INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO

Cryptography

"Key Schedule"

Abstact

Implementing the Key Schedule function of the DES Algorithm, in such a way that anyone can enter a key of 8 bytes and the number of the key it's required.

By:

Jose David Portilla Martinez

Professor:
MSc. NIDIA ASUNCIÓN CORTEZ DUARTE

October 2019

Introduction:

One key element of the DES Algorithm is an algorithm of itself, the Key Schedule Algorithm, which it generates 16 different keys depending of the stage. So the practice was the implementation of such algorithm with minor tweaks.

Literature review:

First, the DES algorithm is a symmetric key algorithm for encryption. DES is a block cipher — meaning it operates on plaintext blocks of a given size (64-bits) and returns ciphertext blocks of the same size. It implements Feistel block cipher. One function DES uses is the Key Schedule Algorithm, which is nothing but the collection of all the subkeys that would be used during various rounds.

A key schedule is an algorithm that expands a relatively short master key (typically between 40 and 256 bits long) to a relatively large expanded key (typically several hundred or thousand bits) for later use in an encryption and decryption algorithm. Key schedules are used in several ways:

a) To specify the round keys of a product cipher. DES [NBS77] uses its key schedule in this way, as do many other product ciphers.
b) To initialize some fixed elements of a cryptographic transform. Khufu [Mer91], Blowfish [Sch94], and SEAL [RC94] use a key schedule this way.
c) To initialize the state of a stream cipher prior to generating keystream. RC4 [Sch96] uses a key schedule in this way. Note that (b) and (c) are the only instances where synchronous stream ciphers can fall prey to any chosen-input attack.

Permuted Choice 1

The Permuted Choice 1 algorithm performs two functions in DES. DES uses a 56-bit key, but technically it takes a 64-bit key as input. The eighth bit of each of the 8 8-bit blocks of the key is specified as a parity bit and the first purpose of Permuted Choice 1 is to drop these bits. The second purpose is to permute the key prior to key expansion. The Permuted Choice 1 algorithm is shown in the Table 1.

57	49	41	33	25	17	9	63	55	47	39	31	23	15
1	58	50	42	34	26	18	7	62	54	46	38	30	22
10	2	59	51	43	35	27	14	6	61	53	45	37	29
19	11	3	60	52	44	36	21	13	5	28	20	12	4

Table 1: PC-1

Permuted Choice 2

After the shift occurs, the round key is generated using the Permuted Choice 2 (PC-2) algorithm. This algorithm is shown in the Table 2.

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

Table 2: PC-2

Shift operation

For each round, both subkeys are shifted to the left by a set amount. These shifts are cumulative and the shift amount for each round is shown in the Table 3.

Round	Shifts
1	1
2	1
3	2
4	2
5	2
6	2
7	2
8	2
9	1
10	2
11	2
12	2
13	2
14	2
15	2
16	1

Table 3: Shift Round

Software (libraries, packages, tools):

- For the GUI was used tkinter
- For the manipulation of the data was used bitstring

Results

While testing the program the key used was:

"Asegurar"

And the key necessaries were the number 2 and 8.

In the next figure it can be seen how the application is set.

Figure 1: Resting Interface

In figure 2 the key require was the number 2.

Figure 2: Testing key 2

In figure 3 the key require was the number 8.

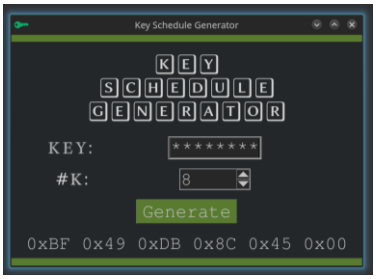

Figure 3: Testing key 8

Finally, it can also be shown a range of keys, in the following figure is shown every key

Figure 4: Testing key 1 through 16

Conclusions:

One thing that I learned was how the information really needed to be seen as bits, so the implementation was easier to make. And one error that happened often was that, as the bitstring was used, shifts and bit operations didn't work properly unless the data was the same length

Probably this practice would've been harder if the previous one wasn't necessary or even if the teacher hadn't helped us in the implementation.

The big takeaway could be that this algorithm implemented in a different way wouldn't had been so efficient as it was, for example, the use of bitwise operation not only did I make the things take whole of a lot of less space, if not that it make the procedure way faster.

References:

- Koblitz, N. (1996). *Advances in cryptology, CRYPTO '96*. Berlin: Springer.
- Medium. (2019). *Key Expansion Function and Key Schedule of DES(Data Encryption Standard) Algorithm*. [online] Available at: https://medium.com/@artistritul1995/key-expansion-function-and-key-schedule-of-des-data-encryption-standard-algorithm-1bfc7476157 [Accessed 23 Oct. 2019].
- Commonlounge.com. (2019). *A Detailed Description of DES and 3DES Algorithms (Data Encryption Standard and Triple DES) | CommonLounge*. [online] Available at: https://www.commonlounge.com/discussion/5c7c2828bf6b4724b806a9013a5a4b99 [Accessed 23 Oct. 2019].

Code

The following code is edited so I can be easily annexed in the document:

The graphic interface:

```
def generateKey():
        kN = keyNumber.get().split('-')
        if (len(kN) > 1):
                tmp = ''
                for i in range(int(kN[0]), int(kN[1]) + 1):
                        tmp += (' ' + str(i).zfill(2) + ') ' +
getNKey(keyOriginal.get(), i) + ' \n')
                nthKey.set(tmp)
        else:
                nthKey.set(' ' + getNKey(keyOriginal.get(), int(kN[0])) + ' ')

if __name__ == "__main__":
        root.minsize(width = 500, height = 500)
        root.maxsize(width = 850, height = 1400)
        root.title('Key Schedule Generator')
        root.configure(bg = bnColor)
        mainWindow = tk.Frame(root)
        mainWindow.pack(expand = 1)
        mainWindow.configure(bg = bgColor)
        tk.Label(mainWindow, text = 'KEY\nSCHEDULE\nGENERATOR', bg = bgColor, fg =
fgColor, font = ftTitle, pady = 10, padx = 10).grid(pady = 10, row = 0, columnspan = 3)
        tk.Label(mainWindow, text = 'KEY: ', bg = bgColor,  font = ftNormal).grid(padx =
5, pady = 10, row = 10, column = 0)
        tk.Entry(mainWindow, textvariable = keyOriginal, bd = 3, font = ftButton, show =
'*', width = 8).grid(padx = 5, pady = 10, row = 10, column = 1)
        tk.Label(mainWindow, text = '#K:', bg = bgColor, fg = fgColor, font =
ftNormal).grid(padx = 5, pady = 10, row = 11, column = 0)
        tk.Spinbox(mainWindow, textvariable = keyNumber,  from_ = 1, to = 16, font =
ftButton, width = 5, wrap = True).grid(padx = 5, pady = 10, row = 11, column = 1)
        tk.Button(mainWindow, text = "Generate", font = ftButton, bg = bnColor, command =
generateKey).grid(padx = 5, pady = 10, row = 14, column = 0, columnspan = 3)
        tk.Label(mainWindow, textvariable = nthKey, font = ftButton, bg =
bgColor).grid(padx = 6, pady = 10, row = 15, column = 0, columnspan = 3)
        nthKey.set('---->nth Key<----')
        img = tk.Image("photo", file = "keyi.gif")
        root.wait_visibility(root)
        root.wm_attributes('-alpha', 0.85)
        root.tk.call('wm','iconphoto', root._w, img)
        root.mainloop()
```

The algorithm itself, omitting the array declarations:

```
def bytesFormatted(arrHex):
	return " ".join('0x{0:0{1}X}'.format(n, 2) for n in arrHex)

def permutedChoice(keyInput, PC):
	keyPermuted, j, bit = [0] * 7, 0, 0
	for i in (arrPC2 if PC else arrPC1):
		if (bit == 8): j, bit = j + 1, 0
		row = i >> 3
		column = i % 8
		if (i % 8 == 0): row -= 1
		if ((ord(keyInput[row]) & (128 >> ((column - 1) % 8)))):
			keyPermuted[j] |= (128 >> bit)
		bit += 1

	return keyPermuted

def keyGenerator(keyPermuted, nRound):
	nShifts = shiftRoundACC[nRound - 1]
	C0 = BitArray(bytesFormatted(keyPermuted[:3] + [keyPermuted[3]])) >> 4
	D0 = BitArray(bytesFormatted([keyPermuted[3] & 15] + keyPermuted[4:]))
	CN = ((C0 << nShifts) | (C0 >> (28 - nShifts))) & BitArray('0x0fffffff')
	DN = ((D0 << nShifts) | (D0 >> (28 - nShifts))) & BitArray('0x0fffffff')
	CN.append('0x0000000')
	KEYN = BitArray(hex((CN.int | DN.int)))
	if (KEYN.len < 56):
		KEYN.prepend('0x' + ('0' * ((56  - KEYN.len) // 4)))
	return ([i for i in permutedChoice([chr(int(KEYN.hex[i : i + 2], 16))
                                        for i in range(0, len(KEYN.hex),
2)], 1)][:-1])

def getNKey(keyString, keyNumber):
	return bytesFormatted(keyGenerator(permutedChoice(keyString, 0),
keyNumber))
```