Secrect Sharing Progress

Joan Ngure

Secret Sharing Made Short

1 How do you detect cheating when the file is in binary format?

Basic IDA schemes do not deal with malicious parties or with secrecy of information. Our scheme [1] also assumes that all parties are honest. If the secret reconstructed is a string that follows a specific format for example an English text, then cheating can be detected, in case the format is altered or does not make sense. What of a binary strings, how is cheating detected ?

1.1 Error detection codes and correction codes

1.1.1 Hamming codes

We have a message say x bits and parity bits say y. The total message sent = x+y. y is chosen in the following way:
y$\geq \lfloor \log(x+y) \rfloor$.
If x = 4 then y$\geq$3

Example:
Let's say we want to send the message 1101 with 3 parity bits p_1, p_2, p_3. The parity bits are added in positions which are powers of 2 like 1,2,4 etc.

7	6	5	4	3	2	1
1	1	0		1		
			p_3		p_2	p_1

How do we calculate the parity bits?

	p_3	p_2	p_1
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

For each we will pick all the numbers where there is a 1.
$p_1 = \{1, 3, 5, 7\}$
$p_2 = \{2, 3, 6, 7\}$
$p_3 = \{4, 5, 6, 7\}$
For instance p_1 controls positions $\{3, 5, 7\}$. Which contain 101, hence p_1 will be 0 since we need an even number of 1's. Using the same procedure for p_2 and p_3 we get:

7	6	5	4	3	2	1
1	1	0	0	1	1	0
			p_3		p_2	p_1

The code sent is 1100110.
Imagine an error occurs and the receiver gets say 1110110. How do we correct it? We check the correctness of the parity bits.

7	6	5	4	3	2	1
1	1	1	0	1	1	0
			p_3		p_2	p_1

$p_1 = \{3, 5, 7\}$= 111= 1 = error
$p_2 = \{3, 6, 7\}$ = 111= 1 = correct
$p_3 = \{5, 6, 7\}$ = 111= 1 = error
An error occurred in p_1 and p_3. Tracing in back, the only position common to p_1 and p_3 not in p_2 is 5. Therefore an error must have occurred in position 5. Additionally we can also find the error position by adding the positions of p_1 and p_3 which is 4+1 = 5.

1.1.2 Forward Correcting Codes

Depends on the hamming distance measure. Hamming distance checks the distance between two bit strings of the same length. Consider the following two bit strings:

1	0	1	1	0	1
1	1	0	1	0	0
	x	x			x

We can see that the hamming distance is 3.
The sender and the receiver have a dictionary of codewords. Usually the distance between any two codewords is at least 3. Consider the following example.

bit	codeword
00	00000
01	11100
10	10011
11	01111

The sender wants to send the following message:

01	00	01	11	11	10	00	10

Corresponding to :

11100	00000	11100	01111	01111	10011	00000	10011

The receiver receives:

11100	$10000_{(1)}$	$11110_{(2)}$	01111	$01010_{(3)}$	10011	$10011_{(4)}$	10011

Let's focus on the bits in blue since that's where errors occurred. Comparing the received codewords with the valid codewords and calculating the hamming distance for each:

valid codeword	(1)	(2)	(3)	(4)
00000	1	4	2	3
11100	2	1	3	4
10011	2	3	3	0
01111	5	2	2	3

We can see that (4) is actually a valid codeword but it's incorrect since we know what was sent. This implies that three errors occurred which is highly unlikely. For (1) and (2) we can correctly correct them. We replace them with the codewords with the least hamming distance between each of them and the codewords. Only one error has occurred for each. For (3) there are two codewords with the same hamming distance between them and (3), thus we will request the sender to resend that part of the code. The receiver therefore decodes as follows:

01	00	resend	11	11	10	10	10

1.1.3 Checksum

This is an error detection technique. The data is divided into segments of the same bits. The segments are added using 1's compliments (1+1 = 0 carry 1) and the sum is complimented to get the checksum. The checksum is appended at the end of the data as a segment. The receiver then adds all the segments including the appended checksum at the sender's end using 1's compliment and the sum is complimented. If the result is zero accept otherwise reject.
Example:
Sender:

1	2	3	4
10011001	11100010	00100100	10000100

1+2		1	0	0	1	1	0	0	1
		1	1	1	0	0	0	1	0
	1	0	1	1	1	1	0	1	1
									1
(1+2)+3		0	1	1	1	1	1	0	0
		0	0	1	0	0	1	0	0
(1+2+3)+4		1	0	1	0	0	0	0	0
		1	0	0	0	0	1	0	0
	1	0	0	1	0	0	1	0	0
									1
subtraction		0	0	1	0	0	1	0	1
		1	1	1	1	1	1	1	1
		1	1	0	1	1	0	1	0

The sender sends:

1	2	3	4	checksum
10011001	11100010	00100100	10000100	11011010

The receiver:

(1+2+3+4)+ checksum	0	0	1	0	0	1	0	1
	1	1	0	1	1	0	1	0
subtract	1	1	1	1	1	1	1	1
	1	1	1	1	1	1	1	1
	0	0	0	0	0	0	0	0

In case the receiver doesn't get zero then an error occurred.

1.1.4 Cyclic Redundancy Check (CRC)

This is an error detection technique where redundant bits are appended at the end of the data to be sent. It is based on a generator that is known to both the receiver and the sender. The generator is a polynomial of degree n for n $\in \mathbb{Z}$. Take n =4 and the polynomial $x^3 + x^2 + 1$. We have coefficients of degree 3,2,0 but 1 is missing. Thus, the generator will be 1101. Take 101110001 to be the data the sender was to send. Since n=4, we append then we will append three zeros at the end of the data 1011100010000. We will then take the xor of the generator from the first leading one from the left as shown below:

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

The CRC is therefore 101. The sender will therefore transmit 101110001101. The receiver will use the generator to perform xor with the data they receive, from the leading one from the left as shown above. If the result is zero then no error occurred, otherwise, an error occurred.

1.2 Conclusion

The above methods are efficient for short strings and inefficient for reasonably long ones.