binary Key	Character	Binary Key	Character#1
000000	A	100000	6
000001	B	100001	7
000010	C	100010	8
000011	D	100011	9
000100	E	100100	SPACE
000101	F	100101	?
000110	G	100110	_
000111	H	100111	#
001000	I	101000	$
001001	J	101001	%
001010	K	101010	’
001011	L	101011	(
001100	M	101100	)
001101	N	101101	*
001110	O	101110	+
001111	P	101111	,
010000	Q	110000	-
010001	R	110001	.
010010	S	110010	/
010011	T	110011	:
010100	U	110100	;
010101	V	110101	<
010110	W	110110	=
010111	X	110111	>
011000	Y	111000	@
011001	Z	111001	|
011010	0	111010	
011011	1	111011	^
011100	2	111100	!
011101	3	111101	RESERVED
011110	4	111110	START_HiddenMessage
011111	5	111111	END_HiddenMessage