EEL7123 - Tópicos Avançados em Sistemas Digitais
Projeto: tópico 1

Alunos: Alexandre Hoffmann Genthner 16103372
Bruna Suemi Nagai 20100793

Análise das operações quadráticas modulares

A partir de valores dados dos módulos, foram obtidos pseudo-módulos de formato $\bmod(2^n,\ \mathrm{RSA}_i)$ pela análise do expoente n e pela quantidade de 1's na representação em binário ou em codificação Booth de k, que é o resto modular obtido para cada n.

Módulo original	Pseudo-módulo	k	Recodificação Booth de k	Quantidade de 1's
341	$\lvert 2^9 \rvert_{341} = 171$	$k_{171} = 010101011$	$k'_{171} =$ 1-11-1-110-1	5
341	$\lvert 2^{10} \rvert_{341} = 1$	$k_1 = 0000000001$	$k'_1 =$ 000000001-1	1
409	$\lvert 2^9 \rvert_{409} = 103$	$k_{103} = 001100111$	$k'_{103} =$ 010-10100-1	4
409	$\lvert 2^{10} \rvert_{409} = 206$	$k_{206} = 011001110$	$k'_{206} =$ 010-10100-10	4
409	$\lvert 2^{11} \rvert_{409} = 3$	$k_3 = 000000011$	$k'_3 =$ 000000010-1	2
5461	$\lvert 2^{13} \rvert_{5461} = 2731$	$k_{2731} = 1010101011$	$k'_{2731} =$ 1-11-11-11-11-110-1	7
5461	$\lvert 2^{14} \rvert_{5461} = 1$	$k_1 = 000000001$	$k'_1 =$0000000001-1	1
4681	$\lvert 2^{13} \rvert_{4681} = 3511$	$k_{3511} = 110110110111$	$k'_{3511} =$ 10-110-110-1100-1	8
4681	$\lvert 2^{14} \rvert_{4681} = 2341$	$k_{2341} = 100100100101$	$k'_{2341} =$ 01-101-101-101-11-1	5
4681	$\lvert 2^{15} \rvert_{4681} = 1$	$k_1 = 000000001$	$k'_1 =$ 000000001-1	1
21845	$\lvert 2^{15} \rvert_{21845} = 10923$	$k_{10923} = 101010101010111$	$k'_{10923} =$ 1-11-11-11-11-11-110-1	8
21845	$\lvert 2^{16} \rvert_{21845} = 1$	$k_1 = 000000001$	$k'_1 =$ 000000001-1	1
6553	$\lvert 2^{13} \rvert_{6553} = 1639$	$k_{1639} = 11001100111$	$k'_{1639} =$ -1010-10100-1	5
6553	$\lvert 2^{14} \rvert_{6553} = 3278$	$k_{3278} = 110011001110$	$k'_{3278} =$ 01-1010-10100-10	6
6553	$\lvert 2^{15} \rvert_{6553} = 3$	$k_3 = 000000011$	$k'_3 =$ 000000010-1	2
1363	$\lvert 2^{11} \rvert_{1363} = 685$	$k_{685} = 1010101101$	$k'_{685} =$ 1-11-11-110-11-1	6
1363	$\lvert 2^{12} \rvert_{1363} = 7$	$k_7 = 0000000111$	$k'_7 =$ 000000100-1	2

Tabela 1: Pseudo-módulos obtidos a partir das análises de k em representação binária e em codificação Booth.

Módulo original	Pseudo-módulo	n	k	Quantidade de 1's em k	Atraso [ns]
341	$2^9 - 171$	9	171	5	3.50
341	$2^{10} - 1$	10	1	1	1.60
409	$2^9 - 103$	9	103	4	3.00
409	$2^{10} - 206$	10	206	4	3.10
409	$2^{11} - 3$	11	3	2	2.20
5461	$2^{13} - 2731$	13	2731	7	4.90
5461	$2^{14} - 1$	14	1	1	2.00
4681	$2^{13} - 3511$	13	3511	8	5.40
4681	$2^{14} - 2341$	14	2341	5	4.00
4681	$2^{15} - 1$	15	1	1	2.10
21845	$2^{15} - 10923$	15	10923	8	5.60
21845	$2^{16} - 1$	16	1	1	2.20
6553	$2^{13} - 1639$	13	1639	5	3.90
6553	$2^{14} - 3278$	14	3278	6	4.50
6553	$2^{15} - 3$	15	3	2	2.60
1363	$2^{11} - 685$	11	685	6	4.20
1363	$2^{12} - 7$	12	7	2	2.30

Tabela 2: Tempo de atraso para cada pseudo-módulo.

Análise de aplicação em mineração de cripto-moeda

Para o cálculo do atraso foi utilizado a seguinte expressão: $\Delta = 1{+}0,1{\cdot}(n{-}4){+}0,5{\cdot}(\text{ones}{-}1)$ em que *ones* representa o número de elementos não nulos contabilizados no termo (i.e. igual a 1), inicialmente em binário e posteriormente recodificado pelo algoritmo de Booth que, a partir de consultas com materiais de apoio, segue a seguinte norma:

Multiplicador Bit i	Multiplicador Bit $i-1$	Recodificação
0	0	0
0	1	+1
1	0	-1
1	1	0

Os resultados para os primeiros números utilizados na codificação RSA e do maior número (RSA-2048), o valor de atraso mínimo e o valor de n cujo atraso foi o menor. A análise deste projeto levou em consideração que o objetivo final era obter o menor atraso possível e, como o cálculo do atraso depende tanto de n quanto de *ones*, ambas as variáveis foram avaliadas.

O código implementado em Python (Anexo A), recebe os valores modulares originais em decimal, faz a conversão para a representação binária e faz a codificação de Booth. A partir das duas codificações, é possível contar quantas vezes o número 1 aparece em cada uma das representações. Em relação ao valor de n, é possível obtê-lo pelo comprimento do vetor binário gerado. Assim, temos as informações necessárias para o cálculo do atraso. Esse procedimento é repetido para os próximos valores maiores do expoente n, e no caso foram testados para até as 2.000 primeiros valores. A partir dos vários testes, foi possível obter o valor ótimo para o menor atraso para cada um dos valores modulares RSA, onde temos os valores abaixo.

Além disso, temos os gráficos plotados para todos os valores de *ones* (em azul) e para os valores de atraso (em laranja), para alguns dos valores de RSA. Podemos observar que os *ones* aparentam ter uma distribuição aleatória e que o atraso não necessariamente aumenta proporcionalmente a n.

RSA100

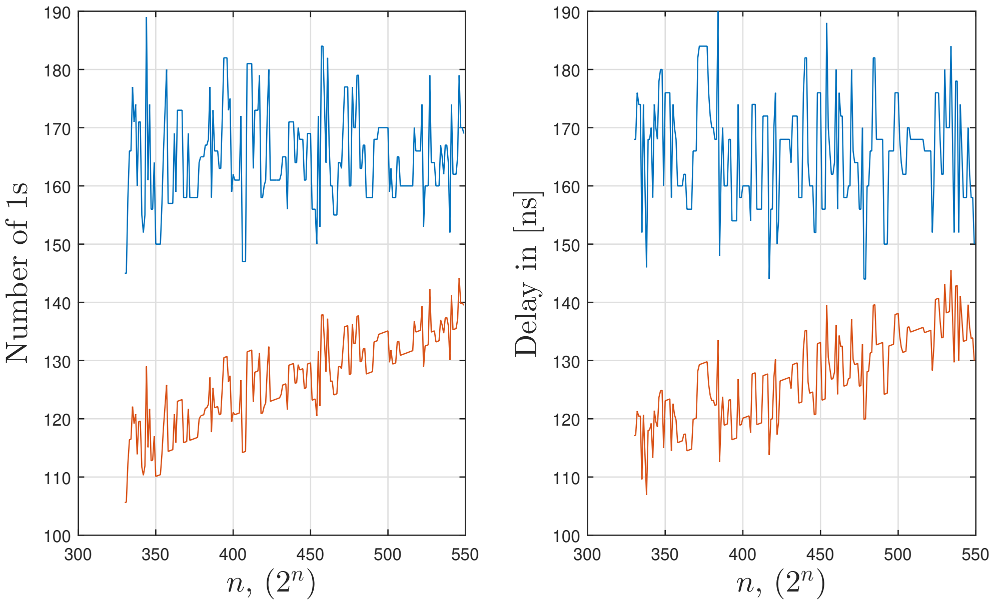

Figura 1: Número de 1's e o atraso calculado.

Atraso mínimo de 105,6 [ns] para $n = 330$ e $ones = 145$, utilizando coficação binária. Se utilizasse codificação Booth, teríamos um atraso mínimo de 106,9 [ns], para $n = 338$ e $ones = 146$.

RSA110

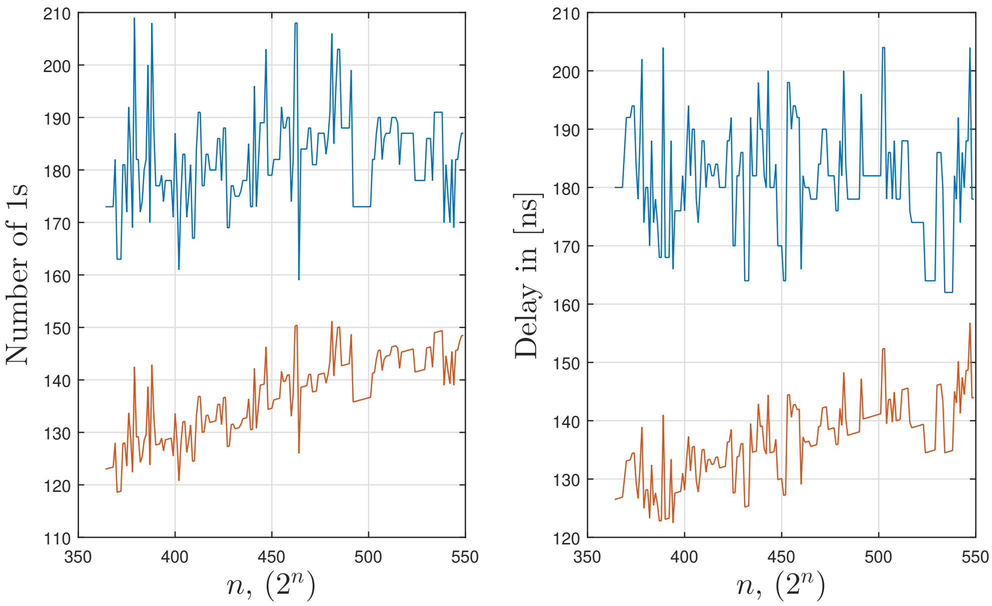

Figura 2: Número de 1's e o atraso calculado.

Atraso mínimo de 118,6 [ns] para $n = 370$ e $ones = 163$, utilizando a codificação binária. Se utilizasse codificação Booth, teríamos um atraso mínimo de 122,5 [ns], para $n = 394$ e $ones = 166$.

RSA120

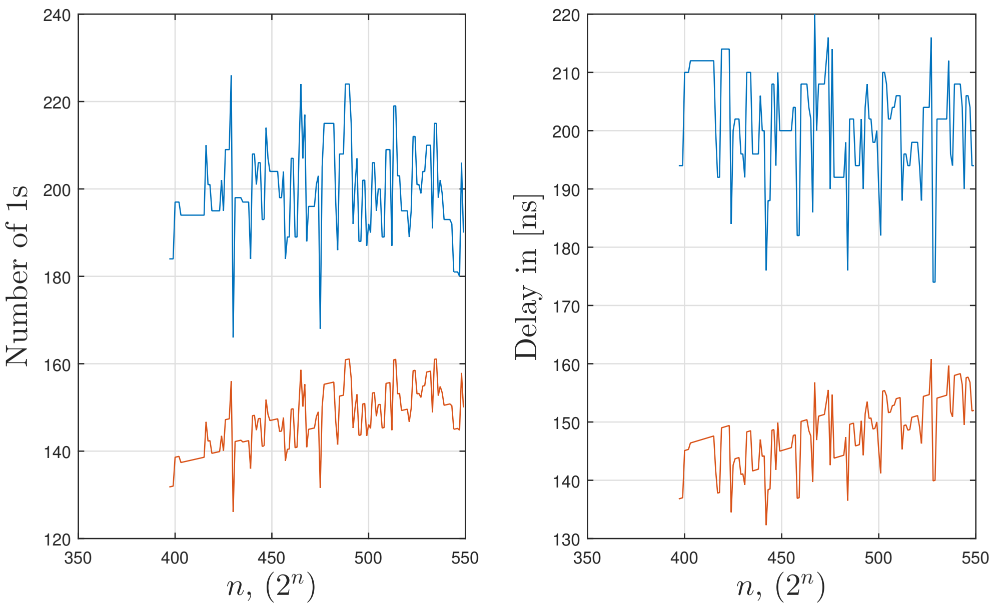

Figura 3: Número de 1's e o atraso calculado.

Atraso mínimo de 126,1 [ns] para $n = 430$ e $ones = 166$, utilizando a codificação binária. Se utilizasse codificação Booth, teríamos um atraso mínimo de 132,3 [ns], para $n = 442$ e

$ones = 176$.

RSA129

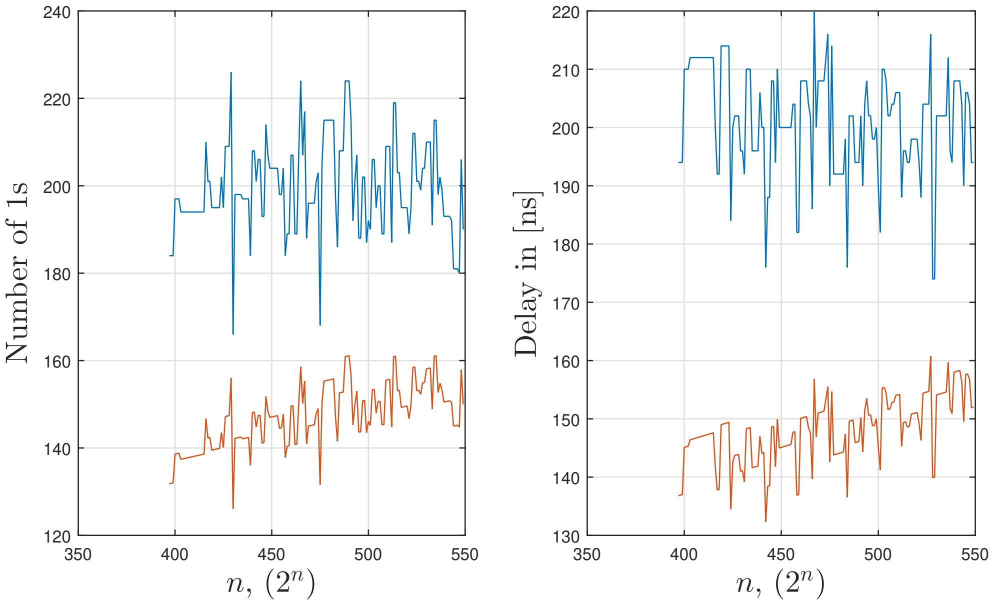

Figura 4: Número de 1's e o atraso calculado.

Atraso mínimo de 142,2 [ns] para $n = 471$ e $ones = 190$, utilizando a codificação binária. Se utilizasse codificação Booth, teríamos um atraso mínimo de 145,1 [ns], para $n = 460$ e $ones = 198$.

RSA130

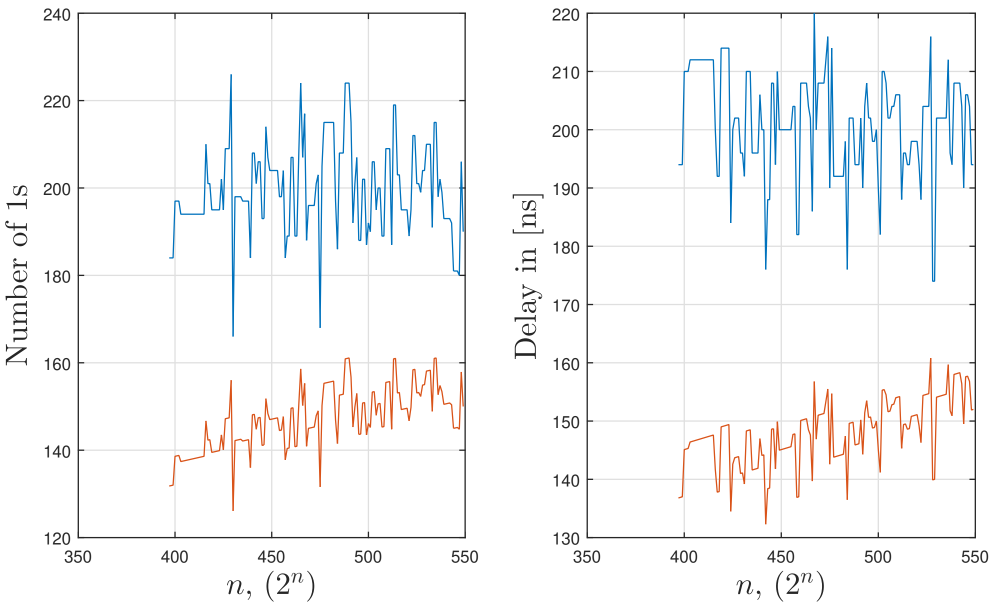

Figura 5: Número de 1's e o atraso calculado.

Atraso mínimo de 142,1 [ns] para $n = 450$ e $ones = 194$, utilizando codificação Booth. Se utilizasse codificação binária, teríamos um atraso mínimo de 144,6 [ns], para $n = 450$ e $ones = 200$.

RSA140

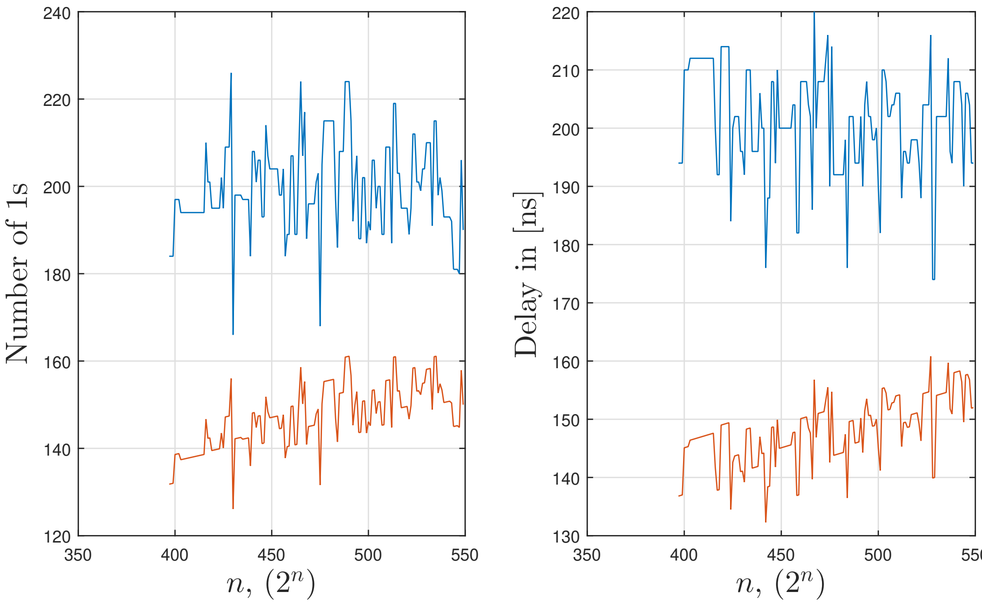

Figura 6: Número de 1's e o atraso calculado.

Atraso mínimo de 150,4 [ns] $paran = 503$ e $ones = 200$, utilizando a codificação binária. Se utilizasse codificação Booth, teríamos um atraso mínimo de 159,1 [ns], para $n = 470$ e $ones = 224$.

RSA2048

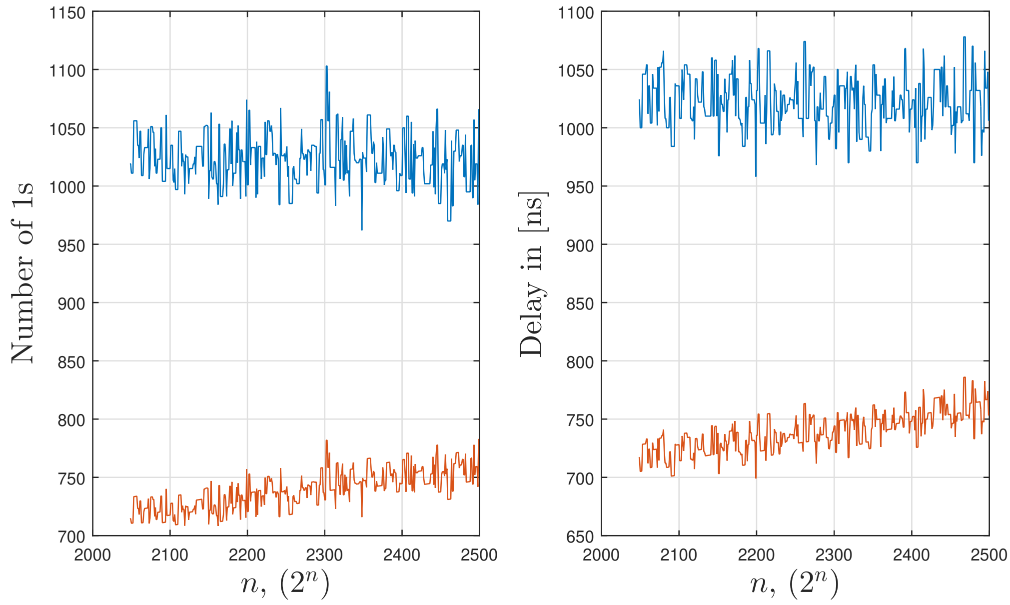

Figura 7: Número de 1's e o atraso calculado

Atraso mínimo de 699,0 [ns] para $n = 2199$ e $ones = 958$, utilizando a codificação Booth. Se utilizasse codificação binária, teríamos um atraso mínimo de 708,3 [ns], para $n = 2162$ e $ones = 984$.

Anexo A

```
if __name__ == '__main__':

    n = 200
    n_min = 0
    min_delay = 10000

    while n < 2500:

        entry = 2 ** n
        # mod = 1522605027922533360535618378132 6...
        #       37429718068114961380688657908494...
        #       58012296325895289765400035069200...
        #       6139 # RSA100
        # mod = 35794234179725868774991807832568...
        #       45540300377802422822619353290819...
        #       04846702523646774115135161112045...
        #       04060317568667 #RSA110
        #mod =  22701048129543736333425996094749...
        #       36688958753364660847800381732582...
        #       47009162675779735389791151574049...
        #       16674788048747029654847 9 #RSA120
        #mod =  11438162575788886766923577997614...
        #       66120102182967212423625625618429...
        #       35706935245733897830597123563958...
        #       70505898907514759929002687954354
        #       1 #RSA129
        #mod =  18070820886874048059516561644059...
        #       05566278102516769401349170127021...
        #       45005666254024404838734112759081...
        #       23033717818879665631820132148805...
        #       57 #RSA130
        #mod =  21290246318258757547497882016271...
        #       51749780670396327721627823338321...
        #       53819499840564959113665738530219...
        #       18316783107387995317230889569230...
        #       873441936471 #RSA140
        #mod =  22601385262034057849416540486101...
        #       97513508038915719776718321197768...
        #       10944564181796667660859312130658...
        #       25772506315628866769704480700018...
        #       11149711863002112487928199487482...
        #       06607013106658664608332798280356...
        #       03792053919801399464969552 61
        mod =   25195908475657893494027183240048...
                39857142928212620403202777713783...
                60436620207075955562640185258807...
                84406918290641249515082189298559...
                14917618450280848912007284499268...
                73928072877767359714183472702618...
                96375014971824691165077613379859...
                09570009733045974880842840179742...
                91006424586918171951187461215151...
                72654632282216869987549182422433...
                63725908514186546204357679842338...
                71847744479207399342365848238242...
```

```
                81198163815010674810451660377306...
                05620161967625613384414360383390...
                44149526344321901146575444541784...
                24020924616515723350778707749817...
                12577246796292638635637328991215...
                48314381678998850404453640235273...
                81951378636564391212010397122822...
                120720357 #RSA2048

        if mod > entry:
            #print('ERROR, MOD VALUE TO HIGH! %d' % n)
            n = n + 1
        else:
            number = entry % mod
            original_binary = bin(number)
            string_size = len(original_binary)

            original_list = list(original_binary)
            original_list[1] = '0'
            binary = "".join(original_list)

            cpy_binary = binary
            new_binary = binary

            new_binary += '0'
            cpy_binary += '0'

            string_list = list(new_binary)

            counter = 2
            counter_ones_original = 0
            counter_zeros = 0

            #print('The binary no. for %d >> 0b %s' % (number, binary))
            #print('string size :: %d' % string_size)

            while counter < string_size:
                if binary[counter] == '1':
                    counter_ones_original = counter_ones_original + 1
                else:
                    counter_zeros = counter_zeros + 1
                counter = counter + 1

            new_string_size = len(cpy_binary)
            counter = new_string_size
            aux_0 = 0
            aux_1 = 0

            while counter > 2:
                aux_0 = cpy_binary[counter-1]
                aux_1 = cpy_binary[counter-2]
                if aux_1 == '0':
                    if aux_0 == '0':
                        string_list[counter-1] = '0'
                    else:
                        string_list[counter-1] = '1'
                else:
                    if aux_0 == '0':
                        string_list[counter-1] = 'n'
```

```
                    else:
                        string_list[counter-1] = '0'
                counter -= 1

            new_new_binary = "".join(string_list)

            counter = 0
            counter_ones = 0

            while counter < new_string_size:
                if new_new_binary[counter] == '0':
                    counter_zeros = counter_zeros + 1
                else:
                    counter_ones = counter_ones + 1
                counter = counter + 1

            delay_original = 1 + 0.1*(n-4) + 0.5*(counter_ones_original-1)
            delay_recoded = 1 + 0.1*(n-4) + 0.5*(counter_ones-1)

            #print('extended multiplier :: %s' % cpy_binary)
            #print('recoded binary form :: %s' % new_new_binary)
            print('%d,%d,%.1f,%d,%.1f' % (n, counter_ones_original, ...
               delay_original, counter_ones, delay_recoded))

            if delay_recoded < delay_original:
                min_temp = delay_recoded
            else:
                min_temp = delay_original

            if min_temp < min_delay:
                min_delay = min_temp
                n_min = n

            n = n + 1
    print('delay minimum = %f for n = %d' % (min_delay, n_min))
```