E.N.S.E.I.R.B.

Linux4Nano Project

Synthesis

CONCERNING IPOD NANO 2G AND CRYPTOGRAPHY

A Linux4Nano Team document

April 6th 2007

Abstract

In September 2006 Apple issued a new generation of iPod Nano, a portable multimedia player. A new firmware obfuscation mechanism has been introduced, preventing any other firmwares but Apple's one to be loaded on the iPod. This article presents all the accumulated knowledge about this obfuscation mechanism in an attempt to install other operating systems on this platform.

Contents

Introduction

On September 12th 2006, Apple issued the iPod Nano 2G. This second-generation, which is smaller than the first one, brings new features such as a brighter display, an upgraded battery life and new storage options (including 2, 4 and 8GB models).

In addition to those numerous features, Apple brings a new firmware obfuscation mechanism. This barrier prevents other firmwares from loading on the iPod. For instance, IN2G is currently not supported by iPodLinux project[1].

The aim of this document is to provide all the knowledge which have been found about the firmware encryption since the beginning of the project.

As an introduction, basic knowledge compulsory to archive experiences will be explained. After, the facts we have obtained up to now will be summarized. Then firmware encryption hypothesis which have been invalided will be focused. Finally, hypothesis we are currently assuming will be presented.

[1] `http://www.ipodlinux.org`

Chapter 1

Getting firmwares

First of all, to achieve experiences so as to obtain information about encryption, you need a/several version/s of iPod Nano 2G firmware. To obtain a firmware, you need a dump of the iPod. Two ways are offered to obtain it:

1. from Internet;
2. directly from the iPod.

1.1 Getting a dump from the Internet

Two versions of iPod dump are available on Internet. They can be found at following addresses:

- version `1.1.1`: `http://rapidshare.com/files/15681296/iPod_19.1.1.1.ipsw.html`
- version `1.1.2`: `http://www.felixbruns.de/iPod/firmware`

You will obtain a `*.ipsw` file. Rename it to a `*.zip` file, unzip it and you will get a file named `Firmware-w.x.y.z`. This file corresponds to `dump.img`. To extract the firmware from a dump, jump to section 1.3.

1.2 Getting a dump file from the iPod

To extract a dump from your iPod read following instructions.

1.2.1 Under GNU/Linux

1. plug the iPod to one usb port of your computer;
2. execute following command:

   ```
   $ dd if=/dev/sdX1 of=dump.img
   ```

   Change `/dev/sdX1` according to your configuration, `/dev/sdX` device file has to correspond to your iPod. Usually `X` has to be replaced by `a` or `b`;
3. after the execution of previous command, you will obtain a file named `dump.img`, it correspond to data dumped from your iPod.

When it is done, jump to the 1.3 section.

1.2.2 Under Windows

1. plug the iPod to one usb port of your computer;
2. goto `Start Menu>Execute`, type `cmd` and validate;
3. get the windows version of `dd` at `http://www.chrysocome.net/dd`;
4. then type the following command:

   ```
   > dd --list
   ```

5. search in `NT Block Device Objects` following sequence:

   ```
   \\?\Device\(XXXXXXXX)\Partition0
   link to \\?\Device\(XXXXXXXX)\(\ldots)
   Removable media other than floppy. Block size = 2048
   size is 4060086272 bytes
   ```

6. look for a size of 2Go, 4Go or 8Go depending on your Nano version;
7. note the `(XXXXXXX)` field (it looks like `Hardisk1` or similar);
8. type the following command:

   ```
   > dd if=\\?\Device\(XXXXXXX)\Partition0  of=dump.img bs=2048
       skip=63 count=11860 --progress
   ```

9. after the execution of previous command, you will obtain a file named `dump.img`, it corresponds to data dumped from your iPod.

When it is done, jump to the 1.3 section.

1.3 Extracting firmwares from the dump file

Three files have to be extracted from the dump:

1. `osos.fw`: ciphered Apple's main software which contains the operating system code;
2. `aupd.fw`: ciphered Flash ROM updater;
3. `rsrc.fw`: unencrypted iPod Nano file system. Apple uses it to store data needed for accessories such as *Nike iPod Sport Kit*.

To extract firmwares from dump file, follow the next instructions:

1. checkout the latest version of `extract2g` at the following address: `http://svn.gna.org/viewcvs/linux4nano/trunk/extract2g` and compile it using `make` command;
2. type the following command to obtain the list of available files:

   ```
   $ extract2g -l dump.img
   ```

3. using the command

   ```
   $ extract2g -A dump.img
   ```

   extract from `dump.img` following files:

- *osos.fw*;
- *aupd.fw*;
- *rsrc.fw*.

4. to obtain more information on `extract2g` software you can type the following command:

```
$ extract2g --help
```

Under Windows, replace `extract2g` by `extract2g.exe`.

1.4 Getting more dump files

Comparison between different firmware versions brings a lot of information. Thus, linux4nano team added an option to `extract2g` which make a hash sum of the firmware in argument. By making a hash sum of your firmware and send it to the development team, it can be easily known whether you have a firmware version which have not yet been identified. To obtain the hash sum, follow next instructions. Thanks to community tree versions have already been listed:

1. version `1.0.2`;
2. version `1.1.1`;
3. version `1.1.2`.

Instruction to compute hashes :

- check out the latest version of `extract2g` at the following address: `http://svn.gna.org/viewcvs/linux4nano/trunk/extract2g` and compile it using `make` command;
- to obtain firmware hash sum, use `-H` option as following:

```
$ extract2g -H dump.img
```

- the result will look like to:

```
./extract2g compiled at 14:23:05 Feb 3 2007.

osos: 0x******** rsrc: 0x******** aupd: 0x********
```

- just send it by mail to `sendyourfirmwares@linux4nano.org`.

Under Windows, replace `extract2g` by `extract2g.exe`.

Chapter 2

Facts about firmwares and encryption

This chapter presents all collected facts (no hypothesis).

2.1 Proof of encryption

On previous iPod version, `osos` and `aupd` files contain binary data. Thus, we can assume that extracted files still contained binary data. A quick overview on `osos` and `aupd` files shows that there are divised in three parts:

1. 512 bytes (the *header*);
2. 1536 bytes of zeros;
3. variable length data.

Using a disassembler such as *Objdump*[1] or *IDA pro*[2] on first and third parts, incoherent ARM9 code is gotten[3], thus it is not an unencrypted ARM9 binary.

The code obtained from the first part of `osos.fw` is:

```
 0: 00000000   andeq r0, r0, r0
 4: 00000002   andeq r0, r0, r2
 8: 00000001   andeq r0, r0, r1
 c: 00000040   andeq r0, r0, r0, asr #32
10: 00000000   andeq r0, r0, r0
14: 00615800   rsbeq r5, r1, r0, lsl #16
18: 20fd5db4   ldrcsh  r5, [sp], #212
1c: 9ab034c3   bls 0xfec0d330
20: 2add0a27   bcs 0xff7428c4
24: 1994dbe0   ldmneib r4, {r5, r6, r7, r8, r9, r11, r12, lr, pc}
28: 755037fb   ldrvcb  r3, [r0, -#2043]
2c: f7304faf   ldrnv r4, [r0, -pc, lsr #31]!
30: 3477d43f   ldrccbt sp, [r7], -#1087
34: 70a3a52b   adcvc r10, r3, r11, lsr #10
...
```

and the code from third part is:

[1] `http://www.gnu.org/software/binutils`

[2] `http://www.datarescue.com/idabase/index.htm`

[3] Instruction Set Quick reference Card: `http://www.arm.com/pdfs/QRC0001H_rvct_v2.1_arm.pdf`

```
800: 60844d84  addvs r4, r4, r4, lsl #27
804: d6588dfe  undefined
808: 485a875b  ldmmida r10, {r0, r1, r3, r4, r6, r8, r9, r10, pc}^
80c: b1fb2f08  mvnlts  r2, r8, lsl #30
810: c071dfd4  ldrgtsb sp, [r1], -#244
814: 5d9ecf03  ldcpl 15, cr12, [lr, #12]
818: a5d76325  ldrgeb  r6, [r7, #805]
81c: 609404a7  addvss  r0, r4, r7, lsr #9
820: 7576fdcb  ldrvcb  pc, [r6, -#3531]!
824: 56872792  undefined
828: 9edc6a78  mrcls 10, 6, r6, cr12, cr8, {3}
82c: 5851ae0f  ldmplda r1, {r0, r1, r2, r3, r9, r10, r11, sp, pc}^
830: 7cfe2050  ldcvcl  0, cr2, [lr], #320
 ...
```

This operation gave coherent code when it is performed on previous iPod generations. This clue asserts that firmwares are ciphered. Moreover, the `strings` command which returns character strings found in the input, returns on `osos.fw` no sense sentence (such as `>ZNj?Ah6qMB$l_DCF[`) whereas on previous unencrypted firmware it gives normal sentences (such as *Apple Computer, Inc.*).

2.2 Firmware structure

2.2.1 Overview

Figure 2.1 resumes simple firmware structure.

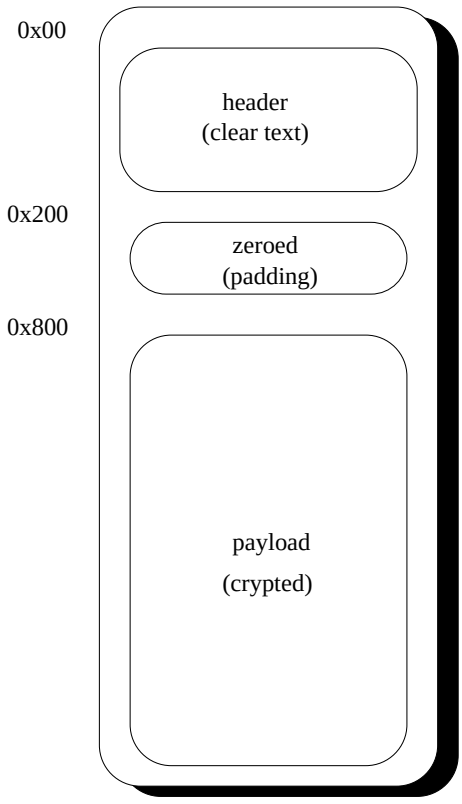

Figure 2.1: Simple firmware structure

2.2.2 Firmware sizes

Firmware version	Size
osos 1.0.2	0x5FB000
aupd 1.0.2	0x10D800
osos 1.1.1	0x615800
aupd 1.1.1	0x10D800
osos 1.1.2	0x5FA000
aupd 1.1.2	0x10E000

2.2.3 Header structure

We firstly compared `osos.fw` and `aupd.fw`. They begin with the same 5 bytes which exist in all known versions. We also found that the next field is the length of the data. This information is already presents in *directory entries*[4]. Then we looked for similarities and differences between each `osos` version. We could determine seven fields but without sense for the moment. Figure 2.2 describes the structure found. We found the same structure comparing each `aupd` version. The fact that bytes are *identical* means that they are identical between all `osos` versions and all `aupd` versions, however they are different comparing `osos` and `aupd`.

Relative address	Values
0x0	00 00 00 00
0x4	02 00 00 00
0x8	01 00 00 00
0xc	40 00 00 00
0x10	00 00 00 00
0x14	length of file data in little-endian
0x18	4 unknown identical bytes
0x1C	20 unknown different bytes
0x30	420 unknown identical bytes
0x1D4	20 unknown different bytes
0x1E8	24 unknown identical bytes
0x200	header end

Figure 2.2: `osos` header structure description

2.2.4 Data structure

There is no header like structure in data part. Regarding figures 2.3 and 2.4, comparison between available firmwares give us a lot of information.

Version	1.0.2	1.1.1	1.1.2
1.0.2	-	0	0
1.1.1	0	-	1680 bytes (0x800 -> 0xE90)
1.1.2	0	1680 bytes (0x800 -> 0xE90)	-

Figure 2.3: Number of identical bytes at the beginning of `osos` data part

[4]For more details, read the hardware synthesis avaible at `http://www.linux4nano.org/drupal/files/hardware_synth.pdf`

Version	1.0.2	1.1.1	1.1.2
1.0.2	-	240 (0x800 -> 0x8F0)	32 (0x800 -> 0x820)
1.1.1	240 (0x800 -> 0x8F0)	-	32 (0x800 -> 0x820)
1.1.2	32 (0x800 -> 0x820)	32 (0x800 -> 0x820)	-

Figure 2.4: Number of identical bytes at the beginning of `aupd` data part

2.3 Entropy

The entropy is a measure of disorder and correlation between data. The entropy $\mathcal{E}$ is computed thanks to following formula:

$$\mathcal{E} = -\sum_{k=0}^{255} \frac{c_k}{n} \log_2 \left(\frac{c_k}{n} \right)$$

where k corresponds to a character, c_k corresponds to the number of character k occurrences and $n = \sum_{k=0}^{255} c_k$.

A high entropy (8 is the maximum) means that file looks like random files. Figure 2.5 gives entropy values of `osos` and `aupd`.

Firmware	Value
osos 1.0.2	7.999958
aupd 1.0.2	7.999459
osos 1.1.1	7.999959
aupd 1.1.1	7.999497
osos 1.1.2	7.999964
aupd 1.1.2	7.999496

Figure 2.5: `osos` and `aupd` entropies

Chapter 3

Old hypothesis about ciphered data cryptanalysis

First, simple hypothesis were supposed. All the following hypothesis are based on statistical hypothesis. The aim of this chapter is to validate/invalidate the use of "basic" encryption algorithms by Apple to protect its firmware.

3.1 Monoalphabetic substitution

A first algorithm considered is monoalphabetic substitution. Each character (ASCII) corresponds to an other. For instance, adding 13 modulo 256 is a possible encryption function. Any substitution on the ASCII characters is possible.

This type of algorithm keeps one of the original text's property : the frequency of each character is the signature of a text. A simple program (available on the svn of our project[1]) was made to analyze the dumped files.

Unfortunately, according to figure 3.1, each ASCII character frequency is 1/256, that means all characters appear in the text as many time as others.

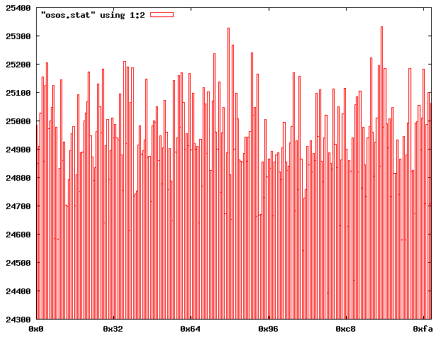

Figure 3.1: `osos` character frequencies

This kind of repetition means that algorithm used is more complex than a simple monoalphabetic substitution.

[1] `http://svn.gna.org/viewcvs/linux4nano/trunk/stats/`

3.2 Polyalphabetic substitution

The first assertion has been invalidated. Using several different substitutions could basically give a balanced repartition. Thus, this hypothesis can be assumed. It can be supposed substitution number is a multiple of 4 (which represents ARM9 instruction length).

To assert this hypothesis, the previous program was modified in order to analyze the first bytes of each "instruction", corresponding to the opcode.

Again, according to figure 3.2, each ascii character frequency is 1/256. Furthermore, comparing with an analysis done on a ARM binary version of `ls` software (cf figure 3.3), frequencies are really different. Thus polyalphabetic substitution hypothesis has also been invalidated.

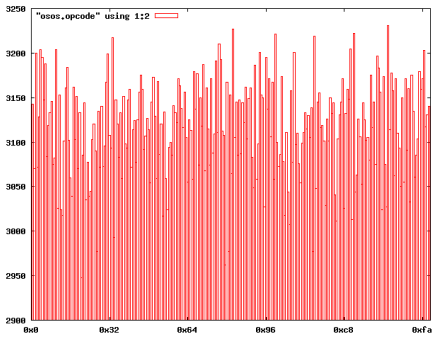

Figure 3.2: Statistics on `osos` opcodes

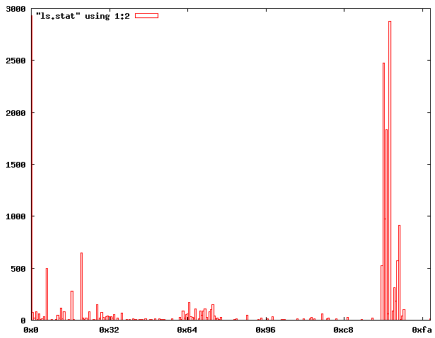

Figure 3.3: Statistic on `ARM` opcode (considering `ls` ARM binary)

3.3 Compression

An other hypothesis which had been supposed is: the encryption algorithm is a compression algorithm. Following facts rejecting this hypothesis:

- firmware size does not seem to have been reduced;
- an important time is needed to decompress it;
- ciphered file entropy is too high to correspond to a compression algorithm.

An compression, added to an other encryption algorithm, can be considered as a valid hypothesis, however the time needed to obtain the unencrypted file will increase. Therefore, it can be assume that algorithm type is not used.

3.4 Substitution by 4 byte blocks

Another hypothesis which can be made is the following: files are encrypted using substitution by blocks multiple of 4. A program which reads files 4 bytes by 4 bytes, was designed. No repetitions could be found thus, this assertion is false.

3.5 Pattern research

It can be supposed that the length of the blocks is not a multiple of 4. However, finding repetitions of any length in a text requires a lot of memory.

fv program found at following address`http://www.fantascienza.net/leonardo/ar/string_repetition_statistics/string_repetition_statistics.html` does repetition statistics using hash techniques. It can be seen, on the figure 3.4, that files look like pure random text.

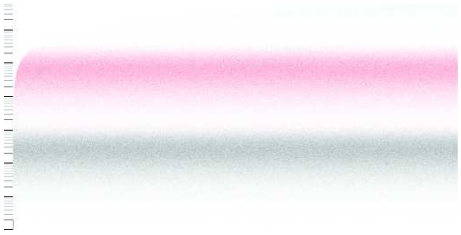

Figure 3.4: Pattern repetition statistics

Finally, it seems that iPod encryption algorithm is a stronger algorithm (such as DES, AES...). High entropy highlighted in section 2.3 confirms this assumption.

Chapter 4

Current prospects

4.1 Encryption algorithm class: stream cipher

4.1.1 Definition

According to Wikipedia [1]:

> " In cryptography, a stream cipher is a symmetric cipher in which the plaintext digits are encrypted one at a time, and in which the transformation of successive digits varies during the encryption. An alternative name is a state cipher, as the encryption of each digit is dependent on the current state. In practice, the digits are typically single bits or bytes.
>
> Stream ciphers represent a different approach to symmetric encryption from block ciphers. Block ciphers operate on large blocks of digits with a fixed, unvarying transformation. This distinction is not always clear-cut: in some modes of operation, a block cipher primitive is used in such a way that it acts effectively as a stream cipher. Stream ciphers typically execute at a higher speed than block ciphers and have lower hardware complexity. However, stream ciphers can be susceptible to serious security problems if used incorrectly: see stream cipher attacks — in particular, the same starting state must never be used twice."

4.1.2 Arguments

In previous versions of iPod (since mini iPods), `aupd` file was encrypted. We know that iPod 5G `aupd` file is encrypted thanks to RC4 algorithm. You can find $\mathcal{C}$ implementation of decryption algorithms in chapter 4.2.

According to chapter 3, several simple encryption methods were invalidated. Furthermore, *a priori*, there is no reason for Apple to choose another encryption class of algorithms. Another argument in favor to use a stream cipher algorithm is that it is easy to implement and it is fast (linear complexity regarding input length).

Comparing `osos` version 1.1.1 and 1.1.2 (cf figure 2.3), the first 1680 bytes of data part (cf subsection 2.2.4) are the same. With a stream cipher encryption method, the modification of a bit has repercussions on all following bits. Assuming that there has been no major correction between two versions, the 1681st can be the first different byte. Furthermore, data part of `osos` version 1.0.2 has no similar bytes with two other versions. If we assume that there have been major corrections (notably, the first byte is different), this hypothesis is coherent. Comparison between `aupd` versions does not invalidate this hypothesis.

[1] `http://en.wikipedia.org/wiki/Stream_cipher`

A key is used by stream cipher to make an initialization vector (IV [2]). The IV is a high entropy block of bits that is required in first encryption step. To decrypt a stream cipher encrypted file, we need to find the key.

4.2 Key emplacements: header and/or bootloader

It is improbable for the key to be fully located in read only memory. Indeed, if someone finds the key (using for instance a *brutforce attack*, Apple cannot protect anymore (without make important modifications) the firmware.

In a previous version (iPod 5G), a part of the key was hidden in the header. The key was computed by an algorithm located in the bootloader using the header (read-write memory) and a constant which is located in the bootloader (*a priori* read-only memory). We can assume that Apple keeps a similar method for iPod Nano 2G.

For each firmware versions (of iPod Nano 2G), there is no similarity between `osos` and `aupd` files. Thus, we can assume that there are two keys, one to encrypt `osos` and another to encrypt `aupd`. Regarding similarities between `osos` versions and similarities between `aupd` versions in data part beginning, it can be supposed that since version 1.0.2 keys have not been changed. As Apple knows that the key has no been found yet, Apple has no reason to change it.

According to section 2.2.3, `osos` and `aupd` headers are very different, however, each version of `osos` headers have a lot of similar field and its the same for `aupd` headers. This is an other argument in favor with the hypothesis that a part of the key is hidden in the header.

[2] `http://en.wikipedia.org/wiki/Initialization_vector`

References

In this chapter, some useful online references can be found. They all provide some piece of information likely to be used in order to break the encryption protocol applied on the firmwares, or in order to use some flaws to avoid the confrontation with the encryption.

Books

- Cryptanalysis a Study of Ciphers and Their Solutions, Helen Fouche Gaines, *Dover Publications*
- Handbook of Applied Cryptography, Alfred J. Menezes, *Crc*
- Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition, Bruce Schneier, *Wiley*
- Practical Cryptography, Niels Ferguson, *Wiley*

Online Books

- `http://osxbook.com/` an online book which provide a lot of information about the MACOSX system and its binary obfuscation
- `http://osxbook.com/book/bonus/chapter7/binaryprotection/index.html` Understanding Apple's Binary Protection in Mac OS X chapter
- `http://apple.slashdot.org/article.pl?sid=06/10/30/2147231`) How Encrypted Binaries Work In Mac OS X discussion forum
- `http://en.wikibooks.org/wiki/Reverse_Engineering` a Wikibooks very complete article exploring numerous aspects of reverse engineering. It can provide some details that might have been forgotten
- `http://en.wikibooks.org/wiki/Reverse_Engineering/File_Formats`

Some firmware reverse engineering

- `http://www.minidisc.org/pony_mp3n1_interview.html` An interview with Sasha Breginski, head of Pony Engineering, Moscow, on the topic of his team's reverse engineering of the Sony MZ-N1
- Hacking the Xbox: An Introduction to Reverse Engineering, Andrew Huang, Xenatera Press
- `http://www.rockbox.org/twiki/bin/view/Main/TargetStatus` a website that presents a very vast project. This team develops a free opensource multiplatform firmware. It has been adapted to many platforms, such as iPod Nano 1G for instance. Some clues may be found there.

Source code

4.3 RC4

4.3.1 rc4.h

```
/* adapted from http://www.cypherspace.org/adam/rsa/rc4.c */

typedef struct rc4_key
{
   unsigned char state[256];
   unsigned char x;
   unsigned char y;
} rc4_key;


void prepare_key(unsigned char *key_data_ptr, int key_data_len, rc4_key *key);

void rc4(unsigned char *buffer_ptr, int buffer_len, rc4_key *key);
```

4.3.2 rc4.c

```
/* adapted from http://www.cypherspace.org/adam/rsa/rc4.c */

#include <stdio.h>
#include "rc4.h"


#define swap_byte(x,y) t = *(x); *(x) = *(y); *(y) = t

void prepare_key(unsigned char *key_data_ptr, int key_data_len, rc4_key *key)
{
  unsigned char t;
  unsigned char index1;
  unsigned char index2;
  unsigned char* state;
  short counter;

  state = &key->state[0];
  for(counter = 0; counter < 256; counter++)
  state[counter] = counter;
  key->x = 0;
  key->y = 0;
  index1 = 0;
  index2 = 0;
  for(counter = 0; counter < 256; counter++)
  {
    index2 = (key_data_ptr[index1] + state[counter] + index2) % 256;
```

```
    swap_byte(&state[counter], &state[index2]);
    index1 = (index1 + 1) % key_data_len;
  }
}

void rc4(unsigned char *buffer_ptr, int buffer_len, rc4_key *key)
{
  unsigned char t;
  unsigned char x;
  unsigned char y;
  unsigned char* state;
  unsigned char xorIndex;
  short counter;

  x = key->x;
  y = key->y;
  state = &key->state[0];
  for(counter = 0; counter < buffer_len; counter++)
  {
    x = (x + 1) % 256;
    y = (state[x] + y) % 256;
    swap_byte(&state[x], &state[y]);
    xorIndex = (state[x] + state[y]) % 256;
    buffer_ptr[counter] ^= state[xorIndex];
  }
  key->x = x;
  key->y = y;
}
```

4.4 decrypt.c

```
/* Adapted from Franco Zavatti <badblox@doramail.com> code */

#include "rc4.h"
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

/* convertion big/little endian */
unsigned int
readword(unsigned char * data,unsigned int pos)
{

  return (data[pos])+(data[pos+1]<<8)+(data[pos+2]<<16)+(data[pos+3]<<24);
}

/* test if the marker is enable */
char
testmarker(unsigned int marker)
{
  unsigned int mask;
  unsigned int decrypt;
  unsigned int temp1,temp2;

  mask = (marker&0xff)|((marker&0xff)<<8)|((marker&0xff)<<16)|((marker&0xff)<<24);
  decrypt = marker ^ mask;
  temp1=decrypt>>24;
  temp2=decrypt<<8;
```

```
  if (temp1==0){
    return 0; /* false */
  }
  temp2=temp2>>24;
  decrypt=decrypt<<16;
  decrypt=decrypt>>24;
  if ((temp1<temp2)&&(temp2<decrypt)){
    temp1=temp1&0xf;
    temp2=temp2&0xf;
    decrypt=decrypt&0xf;
    if ((temp1>temp2)&&(temp2>decrypt)){
      if (decrypt!=0) return 1; // This marker is enable!
    }
  }

  return 0;
}



int
main(int argc, char ** argv)
{

  FILE * fdin, * fdout;
  unsigned char header[512];
  int i,j;
  unsigned int key = 0,pos = 0;
  unsigned int offset[]={0x5,0x25,0x6f,0x69,0x15,0x4d,0x40,0x34};
  unsigned int amarker,word,temp1;
  unsigned int constant = 0x54c3a298;
  unsigned int r1,r2,r12,r14;
  unsigned char cle[4];
  unsigned char buff[4];
  rc4_key rc4k;
  char isencrypted = 0;

  if (!(fdin = fopen(argv[1], "r"))) {
    perror("open");
    return 1;
  }


  /* Read the first 512 bytes of the file */

  for (i=0;i<512;i++) {
    header[i]=getc(fdin);
  }


  /* Compute the key */

  for (i=0;i<8;i++) {
    pos=offset[i]*4;
    amarker = readword(header,pos);
    printf("Marker %d : %x\n",i+1,amarker);
    if (testmarker(amarker)){
      isencrypted = 1;
```

```
      printf("Marker %d is set\n", i+1);
      pos =(offset[i+1]*4)+4;
      for (j=0;j<2;j++) {
        word=readword(header,pos);
        temp1=amarker;
        temp1^=word;
        temp1^=constant;
        key=temp1;
        pos+=4;
      }
      r1=0x6f;
      for (j=2;j<128;j+=2){
        r2=readword(header,j*4);
        r12=readword(header,(j+1)*4);
        r14=r2|(r12>>16);
        r2&=0xffff;
        r2|=r12;
        r1^=r14;
        r1=r1+r2;
      }
      key^=r1;
      printf("Associated key is %x\n",key);
    }
  }

  if(!isencrypted){
    fclose(fdin);
    printf("The input file is uncrypted\n");
    return EXIT_SUCCESS;
  }

  if (!(fdout = fopen(argv[2], "w+"))) {
    perror("open");
    return EXIT_FAILURE;
  }

  cle[3]=(key&0xff000000)>>24;
  cle[2]=(key&0xff0000)>>16;
  cle[1]=(key&0xff00)>>8;
  cle[0]=(key&0xff);


  /* Prepare Key */
  prepare_key(cle,4,&rc4k);

  /*Decryption loop*/
  while (!feof(fdin))
  {

    buff[0]=fgetc(fdin);
    buff[1]=fgetc(fdin);
    buff[2]=fgetc(fdin);
    buff[3]=fgetc(fdin);
    rc4(buff,4,&rc4k);
    fputc(buff[0],fdout);
    fputc(buff[1],fdout);
    fputc(buff[2],fdout);
    fputc(buff[3],fdout);
```

```
  }

  fclose(fdin);
  fclose(fdout);
  return EXIT_SUCCESS;
}
```