Radboud University

Bachelor Thesis

Government intervention on consumer crypto hardware: A look at the PX-1000 before and after the NSA's involvement.

Author:
Ben Brücker

Supervisor:
Prof.dr. Bart Jacobs

A thesis submitted in fulfilment of the requirements for the degree of Bachelor of Science

at the

FNWI

July 2014

RADBOUD UNIVERSITY

Abstract

FNWI

Bachelor of Science

Government intervention on consumer crypto hardware: A look at the PX-1000 before and after the NSA's involvement.

by Ben Brücker

"The PX-1000 is a device capable of transmitting encrypted text messages over telephone lines. We show that the original version that was produced by TextLite implements the DES algorithm correctly. In 1983, when Philips USFA distributed this device as the PX-1000Cr, the NSA made a deal with Philips USFA to implement an alternative algorithm. For this alternative version we show that it does implement a stream cipher instead of the DES algorithm."

Acknowledgements

I would like to express my gratitude to my supervisor Prof.dr. Bart Jacobs for his assistance and guidance and for providing me the great subject that led to this thesis. Furthermore I would like to thank Marc Simons and Paul Reuvers of the Crypto Museum for supplying all the material and context around the PX-1000, without them this thesis would not have been possible. Also, I like to thank the second reader Peter Schwabe, his ideas and tips will give direction to the continued research on this subject.

I would like to thank my partner Marloes, who has supported me throughout entire process, both by keeping me sane and helping me putting pieces together. I will be grateful forever for your love. And last but not least, my family, in-laws and friends for all your support and love.

Contents

List of Figures

Chapter 1

Introduction

1.1 The goal of this bachelor thesis

In this bachelor thesis we will take a closer look at the cryptographic algorithms used in the PX-1000 developed by TextLite and the PX-1000Cr by Philips.
First we will look at the PX-1000 version which is presumed to use the DES algorithm. Then we will analyze the PX-1000Cr. This device implements unknown cryptography supplied by the American National Security Agency (NSA) to Philips USFA.

Since this thesis deals with old material and a hands-on topic, there are very few academic references. Therefore this will be a one-off research instead of trying to further already existing work.

1.2 Historical context of the PX-1000

The PX-1000 is a small hand-held device, capable of sending and receiving text messages over phone lines. The PX-1000 was originally developed by the dutch company Text Lite BV. They themselves distributed two versions [2]:

- Text Tell PX-1000: The standard version, featuring DES based encryption.
- Text Tell PX-1000 CALC E: This version forgoes the en/decryption feature in favor of an arithmetic calculator.

The initial version of the Text Tell implemented the Data Encryption Standard (DES). Therefore it was able to encrypt messages securely (for that time) for transmission over phone lines [3].

FIGURE 1.1: The original Text Tell PX-1000 [1]

In the early 1980's, Philips Electronics bought the right to distribute Text Lite's Text Tell in the Netherlands and re-branded it to the Philips PX-1000 [1].

Philips-USFA (UltraSoneFabriek) was Philips branch involved in military and cryptographic research for NATO and the Dutch government. According to sources of the Crypto Museum [1] they had already implemented an encryption scheme "SAVILLE" supplied by the NSA in the Spendex 40 secure phone. [1]

Research by Marcel Metze of the Groene Amsterdammer [4] presumes that in 1983, the NSA began worrying about powerful encryption in the hands of the general public. They shared this concern with Philips-USFA since they were selling the PX-1000 as a relatively cheap device, available to anyone.

Since the NSA already had a relation with Philips-USFA [1], they requested that the PX-1000 got replaced with a version that implements encryption that was supplied by the NSA. It is unknown how exactly this new algorithm works since the details are not disclosed to the general public. But the Text Lite engineers were told that the algorithm could be in strength compared to DES. [1]

Again according to resources of the crypto museum [1], Philips then proceeded to buy back all 12.000 'old' PX-1000 that were already produced, together with 20.000 already produced firmware PROMS for PX-1000. All these were later sold to the NSA for NLG 16,6 million.

For this bachelor thesis we are in possession of memory dumps of both the original PX-1000's ROM, and the EPROM with the revised algorithm by the NSA. These memory

dumps were created by the Crypto Museum by reading the original EPROM and ROM with a specialized reader device. The dumps are available on request from the Crypto Museum, or the author of this thesis.

1.3 Historical use of the PX-1000; Operation Vula

One case where the PX-1000 has been used in an important historical context is Operation Vula, as described by Conny Braam in her book 'Operatie Vula' [5]. This operation tells the story of the anti-apartheid revolutionary (and later President of South Africa) Nelson Mandela [6].

Mandela was the leader of the South African organization ANC (African National Congress). In 1962 he was arrested and convicted for conspiracy to overthrow the state. He was sentenced to imprisonment for life. All over the world people fought for his release and for his goal of abolishing Apartheid [6].

Operation Vula was an underground operation that was put into motion in 1986 [7]. It's goal was to smuggle freedom fighters into South Africa and maintain open communication links between the ANC leaders at home, in prison and in exile. This plan was set into motion by Mac Maharaj, ANC's intelligence officer. Part of this operation was preparing Mandela for his release.

Connie Braam is a Dutch anti-apartheid's activist, and at the time she led the Dutch contribution to Operation Vula [5]. Besides the more physical part of the clandestine operation to communicate with Mandela, like finding make-up artists and undercover couriers. Her task was also to ensure secure communication. When Connie Braam contacted a Philips engineer, he suggested the PX-1000 to her [7]. London was the hub for all communication between the Netherlands, South Africa, Great Britain and later also Lusaka. The PX-1000 was then acquired and put to use to secure the communication between Amsterdam, London and South Africa. Some presume that the PX-1000 link might still have been in use, even after the end of operation Vula [1].

1.4 Was the encrypted link to Mandela secure?

There are doubts about the security of the NSA version of the PX-1000 [1]. Since the NSA insisted that the ROM would be replaced by one of their own making, it raises the suspicion that they might have been able to listen in on conversations encrypted by the PX-1000. In order to find out if these claims have any merit, this bachelor thesis will reverse engineer the code from both versions of the PX-1000.

1.5 Reverse Engineering

The encryption scheme the NSA supplied is not common knowledge. But in order to see how secure it really was, we have to recover the original code from the device itself.

Software reverse engineering is usually done to retrieve the source code of a program in a number of cases [8]:

- The source code was lost,
- to improve the performance of a program,
- To fix a bug (correct an error in the program when the source code is not available),
- To identify malicious content in a program such as a virus,
- To identify malicious content in a program such as a virus or to adapt a program written for use with one microprocessor for use with another.

We are dealing with the first case: The source code was lost (or more specifically, never available to the public). Since this reverse engineering is done from legally obtained material, it should not constitute a copyright violation [9].

In this case, we have the machine code [8], the executable representation of software. In the current day and age, this is typically the result of translating a program written in a high-level language, using a compiler, to an object file, a file which contains platform-specific machine instructions. The object file is made executable using linker, a tool which resolves the external dependencies that the object file has, such as operating system libraries [8]. In the case of the PX-1000 source, a low level programming language such as Assembly will probably have been used, because there are a number of interesting low level bit manipulations that are used to save memory. But this is still considered high level by a CPU [8].

Reverse engineering entails using several different tools. For example a Hex-Editor that translates the binary file to a hexadecimal representation. This reveals human-readable strings, and the hex representation can be matched to CPU opcodes. Another common tool is the disassembler. This tool reads the binary code and then displays each executable instruction in text form. A disassembler cannot give contect to executable instructions and the data used by the program. Hench manual human labor is required [8].

Please see the method chapter for how reverse engineering was done in this theses.

Chapter 2

Research questions

2.1 Main question

Does the pocket computer PX-1000 by Philips contain a different and presumably weaker encryption algorithm than DES after the intervention of the NSA in 1980?

2.2 Sub-question

The main question will be answered by answering the following sub-questions:

- Is it possible to reverse engineer the encryption algorithm from the original ROMS?
- Does the original version of this device implement the DES algorithm?
- How does encryption in the modified code work and compare to DES?

Chapter 3

Method

3.1 Overview of the method

The method to recover the crypto algorithms used in both versions of the PX-1000 consists of 4 distinct steps:

1. Recover assembly code of both the 1983 and 1984 version of the PX-1000 with the DASMx disassembler [10]. And create a memory map,
2. Annotate the source code and create a rough description of the original code,
3. Show that the original code indeed implements DES,
4. Identify and describe the alternative algorithm and compare it to DES.

3.2 Recovering assembly

3.2.1 Disassembly with DASMx

Disassemblers are powerful tools for creating assembly, or assembly-like code from binary images. DASMx, is a free microprocessor opcode disassembler created by Conquest Consultants [10]. The most recent version, and the version used in this bachelor thesis is version 1.30 from October 6th 1999. This program is old, but it has support for the chip used in the PX-1000 and is well suited for the device from 1983 [11].

DASMx supports code-threading. This means that code entry points can be supplied. For example interrupt, org and reset vectors. The disassembler will use each of these

code entry points as known locations for code and will partially emulate the processor. This means that it executes instructions and follows all calls to subroutines and branches. Together with DASMx's multi-pass functionality it can find and disassemble most sections of code from the binary image [10].

3.2.2 Disassembling the PX-1000 ROM and EPROM

The PX-1000 uses the Hitachi HD6303RP CMOS MPU. This 8-bit Micro Processing Unit is completely compatible with the HD6301V1 instruction set. This in turn is compatible with the Motorola 6800 series, and can be disassembled with the DASMx disassembler. The difference between the Hitachi and the Motorola versions is that the Hitachi includes a few additional instructions and optimized pipelining [12].

The Crypto Museum [1] was able to extract the original ROM and EPROM image from both versions of the PX-1000.

The first stage in obtaining the disassembly is checking whether the ROM or EPROM file does have the required size. According to the documentation [3] the PX-1000 has 8kb of ROM or EPROM. This corresponds to the filesizes of files supplied by the Crypto Museum [1].

The next step is supplying known code entry points in a symbol file. The initial parameters used in this bachelor thesis can be found in Appendix A.

The memory categories and corresponding symbols are taken from the PX-1000 Service manual [3]. The entry vectors were taken from the DASMx ebcgame example [10] which incidentally also uses a Motorola 6803.

We can produce a listing and assembly file by running the following commands:

```
DASMx -t <infile.bin>
DASMx -t -a <infile.bin>
(-t = enable code threading, -a = Enable assembly output)
```

The listing output shows the details of every command, including all memory addresses and the opcodes in hexadecimal form. An example snippet of the listing output can be found in figure 3.1 The assembly output leaves out a lot of the detail, but creates code that is easier to read for humans. The snippet in figure 3.2 covers the same code as the listing snippet.

```
E000                start:
E000 : 8E 03 10     "  "        lds     #$0310
E003 : 0F           "  "        sei
E004 : 4F           "O "        clra
E005 : 97 08        "  "        staa    timerCSR
E007 : 86 CA        "  "        ldaa    #$CA
E009 : 97 02        "  "        staa    ioPort1data
E00B : 86 DE        "  "        ldaa    #$DE
E00D : 97 00        "  "        staa    ioPort1dataDirection
E00F : 86 14        "  "        ldaa    #$14
E011 : 97 03        "  "        staa    ioPort2data
E013 : 86 14        "  "        ldaa    #$14
E015 : 97 01        "  "        staa    ioPort2dataDirection
```

FIGURE 3.1: Example of the DASMx listing output

```
start:
    lds    #$0310
    sei
    clra
    staa    timerCSR
    ldaa    #$CA
    staa    ioPort1data
    ldaa    #$DE
    staa    ioPort1dataDirection
    ldaa    #$14
    staa    ioPort2data
    ldaa    #$14
    staa    ioPort2dataDirection
```

FIGURE 3.2: Example of the DASMx assembly output

3.2.3 Problems during disassembly

The code-threading of the disassembly might fail in a number of cases according to the DASMx manual [10]:

1. Pushing an address onto the stack and later performing a return from subroutine,
2. Indexed branch instructions,
3. Use of undocumented opcodes,
4. Self-modifying code.

In both cases of the PX-1000 disassembly we encounter case 2: Indexed branch instructions. The relevant part of the code is shown in figure 3.3.

```
LE195:
    suba    #$90
    tab
    ldx    #$E1A3
    abx
    abx
    ldx    $00,x
    jsr    $00,x ;INFO: index jump
    bra    LE15E
```

FIGURE 3.3: Indexed branch instruction

In this case, the code uses $0xE1A3$ as the starting point of a jump table and adds an offset to the index register to find the correct table entry. Manual inspection shows that this is a table of 36 addresses, and can therefore be defined as a vector table:

```
vectab 0xE1A3  Table:JumpTable 0x24
```

Another problem that we encounter, is the use of undocumented instructions. Undocumented instructions are opcodes that should not be used and that are not documented in the manuals and other documentation of a processor. This makes some parts of the code much harder to analyze. One examples of this is the following instruction:

```
EEEA : 71 7F 02        "q  "        aim    #$7FioPort1data
```

Opcode $0x71$ or the mnemonic *aim* can not be found in the documentation. For each of these instructions, manual research is required.

3.3 Annotating the source code and creating a rough description of the original code.

Creating readable code from the given assembly is very labor intensive, since it requires the reader to understand the intention of the programmer.

The process of transforming the source code to a readable description of the original code involves three steps [10]:

1. Running the disassembler on the binary file to produce a listing or assembly output.

2. Manually inspect the listing or assembly file to find out what some of the code does.

3. When a section of known code is identified, update the listing file to reflect the discovered code.

These 3 steps are a iterative process, as shown in figure 3.4.

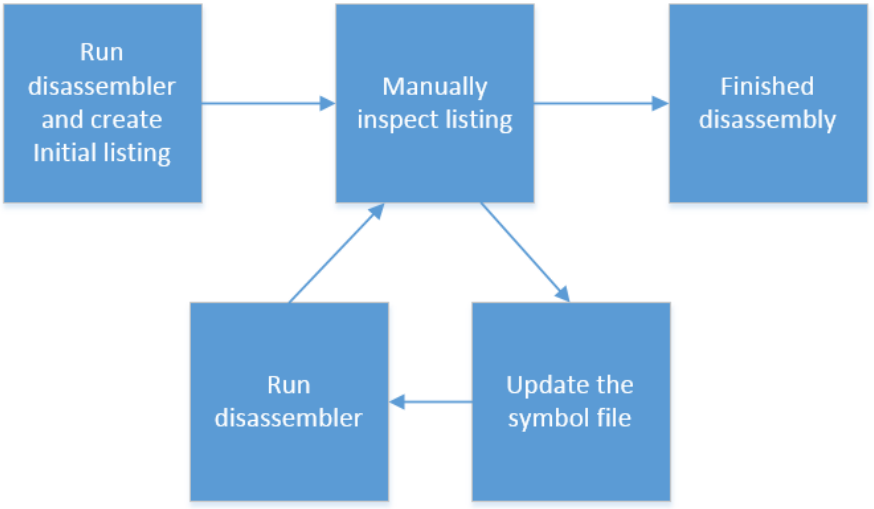

FIGURE 3.4: Schematic presentation of the disassembly process

3.3.1 Annotating source code by example

To show the methodology behind this, I will demonstrate the process of annotating and describing source code by using an example.

The code snippet in figure 3.5 is part of the EPROM dump of the PX-1000 version. This code is an entire function, starting at memory address FE21, and it runs until it encounters the rts (return from subroutine) [12] command on either line 35 or line 38.

3.3.1.1 Manual Inspection, Round 1

During the manual inspection of the code in figure 3.5 we notice the following facts:

- There is a circular reference, usually representing some kind of loop:
  On line 34, bls (branch if lower or same) jumps back to address LFE24. This happens when the value in the index register is lower or same, as the value of X0027 (line 34).

```
 1 LFE21:                  20     bls    LFE4C
 2     ldx    X0025        21     cmpa    #$61
 3     clrb                22     bcs    LFE59
 4 LFE24:                  23     cmpa    #$66
 5     ldaa    $00,x       24     bhi    LFE59
 6     anda    #$7F        25 LFE4C:
 7     cmpa    #$0D        26     eorb    #$01
 8     beq    LFE53        27     bra    LFE53
 9     cmpa    #$20        28 LFE50:
10     beq    LFE50        29     tstb
11     cmpa    #$2C        30     bne    LFE59
12     beq    LFE50        31 LFE53:
13     cmpa    #$30        32     inx
14     bcs    LFE59        33     cpx    X0027
15     cmpa    #$39        34     bls    LFE24
16     bls    LFE4C        35     rts
17     cmpa    #$41        36 LFE59:
18     bcs    LFE59        37     sec
19     cmpa    #$46        38     rts
```

FIGURE 3.5: Snippet of uncommented PX-1000 source code

The only other operations where the index register is used are on line 2 and on line 32. On line 2, the index register becomes the value of X0025 and on line 32 the index register is increased by one.
From all these facts we can deduce that here is a loop that has as initial value the value of X0025, increases by 1 every iteration and stops when it is greater then the value of X0027.
Therefore we can add the following line to our symbol file:
`code 0xFE24 whileX0025<=X0027_increase_x`

- From earlier disassembly, we already know that X0025 represents the first character of a string in memory. Therefore we can add this fact to the symbol file.
  `symbol 0x0025 BeginningOfString`

- This function has two rts (return from subroutine) commands. On line 35 and line 38. The main difference between the two is that the second rts on line 38 first has the carry-bit set by the sec (set carry) [12] command. Therefore we identify these two sections as the following parts of the disassembly:
  `code 0xFE59 Sub:ReturnAndSetCarry`
  `code 0xFE53 Sub:ReturnWithoutCarry`

In order to identify what the rest of the function does, we will follow the disassembly process described in figure 3.4. Therefore the next step is to update the symbol file.

Now the disassembler is run again, and the findings are manually inspected. The result of this action is figure 3.6.

```
 1 LFE21:                              20     bls    LFE4C
 2     ldx    BeginningOfString        21     cmpa    #$61
 3     clrb                            22     bcs    Fnc:ReturnAndSetCarry
 4 while:X0025<=X0027_increase_x:      23     cmpa    #$66
 5     ldaa    $00,x                   24     bhi    Fnc:ReturnAndSetCarry
 6     anda    #$7F                    25 LFE4C:
 7     cmpa    #$0D                    26     eorb    #$01
 8     beq    Fnc:ReturnWithoutCarry   27     bra    Fnc:ReturnWithoutCarry
 9     cmpa    #$20                    28 LFE50:
10     beq    LFE50                    29     tstb
11     cmpa    #$2C                    30     bne    ReturnAndSetCarry
12     beq    LFE50                    31 Fnc:ReturnWithoutCarry:
13     cmpa    #$30                    32     inx
14     bcs    Fnc:ReturnAndSetCarry    33     cpx    X0027
15     cmpa    #$39                    34     bls    LFE24
16     bls    LFE4C                    35     rts
17     cmpa    #$41                    36 Fnc:ReturnAndSetCarry:
18     bcs    Fnc:ReturnAndSetCarry    37     sec
19     cmpa    #$46                    38     rts
```

FIGURE 3.6: Snippet of commented PX-1000 source code

3.3.1.2 Manual Inspection, Round 2

Continuing the manual inspection, we can identify the following parts:

- On line 26, the function eorb (exclusive-or on accumulator b) is called with the hex value 0x01. The only other uses of accumulator b are on line 3 (clrb, clear accumulator b) and on line 29 (tstb, test accumulator b and set status flags). Following the logic in this part of the source, we see that b is used as some kind of boolean, where 0 = false and 1 = true. Therefore we can update the symbol file with:
  `code 0xFE4C Fnc:set_b_true`
- Since this function loops through a number of memory addresses starting at "BeginningOfString" it is reasonable to assume that this function processes characters. On line 6, an AND operation with 0x7F is performed. This results in setting the most significant bit of the current byte stored in a to 0 (since 0x7F equals the bitmask 01111111). This is required because end-of-string characters in the PX-1000 source are identified by setting the most significant bit to 1. By stripping this bit, we can read the actual character.

- We are dealing with strings, so we suspect that the hexadecimal values given are the hexadecimal representations of ASCII characters. By checking these, we find that:

LineNr	Hex Number	ASCII representation
7	0x0D	(CR)
9	0x20	(Space)
11	0x2C	,
13	0x30	0
15	0x39	9
17	0x41	A
19	0x46	F
21	0x61	a
23	0x66	f

- The last unknown part of this subroutine is now line 28, LFE50. This part tests accumulator b and if it does not equal 0 (false) it branches to ReturnAndSetCarry. This breaks the loop that checks all characters in the string. Therefore the following symbol will be added to the symbol file:
  `code 0xFE50 break:if_b_not_false`

3.3.1.3 Manual Inspection, Conclusion

Now that all parts of this routine are known, the next step is to identify what it actually computes.

1. Loop through a string starting at BeginningOfString,
2. Ignore all spaces and comma's at the start of the string,
3. After the first non-space and non-comma character is encountered, check the remaining characters one by one,
4. When all later characters (until end-of-string, or the first CR) represent a hexadecimal number (a-f,A-F,0-9) return from the subroutine without the carry-bit being set. Otherwise set the carry-bit and return.

Conclusion: This function checks if a string holds a hexadecimal representation of a number. If not so, it sets the carry-bit. Otherwise it leaves the carry-bit unset. So the following symbol will be added to the symbol file:
`code 0xFE21 Sub:TestIfStringIsHex`
The final code is referenced in figure 3.7.

```
 1 TestIfStringIsHex:                20     bls    Fnc:set_b_true
 2     ldx    BeginningOfString      21     cmpa    #$61
 3     clrb                          22     bcs    Fnc:ReturnAndSetCarry
 4 while:X0025<=X0027_increase_x:    23     cmpa    #$66
 5     ldaa    $00,x                 24     bhi    Fnc:ReturnAndSetCarry
 6     anda    #$7F                  25 Fnc:set_b_true:
 7     cmpa    #$0D                  26     eorb    #$01
 8     beq    Fnc:ReturnWithoutCarry 27     bra    Fnc:ReturnWithoutCarry
 9     cmpa    #$20                  28 break:if_b_not_false:
10     beq    break:if_b_not_false   29     tstb
11     cmpa    #$2C                  30     bne    ReturnAndSetCarry
12     beq    break:if_b_not_false   31 Fnc:ReturnWithoutCarry:
13     cmpa    #$30                  32     inx
14     bcs    Fnc:ReturnAndSetCarry  33     cpx    X0027
15     cmpa    #$39                  34     bls    LFE24
16     bls    Fnc:set_b_true         35     rts
17     cmpa    #$41                  36 Fnc:ReturnAndSetCarry:
18     bcs    Fnc:ReturnAndSetCarry  37     sec
19     cmpa    #$46                  38     rts
```

FIGURE 3.7: Snippet of completed PX-1000 source code

3.4 Show that the original code implements DES

There is no formal method to identify a DES implementation, especially since the DES standard only describes the algorithm and not the actual implementation. So we will have to look for all the following identifying features of DES in the source itself [13]:

- Initial and final permutation steps,
- 16 identical rounds,
- Splitting of 64-bits of input into 2 32-bit halves,
- The signature Feistel scheme,
- A Feistel function implementing the following steps:
  - Expansion: Expanding the 32-bit half-block to 48-bits,
  - Key mixing: Using the XOR operation between the 48-bit input and the 48-bit round key,
  - Substitution: Split the resulting 48-bits into 8 6-bit blocks. Then feed these blocks to the S-boxes and transform each 6-bit block into a 4-bit block,
  - Permutation: Permute the resulting 32 output bits according to a fixed permutation scheme.

- A key scheduling algorithm that generates the round keys from an initial 56-bit key.

Chapter 4

Results of the PX-1000 analysis

For the original PX-1000 this bachelor thesis focused on the section of the source code that implements the encryption. We were able to identify the DES function that takes 64 bits of plain- or cyphertext input and 56 bits of key material in order to create 64 bits of output [13]. Here we will describe the key features of the DES implementation in the PX-1000 and show that this is implemented correctly.

4.1 DES Permutations

The PX-1000 makes use of a single function for all permutation operations in the DES encryption. Specifically [13]:

- Initial permutation (IP). In the initial phase of DES, each bit of the 64 bits of input is subject to the initial permutation. This rearranges the entire block.
- Final permutation (IP^{-1}). In the final phase, the result is permuted with the exact inverse of the initial permutation.
- Expansion function (E). Each round of the DES algorithm will expand the right (R_i) half of the key (32 bits) to 48 bits by duplicating some bits. This is done in order to match the length of (R_i) to the length of the Round Key (K_i).
- Permutation (P). After the S-Box substitution in every round of the DES algorithm, the resulting 32 bits will be permuted. This permutation is designed so that each of the four S-Box output bits are spread over 4 different S-Boxes in the next round.

- Permuted choice 1 (PC-1). Permuted Choice 1 selects the 56 key bits that will be used from the initial 64 bits of key material.
- Permuted choice 2 (PC-2). Permuted Choice 2 is part of the Key Schedule and will select the 48 bit subkey from the given 56 bit key.

The permutation function in the PX-1000 is located between address 0xF5AF and 0xF639. Before this function is called, the Permutation_Table_Index (0x0119) is set to an offset for 0xF9CC that will reference a table holding all the parameters for the current permutation. For an overview of these parameters see table 4.1.

Perm_Table_Index	Input addr	Output addr	Perm Table addr	Perm Table length
0xFF (PC-1)	003A	00C2	0xFA82	0x3F
0x08 (PC-2)	00C2	00CA	0xFAC2	0x2F
0x11 (IP)	010E	00D8	0xFA02	0x3F
0x1A (E)	00DC	00C2	0xFAF2	0x2F
0x23 (P)	00CA	00CA	0xFB22	0x1F
0x2C (IP^{-1})	00D8	010E	0xFA42	0x3F

TABLE 4.1: The parameters for the PX-1000 permutation function

The actual permutation table is different from the standard table because it uses another form of encoding. While the normal DES table is a one to one match from an input to an output bit, the PX-1000 choses a more specific approach. The lower three bits denote byte offset for the input bytes and the next three bits indicate the bit position in this input byte. See figure 4.1 for the corresponding code.

```
Sub:DES_Permutation:
  # Start of by referencing the end
  # of the Permutation_Table_Index
  ldaa   Permutation_Table_Index
  adda   #$09
  staa   Permutation_Table_Index
  staa   Temp
  ldaa   #$08
  staa   Counter

loop:Initilize_Perm_Param:
  # Read the parameters in offset
  # by the Permutation_Table_Index
  # from F9CC.
  # Store these parameters in
  # 0x00E0 - 0x00E8
  clr    X00D4
  ldx    #$F9CC
  ldab   Temp
  abx
  ldaa   $00,x
  clr    X00D2
  ldx    X00D2
  staa   $E0,x
  dec    Temp
  dec    Counter
  bge    loop:Initilize_Perm_Param

for:Each_Output_Byte:
  ldaa   X00E7
  staa   InnerLoopCounter
  clr    Counter

for:Each_Output_Bit:
  # For each bit in the output, find
  # the matching pit in the input.
  # See the figure about Permutation
  # Encoding for details
  ldx    PermutationTableLocation
  ldab   PermutationTableLength
  abx
  clr    X00D2
  ldaa   $00,x
  staa   Temp
  anda   #$07
  tab
  ldaa   Temp
  stab   Temp
  asra
  asra
  asra
  staa   X00D7
  ldx    PermutationInput
  clr    X00D4
  ldab   Temp
  abx
  ldaa   $00,x
  ldx    #$FB42
  ldab   X00D7
  abx
  anda   $00,x
  bne    Fnc:Perm_bit_1
  ldab   #$00
  orab   Counter
  bra    Fnc:Perm_bit_0

Fnc:Perm_bit_1:
  ldab   #$01
  orab   Counter

Fnc:Perm_bit_0:
  stab   Counter
  ldaa   Counter
  dec    PermutationTableLength
  bmi    break:permutation_finished
  dec    InnerLoopCounter
  beq    beq break:all_bits
  asl    Counter
  bra    for:Each_Output_Bit::

break:all_bits_finished:
  psha
  jmp    for:Each_Output_Byte

break:permutation_finished:
  psha
  ldx    PermutationOutput
  ldab   X00E8
  abx

Fnc:Store_Perm_Results:
  # Store the permutation results at
  # loc starting in PermutationOutput
  # Specified in the parameters
  pula
  staa   $00,x
  dex
  decb
  bge    Fnc:Store_Perm_Results
  rts
```

Figure 4.1: PX-1000: The DES Permutations implementation

4.2 S-Box Substitution

Once the right (R_i) side of the input for each round is expanded to 48 bits and exclusive-OR'd with the Round Key (K_i), these 48 bits are split into 8 6-bit blocks. Each of these 6-bit blocks is fed through one of the eight S-Boxes in order to get a non-linear substitution [13].

Just as with the permutation, the PX-1000 uses a non-standard way to represent the tables for the S-Boxes that are needed by the DES encryption.

The PX-1000 combines the following S-Boxes: 1 and 2, 3 and 4, 5 and 6, 7 and 8. Each combination is represented by a vector of bytes. Where the left 4 bits of every byte correspond to the odd S-Box and the right 4 bits correspond to the even S-Box (Figure 4.3).

```
Sub:S_Box_Substitution:
  ldaa  #$C0  #Offset S-Box 7&8
  staa  S-Box_Offset
  ldaa  #$07
  clr   Null
  staa  Counter
loop:From_SBox7&8_to_SBox1&2:
  ldx   Null
  ldaa  $C2,x
  oraa  S-Box_Offset
  tab
  ldx   S-Box_Table
  abx
  ldaa  $00,x
  anda  #$0F
  staa  Temp
  dec   Counter
  ldx   Null
  ldaa  $C2,x
  oraa  S-Box_Offset
  tab
  ldx   S-Box_Table
  abx
  ldaa  $00,x
  anda  #$F0
  oraa  Temp
  psha
  ldaa  S-Box_Offset
  suba  #$40  #Decrease S-Box offset
  staa  S-Box_Offset
  dec   Counter
  bge   loop:From_SBox7&8_to_SBox1&2
  ldx   #$0000
loop:Store_Substitutions:
  pula
  staa  $CA,x
  inx
  cpx   #$0004
  bne   loop:Store_Substitutions
  rts
```

FIGURE 4.2: PX-1000: The S-Box substitution function in assembly (Address F690-F6D8).

PX-1000 S-Box 1 and 2			
0x E F	0x 3 9	0x 4 0	0x F 5
0x 0 3	0x A C	0x F D	0x 5 B
0x 4 1	0x A 7	0x 1 E	0x C 8
0x F D	0x 6 0	0x C 8	0x B 6
0x D 8	0x 6 2	0x E 7	0x 9 C
0x 7 4	0x C 1	0x 8 A	0x 3 7
0x 1 E	0x C D	0x 8 B	0x 7 6
0x 4 7	0x B A	0x 2 1	0x E C
0x 2 6	0x 5 C	0x D A	0x 3 9
0x E F	0x 9 6	0x 4 3	0x A 0
0x F B	0x 9 0	0x 6 4	0x A 3
0x 2 2	0x 5 9	0x 9 F	0x 0 5
0x B 3	0x 0 5	0x 2 D	0x 5 2
0x D 8	0x 3 B	0x 1 4	0x 6 E
0x 8 4	0x 7 A	0x B 1	0x 0 F
0x 1 E	0x 8 5	0x 7 2	0x D 9

S-Box 1 from DES specification																
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	9	7
1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

FIGURE 4.3: The match between the PX-1000 S-Boxes and the DES specification S-Boxes.

```
Sub:DES:
  ...
  # Initial permutation
  jsr   Sub:DES_Permutation

  # START OF THE 16 ROUNDS
loop:DES_Round_(16_times):

  # START OF FEISTEL FUNCTION
  # Expansion function
  jsr   Sub:DES_Permutation
  ...
loop:DES_Key_mixin:
  # Mix the key with expanded block
  ...
  bge   loop:DES_Key_mixin
  # S-Box substitution
\label{fig:16-rounds}
  jsr   Sub:S_Box_Substitution
  # Permutation
  jsr   Sub:DES_Permutation
  # END OF FEISTEL FUNCTION
  ...
loop:XOR_halves:
  ...
  # Merge both halves together
  bge   loop:XOR_halves
    ...
  bra   loop:DES_Round_(16_times)
    # END OF THE 16 ROUNDS

    # After the 16 rounds do the
    # final permutation
  Fnc:Final_Permutations:
    jmp   Sub:DES_Permutation
```

FIGURE 4.4: Snippets of code that make up the 16 DES rounds

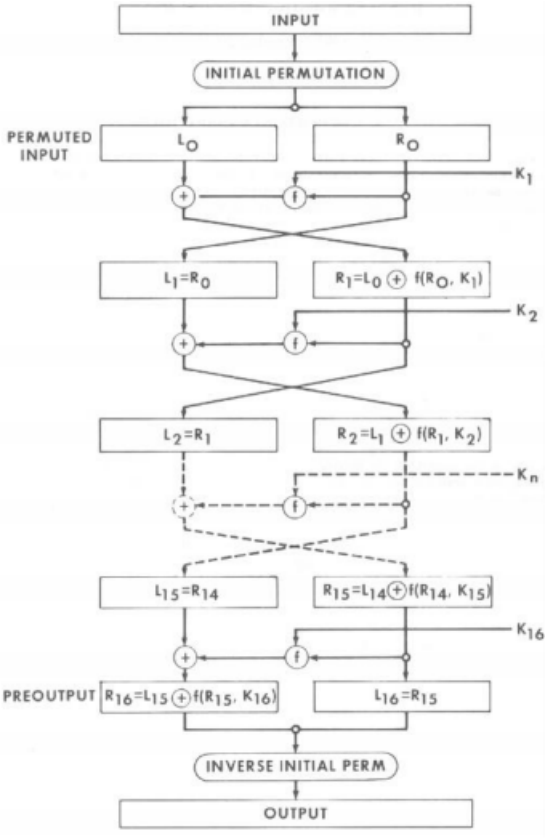

FIGURE 4.5: Schematic description of the DES rounds

res

4.3 16 identical rounds and the signature Feistel scheme

One of they key identifying features of DES are its 16 identical rounds. Each of those rounds consists of the following stages [13]:

- The Feistel function,
  - Expansion of (R_i) from 32 to 48 bits,
  - Key Mixing of the Round key K_i and the expanded R_i,
  - Substitution of the resulting 48 bits by use of the S-Boxes to produce 32 result bits,
  - Permutation of these 32 bits.
- Exclusive OR between thr result of the Feistel function and L_i.

See figure 4.4 for the overview of how this is implemented in the PX-1000.

4.4 Splitting of 64-bits of input into 2 32-bit halves

Before the 16 DES rounds, the key is split into two 32 bit halves. This part is trivial in assembly, because one can easily reference the lower and upper 4 bytes to get half the plaintext/cyphertext.

4.5 A key scheduling algorithm that generates the round keys from an initial 56-bit key.

Figure 4.8 shows the PX-1000 implementation of the DES key scheduling function. This function discards 8 parity bits of the initial 64 bit key and devised the rest into two 28 bit halves by using the PC-1 permutation [13].

By rotating these halves by one or two positions between each round and then combining them, all 16 round keys will be produced.

4.6 The DES Shift Key function

Figure 4.9 illustrates the shift-key function a single round in the DES key-schedule algorithm. This function takes the two 28-bit halfs of the 56 key bits selected by Permuted Choice 1 (PC-1). In each of these rounds, all bits are rotated left by one or two bits, depending on the current round.

FIGURE 4.6: PX-1000: DES Key Schedule.

```
Sub:DES_keygen:
  ldx   #$00D2
loop:Clear_Result_Location:
  clr   $00,x
  inx
  cpx   #$00D7
  bls   loop:Clear_Result_Location
  ldaa  #$FF
  staa  X0118
  staa  Permutation_Table_Index
  ldaa  #$0F
  staa  DES_Round_Counter
        # Permuted Choice 1
  jsr   Sub:DES_Permutation

for:round1_to_round16:
        # Shift 1 or 2 depending
        # on the round
  jsr   Sub:DES_Shift_Key
        # Permuted Choice 2
  jsr   Sub:DES_Permutation
        # Some administration
  ldaa  X0118
  adda  #$08
  staa  X0118
  staa  Temp
  ldaa  #$07
  staa  Counter
  clr   X00D2
  clr   X00D4
loop:Save_Result:
  ldx   X00D2
  ldaa  $CA,x
  ldx   X00D4
  staa  $42,x
  dec   Temp
  dec   Counter
  bge   loop:Save_Result
  dec   DES_Round_Counter
  bmi   break
        # Reset Permutation table
  ldaa  Permutation_Table_Index
  suba  #$09
  staa  Permutation_Table_Index
  jmp   for:round1_to_round16
break:
  rts
```

FIGURE 4.7: The PX-1000 implementation of the DES Key Schedule

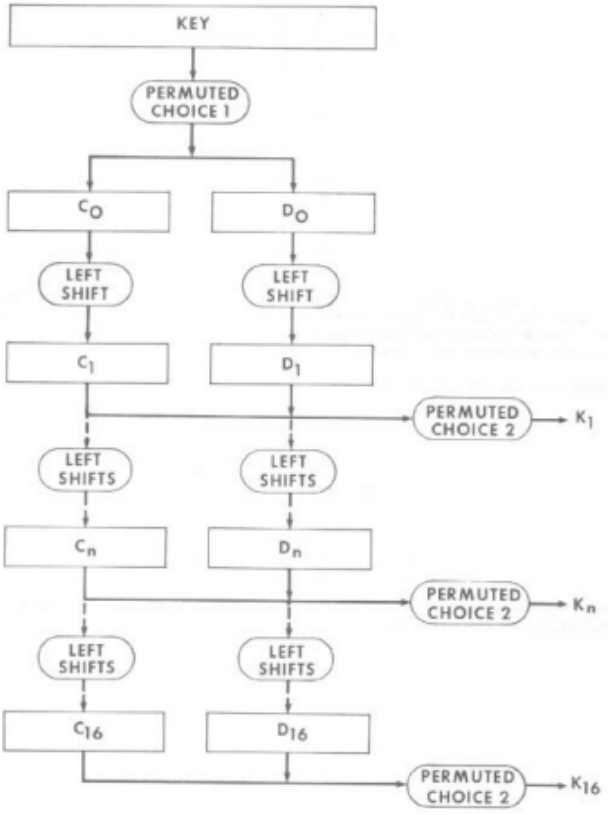

FIGURE 4.8: Schematic description of the DES key schedule

FIGURE 4.9: PX-1000: DES Shift Keys implementation.

```
Sub:DES_Shift_Key:
    ldx    DES_key_schedule_rotations
    ldab   desRoundCounter
    abx
    ldab   $00,x  # Loads the value
                  # for the current roun
Fnc:DES_Shift_Once:
    ldaa   LeftKey(Bit_4_1)
    asla
    rol    LeftKey(Bit_12_5)
    rol    LeftKey(Bit_20_13)
    rol    LeftKey(Bit_28_21)
    bcc    Fnc:No_Circular_shift
    oraa   #$10   # Set pos 1 to 1
Fnc:No_Circular_shift:
    staa   LeftKey(Bit_4_1)
    ldaa   RightKey(Bit_4_1)
    asla
    rol    RightKey(Bit_12_5)
    rol    RightKey(Bit_20_13)
    rol    RightKey(Bit_28_21)
    bcc    Fnc:No_Circular_shift
    oraa   #$10   # Set pos 1 to 1
Fnc:No_Circular_shift:
    staa   RightKey(Bit_4_1)
    decb
    bne    Fnc:DES_Shift_Once
    rts

DES_key_schedule_rotations:
    db    $01    #Round 16
    db    $02    #Round 15
    db    $02    #Round 14
    db    $02    #Round 13
    db    $02    #Round 12
    db    $02    #Round 11
    db    $02    #Round 10
    db    $01    #Round 9
    db    $02    #Round 8
    db    $02    #Round 7
    db    $02    #Round 6
    db    $02    #Round 5
    db    $02    #Round 4
    db    $02    #Round 3
    db    $01    #Round 2
    db    $01    #Round 1
```

(A) The DES Shift Keys function in assembly (Address F6D9-F719

Iteration Number	Number of Left 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

(B) The DES Shift table from the NIST DES standard

```
Key layout in memory:

Left Half Shift:
(The last 4 bits of C5 are ignored for
the 28 bit subkey)
   C2       C3       C4       C5
 ←←←←←←←←←←←←←←←←←←←←←←←←←←←←←←
xxxxxxxx xxxxxxxx xxxxxxxx xxxxZZZZ
↓                                 ↑
|---------------------------------|

Right Half Shift:
(The last 4 bits of C5 are ignored for
the 28 bit subkey)
   C6       C7       C8       C9
 ←←←←←←←←←←←←←←←←←←←←←←←←←←←←←←
xxxxxxxx xxxxxxxx xxxxxxxx xxxxZZZZ
|                                 ↑
|---------------------------------|
```

(C) The layout of the roundkey in memory

Chapter 5

Results of the PX-1000Cr analysis

The first indication that the PX-1000Cr does not implement DES is that it is not compatible with the older PX-1000 [1]. Also when looking at the disassembeled code, it shows that there are not enough non-code bytes that might be used to store the DES S-Boxes and permutation tables. (The original PX-1000 used more then 500 bytes for these tables [13].)

When reading section 4.2, please keep the following two things in mind:

- There is an abundant use of figures to explain the encryption scheme. In these figures, blue squares denote memory addresses, white squares denote operations and numbers in white squares represent literal hexadecimal values.
- The source for the PX-1000Cr is not yet fully reverse-engineered. So the findings in this chapter are representative of the current knowledge of this encryption scheme, but might be subject to change when more of the code is understood.

5.1 Algorithm analysis part 1: Block- or Stream- cipher

The assumption was that the revised algorithm would be similar to DES, but possibly weaker [4]. This assumption did not hold up when analyzing the code. We can see that there are a lot of operations done on the key, but only one on the input text. Specifically an X-OR (exclusive-or) operation between a key-byte and an input-byte. Therefore this algorithm implements a stream cipher. See figure 5.1.

FIGURE 5.1: PX-1000Cr: The final encryption step in the stream cipher

```
Fnc:FinalEncryptionStep:
    ldx    CurrentCharacterLocation
    ldab   $00,x
    eora   $00,x  # Acc a holds the current key-byte
    staa   $00,x
    inx
    stx    CurrentCharacterLocation
    cpx    EndOfStringLocation
    bhi    EndOfStringToEncrypt
    tst    EnCryptOrDecryptMode
    bmi    Fnc:ChangeSeedForPRNG
    tba
```

5.2 Intermezzo: What is a stream cipher?

A stream cipher is a cipher where a pseudo-random keystream is produced and combined with a plaintext. This keystream is usually generated by using a unique seed value (key) and a pseudo-random function that can generate an infinite stream of pseudo-random numbers [14].

The keystream and the plaintext are, in most cases, interpreted as binary strings that can be easily combined. This combination is done by using the reversible X-OR function. This approach can represent a security vulnerability when used incorrectly. Because of the reversible nature of the X-OR operation, the same key should not be reused. Consider the following example of what happens when a key is reused, where A and B are tho different messages and C is the corresponding keystream [15]:

$E(A) = A\ xor\ C$
$E(B) = B\ xor\ C$

If someone intercepts both $E(A)$ and $E(B)$ he can calculate the following:

$E(A)\ xor\ E(B) = (A\ xor\ C)\ xor\ (B\ xor\ C) = A\ xor\ B\ xor\ C\ xor\ C = A\ xor\ B$ [15]

When both A and B are representations of natural language, then retrieving both is relatively trivial and can be done with paper-and-pencil methods [16].

5.3 Algorithm analysis part 2: Key Scheduling

This stream cipher makes use of a 64 bits of key material. The key scheduling function (Figure 5.2 combines this key in a number of ways, and creates a 32-bit 'round-key' that will be used to encrypt the current character.

5.4 Algorithm analysis part 3: Pseudo-Random Number generator.

Stream ciphers need a pseudo-random number generator in order to generate their key-stream. The algorithm used in the PX-1000Cr makes use of 2 times 16 rounds that mix the key with a predefined vector that shifts one position for each iteration. See figure 5.4. Also, after each iteration, one of the used key-bytes, depending on the round will be updated. See figure 5.3.

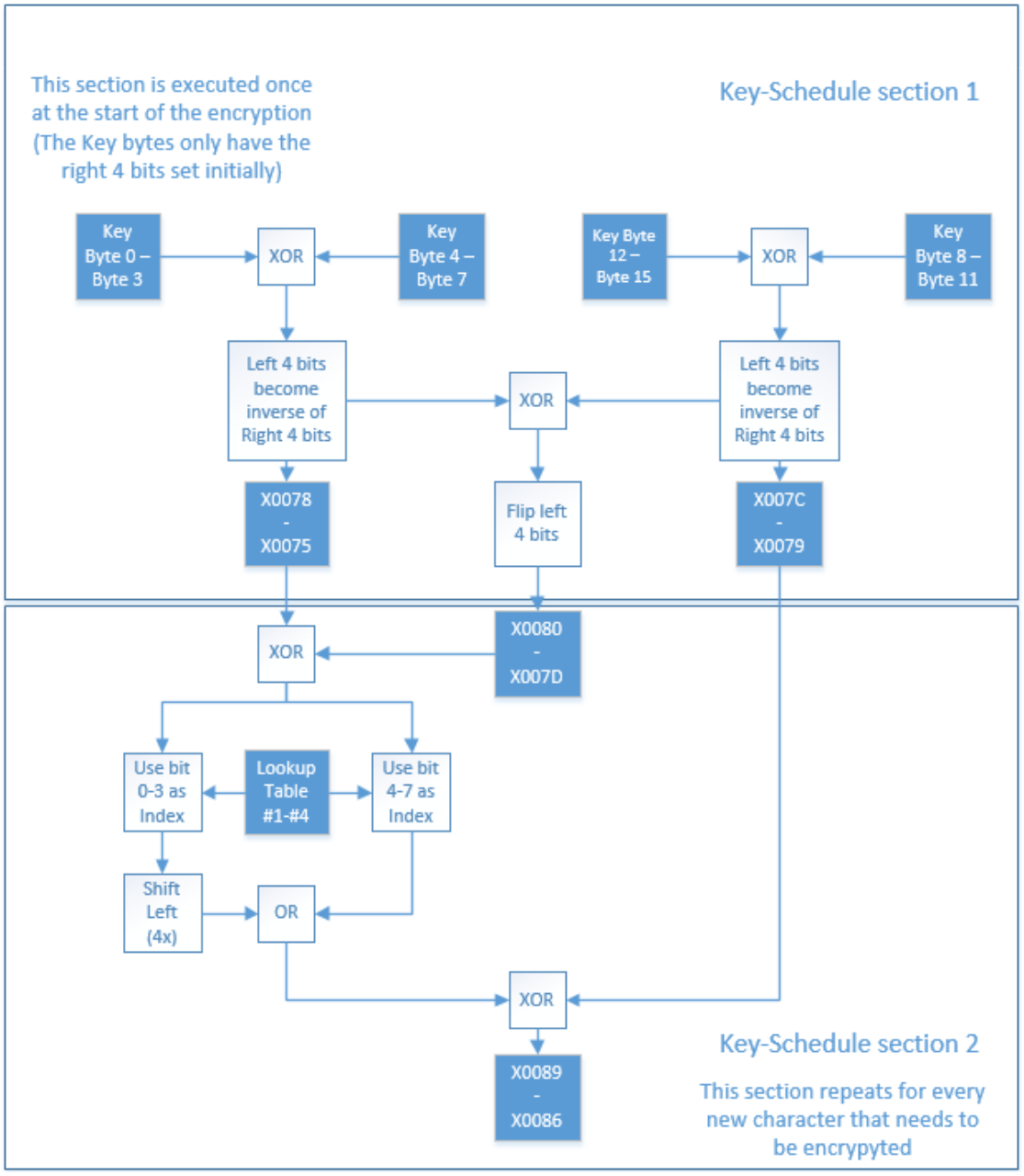

FIGURE 5.2: PX-1000Cr: Key schedule function.

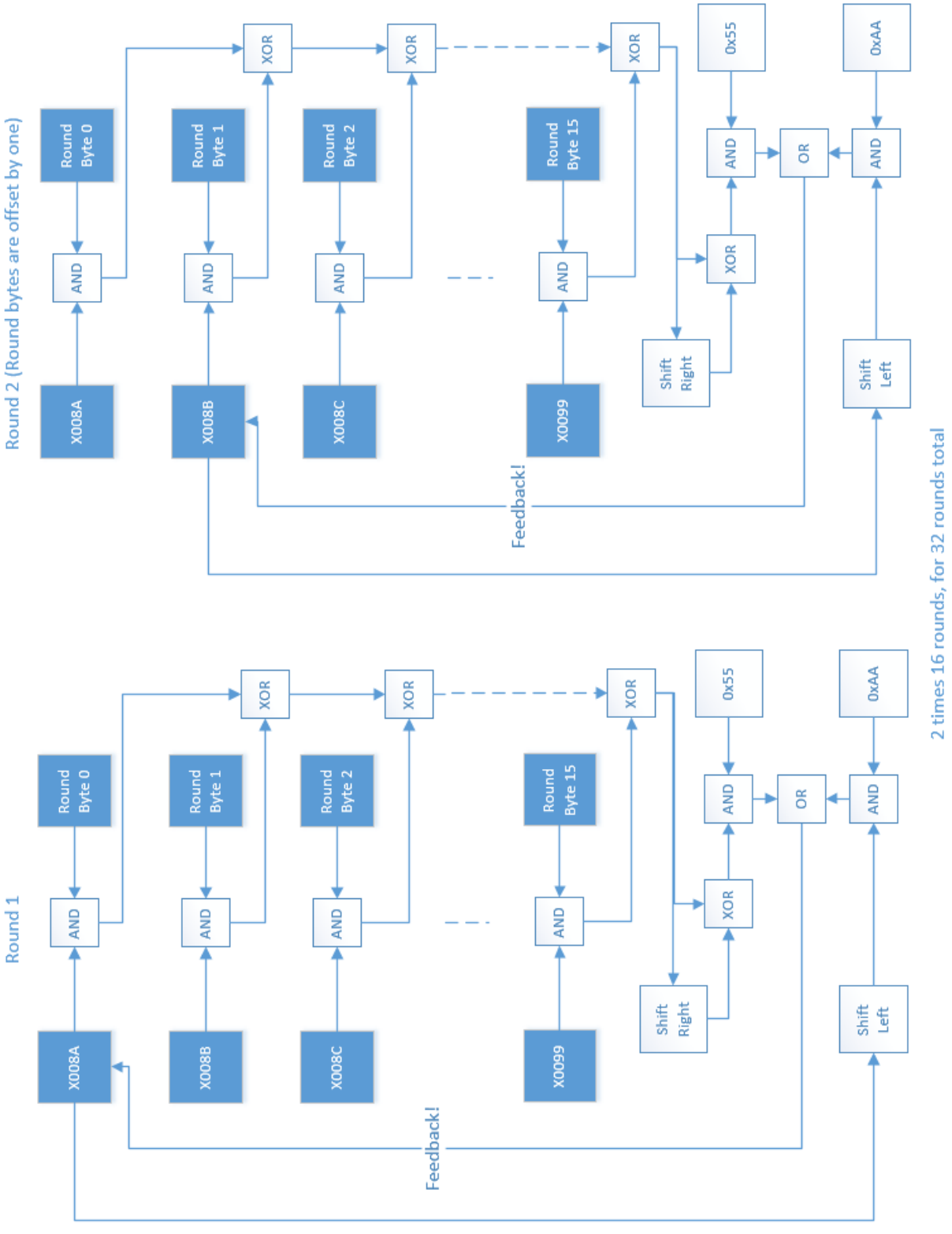

FIGURE 5.3: PX-1000Cr: Pseudo-Random Number Generator (2 Rounds out of 2x16 rounds are shown) X008A – X0099 Are initially key bytes 0-15 plus the inverse in the left 4 bits.

FIGURE 5.4: PX-1000Cr: The rotating vector of bytes to mix in the Psudo-Random Number Generator

06	0B	0A	78	0C	E0	29	7B	CF	C3	4B	2B	CC	82	60	80
80	06	0B	0A	78	0C	E0	29	7B	CF	C3	4B	2B	CC	82	60
60	80	06	0B	0A	78	0C	E0	29	7B	CF	C3	4B	2B	CC	82
82	60	80	06	0B	0A	78	0C	E0	29	7B	CF	C3	4B	2B	CC
CC	82	60	80	06	0B	0A	78	0C	E0	29	7B	CF	C3	4B	2B
2B	CC	82	60	80	06	0B	0A	78	0C	E0	29	7B	CF	C3	4B
4B	2B	CC	82	60	80	06	0B	0A	78	0C	E0	29	7B	CF	C3
C3	4B	2B	CC	82	60	80	06	0B	0A	78	0C	E0	29	7B	CF
CF	C3	4B	2B	CC	82	60	80	06	0B	0A	78	0C	E0	29	7B
7B	CF	C3	4B	2B	CC	82	60	80	06	0B	0A	78	0C	E0	29
29	7B	CF	C3	4B	2B	CC	82	60	80	06	0B	0A	78	0C	E0
E0	29	7B	CF	C3	4B	2B	CC	82	60	80	06	0B	0A	78	0C
0C	E0	29	7B	CF	C3	4B	2B	CC	82	60	80	06	0B	0A	78
78	0C	E0	29	7B	CF	C3	4B	2B	CC	82	60	80	06	0B	0A
0A	78	0C	E0	29	7B	CF	C3	4B	2B	CC	82	60	80	06	0B
0B	0A	78	0C	E0	29	7B	CF	C3	4B	2B	CC	82	60	80	06

5.5 Algorithm analysis part 4: Actual encryption

Before the actual encryption of the plaintext byte takes place, the result of the Pseudo-Random number generator and the round-key will be mixed. See figure 5.5.

After this, the resulting pseudo-random byte will be rotated a number of times equal to the current number of encrypted characters. Then this byte will be X-OR'd with the current plaintext byte and stored at the same location of that plaintext byte. See figure 5.6. Finally, the original round-key will be updated with the result byte of this encryption to assure that the next round uses a different seed. See figure 5.7.

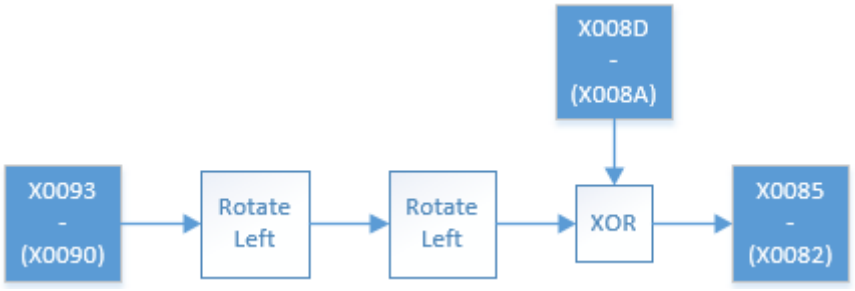

FIGURE 5.5: PX-1000Cr: Mix between the current round-key and the pseudo-random byte.

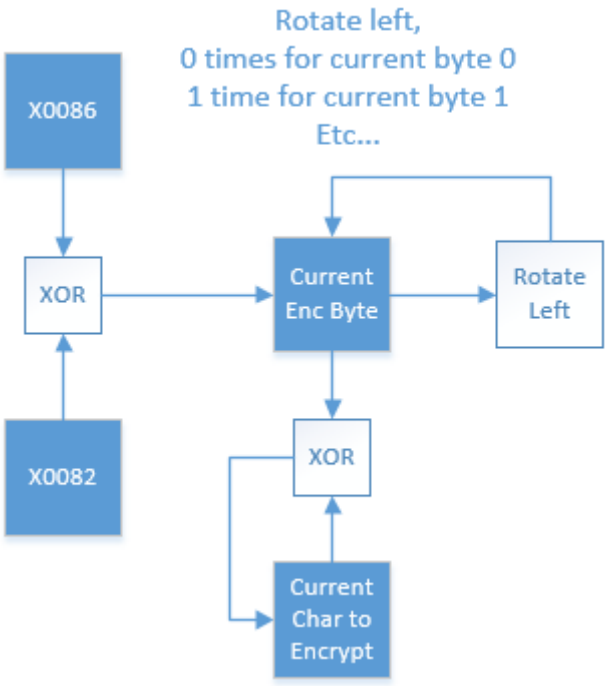

FIGURE 5.6: PX-1000Cr: Actual encryption of the plaintext byte.

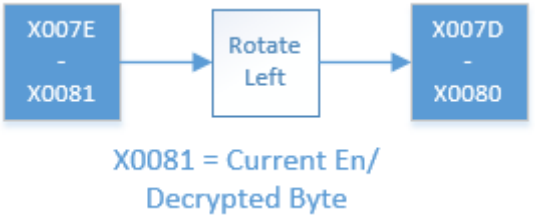

FIGURE 5.7: PX-1000Cr: Update of the PRNG seed.

5.6 Algorithm analysis part 5: The unknown

A small part of the encryption scheme is still unknown. There seems to be a section of code that has no purpose and where the result is discarded (Figure 5.8). This has one out of two reasons:

- The code is not correctly interpreted,
- There is an error in the implementation.

This section is repeated 8 times. The results are saved in accumulator a. Right thereafter, another value is loaded in this accumulator a without first saving this somewhere. Also, the value of X0086 gets rotated left 8 times, resulting in the original value.

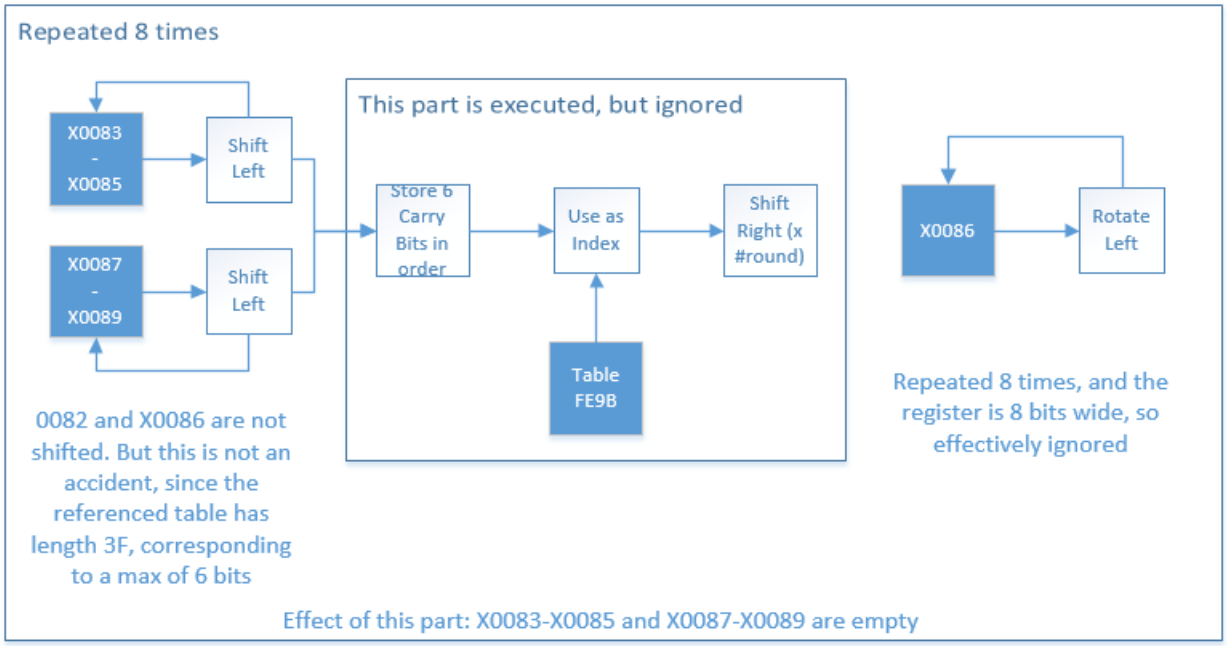

FIGURE 5.8: PX-1000Cr: Section with unknown use in the encryption scheme

Chapter 6

Discussion

6.1 Answers to research questions

This bachelor thesis shows that it is possible to reverse-engineer the crypto algorithms used in both versions of the PX-1000 by Philips. No countermeasures against reverse engineering were found. The deviations that were taken from the formal description of DES do not influence its security, but do aid the actual implementation.

We can now say that the original version of the PX-1000 does undoubtedly implement the DES algorithm, that was secure at the time of the PX-1000.

Also, it is shown that the revised crypto algorithm differs from DES. This confirms the suspicion that the NSA supplied another crypto algorithm. Even though we are not yet sure what this algorithm precisely is and if it is secure, the circumstantial evidence described in chapter 1 hints that this will be easier to break then DES. Therefore, further research is required.

6.2 Discussion about the method

The EPROM reader used by the crypto museum to extract the byte code was not meant to be used in combination with the EPROM in the PX-1000. By pretending that it was a related type of EPROM the reader could be tricked to read it anyway. This resulted in the program being read two or four times. Therefore is no guarantee that the read byte code is 100% correct. But it is very unlikely that in the case of an error this would result into readable code. Therefore we can assume that this is no problem.

The interpretation of the recovered disassembly is manual labor and there is the possibility of errors in the recovered code. One way to check this would be to create a program from the reverse engineered code that is equivalent. Then one can test this against a working PX-1000 and see if the results are the same.

6.3 Discussion about the PX-1000Cr encryption

The exact algorithm used in the PX-1000Cr is still unknown. When there is no existing algorithm that matches the one used in the PX-1000Cr, then it might a good idea to start looking for known vulnerabilities in stream ciphers of the same period.

6.4 Future Research

The encryption scheme used in the revised version is not yet fully recovered. It will be very interesting to completely reverse engineer and cryptanalyze this. Once this is done we can say with much more certainty if the PX-1000 was weakened (on purpose).

Appendix A

Initial listing for the PX-1000

```
; Symbol file for PX-1000 Original code (1983)
; ============================================
cpu  6303        ; Processor Type
org  0xE000      ; Origin address
numformat M      ; Number Format: Motorola

; Memory Map

; Internal Registers            0x0000-0x001F
; ----------------------------------------
symbol 0x0000 ioPort1dataDirection
symbol 0x0001 ioPort2dataDirection
symbol 0x0002 ioPort1data
symbol 0x0003 ioPort2data
symbol 0x0008 timerCSR
symbol 0x0009 CounterHigh
symbol 0x000A CounterLow
symbol 0x000B OutputCompareHigh
symbol 0x000C OutputCompareLow
symbol 0x000D InputCaptureHigh
symbol 0x000E InputCaptureLow
symbol 0x0010 SerialRateAndModeControlRegister
symbol 0x0011 SerialControlAndStatusRegister
symbol 0x0012 SerialReceiverDataRegister
symbol 0x0013 SerialTransmitDataRegister
symbol 0x0014 RWMemoryControlRegister
```

```
; External Memory Space          0x0020-0x007F
; ---------------------------------------

; Internal Ram                   0x0080-0x00FF
; ---------------------------------------
symbol 0x0080 RAM

; Main External Ram              0x0100-0x01FF
; ---------------------------------------

; I/O           0x4000-0x4001 & 0x8000-0x8001
; ---------------------------------------
symbol 0x4000 KeyboardCommands
symbol 0x4001 KeyboardData
symbol 0x8000 DisplayCommands
symbol 0x8001 DisplayData

; Unused
; ---------------------------------------
; 0x2000-0x3FFF
; 0x4002-0x7FFF
; 0x8002-0xDFFF

; EPROM / ROM                    0xE000-0xFFFF
; ---------------------------------------

vector 0xFFF0 sci_vector sci_entry
vector 0xFFF2 tof_vector tof_entry
vector 0xFFF4 ocf_vector ocf_entry
vector 0xFFF6 icf_vector icf_entry
vector 0xFFF8 irq_vector int_entry
vector 0xFFFA swi_vector swi_entry
vector 0xFFFC nmi_vector nmi_entry
vector 0xFFFE res_vector reset
```

Bibliography

[1] Marc Simons Paul Reuvers. Philips PX-1000, 2014. URL `http://www.cryptomuseum.com/crypto/philips/px1000/`.

[2] www.texttell.com WayBack Machine. Internet archive, showing the state of the Text Tell website in 2001, 1995. URL `http://web.archive.org/web/20010406040857/http://texttell.com/home.html`.

[3] TextLite. Technical manual PX-1000 PXP-40, 1980.

[4] Marcel Metze. Ingelijfd door de NSA. *De groene Amsterdammer*, 2014.

[5] Conny Braam. *Operatie Vula*. Uitgeverij Augustus, 2006. ISBN 9789045700465.

[6] Wikipedia. Nelson mandela, 2014. URL `http://nl.wikipedia.org/wiki/Nelson_Mandela`.

[7] TV program by NPS, VPRO. Andere Tijden, The Making of Nelson Mandela. 11 February, 2010.

[8] Randy Torrance and Dick James. The State-of-the-Art in IC Reverse Engineering. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems*, 2009.

[9] Richard H. Stern. Disassembling object code: A misdeed? *IEEE Micro*, 1994.

[10] Conquest Consultants. Dasmx - a microprocessor opcode disassembler, 1999. URL `http://16paws.com/ECU/DASMxx/`.

[11] Hitachi. HD6301 HD6303 Series Handbook, 1989.

[12] Andrew C. Staugaard Jr. *6801, 68701, & 6803 Microcomputer Programming & Interfacing*. 1981.

[13] U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology. FIPS Publication 46-3, Data Encryption Standard (DES), 1999. URL `http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf`.

[14] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, 1996. ISBN 978-0471117094.

[15] Wikipedia. Stream cipher attack, 2014. URL `http://en.wikipedia.org/wiki/Stream_cipher_attack`.

[16] Wikipedia. John Tiltman, 2014. URL `http://en.wikipedia.org/wiki/John_Tiltman`.