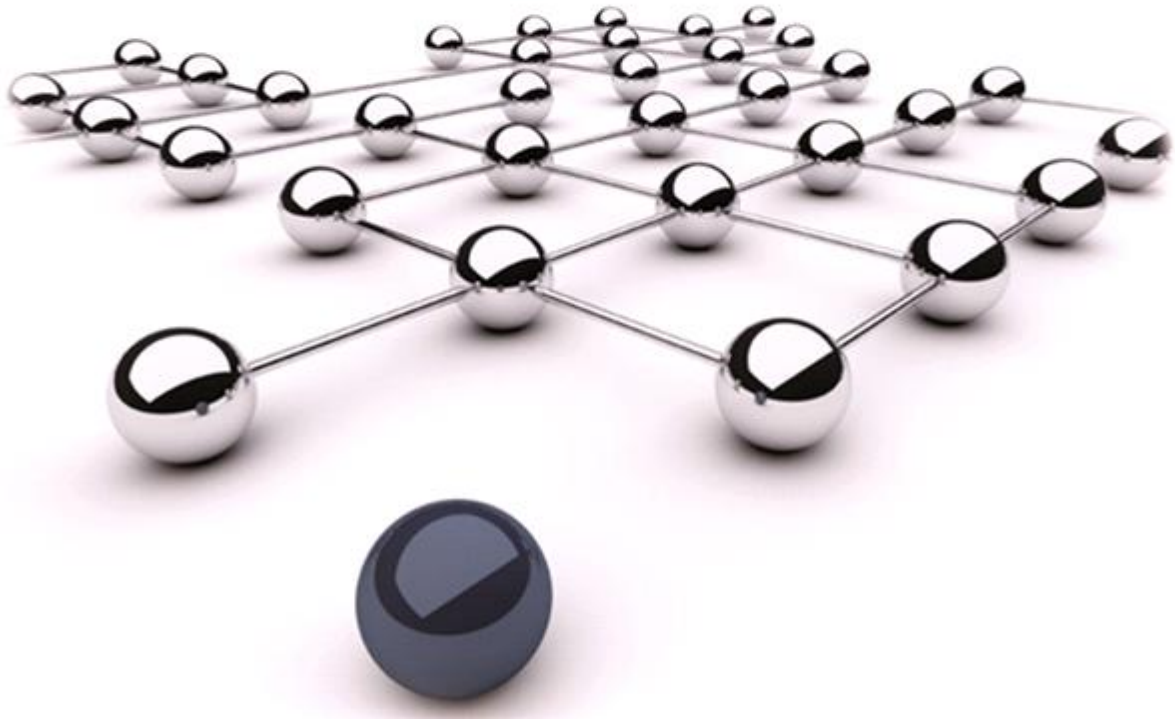
OPPIDA

EXPERT EN SÉCURITÉ
DES SYSTÈMES D'INFORMATION



Challenge SSTIC 2014

Présentation de la solution

REFERENCE : OPPIDA/DOC/2014/SSTIC/694/1.0

www.oppida.fr

6 avenue du Vieil Etang • Bât B • 78180 Montigny le Bretonneux

Tél. : +33 (0)1 30 14 19 00 • Fax : +33 (0)1 30 14 19 09 • Mail : contact@oppida.fr

SARL au capital de 270 000€ • RCS : Versailles B419 296 090 • SIRET : 419 296 090 00030 • Code APE : 6202A

ETAT DES VALIDATIONS

	Rédacteur	Vérificateur	Approbateur
Signé par	Damien Millescamps	Luc Abric	Valérian Perret
Fonction	Consultant	Expert	Directeur des Opérations
Date	02/05/2014	02/05/2014	02/05/2014

SUIVI DE VERSION

Version	Date	Nature des modifications	Page et section
1.0	02/05/2014	Création du document	Toutes

DIFFUSION

Entité	Nom	Coordonnées
SSTIC	Guillaume Delugré	66a65dc050ec0c84cf1dd5b3bbb75c8c@challenge.sstic.org

SOMMAIRE

1 INTRODUCTION

1.1 Contexte

Le Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC) est une conférence francophone organisée annuellement à Rennes. Depuis plus de 10 ans, différents acteurs du monde de la sécurité des systèmes d'information s'y retrouvent afin d'échanger leurs points de vue techniques et scientifiques.

La coutume veut que chaque édition du SSTIC propose un challenge technique accessible librement pour tous. La version 2014 est proposée par Guillaume Delugré (Quarkslab) et peut être trouvée sur:

```
http://communaute.sstic.org/ChallengeSSTIC2014
```

L'objectif est de retrouver une adresse e-mail sous la forme ...@challenge.sstic.org.

1.2 Objectifs du document

Ce document présente une solution à ce challenge pour l'année 2014.

1.3 Périmètre

Le périmètre se limite au fichier *usbtrace.xz* fourni et aux données qu'il contient.

1.4 Références

Article L122-6-1 du code de la Propriété Intellectuelle

III. La personne ayant le droit d'utiliser le logiciel peut sans l'autorisation de l'auteur observer, étudier ou tester le fonctionnement ou la sécurité de ce logiciel afin de déterminer les idées et principes qui sont à la base de n'importe quel élément du logiciel lorsqu'elle effectue toute opération de chargement, d'affichage, d'exécution, de transmission ou de stockage du logiciel qu'elle est en droit d'effectuer.

2 DECOUVERTE

2.1 Analyse de la trace USB

Le challenge commence par une trace USB à analyser:

```
MD5: 3783cd32d09bda669c189f3f874794bf - usbtrace.xz
```

Après extraction, on reconnaît rapidement la sortie de **usbmon**. En regardant le contenu échangé, on peut reconnaître une session *Android Debug Bridge (ADB)* grâce aux commandes *WRTE* / *OKAY* / *SYNC*.

Lors de la session, des répertoires sont synchronisés, on notera particulièrement:

```
sync /sdcard/Documents/
SCSW-2014-Hacking-9.11_uncensored.pdf
NATO_Cosmic_Top_Secret.gpg

sync /data/local/tmp
/data/local/tmp/badbios.bin
```

Seul le fichier *badbios.bin* est mis à jour et se trouve dans la trace.

2.1.1 Le protocole ADB

La documentation du protocole ADB peut se trouver ici:

https://android.googlesource.com/platform/system/core/+/master/adb/protocol.txt

La partie de l'échange qui nous intéresse est la requête *SEND* de la commande *SYNC*:

```
00000820  00 00 4b 01 00 00 a8 ad  ab ba 53 45 4e 44 21 00  |..K.......SEND!.|
00000830  00 00 57 52 54 45 07 02  00 00 00 01 00 00 00 10  |..WRTE..........|
00000840  00 00 2b f3 01 00 a8 ad  ab ba 2f 64 61 74 61 2f  |..+......./data/|
00000850  6c 6f 63 61 6c 2f 74 6d  70 2f 62 61 64 62 69 6f  |local/tmp/badbio|
00000860  73 2e 62 69 6e 2c 33 33  32 36 31 44 41 54 41 00  |s.bin,33261DATA.|
00000870  00 01 00 7f 45 4c 46 02  01 01 00 00 00 00 00 00  |....ELF.........|
[...]
00013c20  00 00 00 44 4f 4e 45 6e  b4 4f 53 4f 4b 41 59 07  |...DONEn.OSOKAY.|
```

Le fichier est transféré par blocs sans identifiants de 65536 octets maximum commençant par *DATA* et dont la taille est codée sur 32 bits, eux même transférés par blocs de 4096 octets maximum par commandes *WRTE* suite à un *SEND* qui prend en second paramètre les permissions du fichier en octal converties en décimal et codées en *ASCII*. Un protocole simple et efficace…

2.1.2 Récupération du binaire

Nous savons donc qu'un fichier du nom de **badbios.bin** avec des permissions **-rwxr-xr-x** est copié sur le téléphone.

Afin de l'extraire, une possibilité consiste à récupérer les données des trames *WRTE* concernant ce fichier. Il y a 19 blocs de 4096 octets et un bloc final de 233 octets.

Après concaténation du résultat, il faut maintenant retirer la surcharge de la requête *SEND*. Les paramètres de la commande sont les suivants:

```
/data/local/tmp/badbios.bin,33261
```

Ils représentent 33 octets et sont à retirer au début, ainsi que les 8 octets du premier *DATA*, puis les 8 octets du *DATA* après le sous-bloc de 65536 octets et finalement les 8 octets du *DONE* à la fin.

L'extraction du binaire peut se faire avec une "simple" ligne de commande :

```
$ egrep '(4096|233) =' usbtrace | cut -d'=' -f2 | xxd -r -p | (dd of=badbios.bin bs=1
skip=$((0x29)) count=65536 && dd of=badbios.bin conv=notrunc oflag=append bs=1 skip=8
count=$((4096*19+233-8*2-0x29-65536))) 2>/dev/null
```

Nous pouvons maintenant inspecter le résultat :

```
$ file badbios.bin
badbios.bin: ELF 64-bits LSB executable, version 1 (SYSV), statically linked, stripped
$ readelf -h badbios.bin
  ELF Header:
    Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
    Class:                             ELF64
    Data:                              2's complement, little endian
    Version:                           1 (current)
    OS/ABI:                            UNIX - System V
    ABI Version:                       0
    Type:                              EXEC (Executable file)
    Machine:                           AArch64
    Version:                           0x1
    Entry point address:               0x102cc
    Start of program headers:          64 (bytes into file)
    Start of section headers:          77680 (bytes into file)
    Flags:                             0x0
    Size of this header:               64 (bytes)
    Size of program headers:           56 (bytes)
    Number of program headers:         3
    Size of section headers:           64 (bytes)
    Number of section headers:         5
    Section header string table index: 4
```

Nous avons donc obtenu un binaire pour ARM 64-bits en Little-Endian.

3 PREMIERE PARTIE : LE BINAIRE ARM64

3.1 Pré-requis

Pour la suite de l'analyse, il est nécessaire d'avoir certains outils spécifiques pour ARM 64-bits:

binutils 2.14 - http://ftp.gnu.org/gnu/binutils/

```
$ ./configure --target=aarch64-linux-gnu --program-prefix=aarch64-linux-gnu-
$ make
$ sudo make install
```

gdb 7.7 - http://ftp.gnu.org/gnu/gdb/

```
$ ./configure --target=aarch64-linux-gnu --program-suffix=-aarch64
$ make
$ sudo make install
```

qemu 2.0 - http://wiki.qemu-project.org/Download

```
$ ./configure --target-list=aarch64-linux-user
$ make
$ sudo make install
```

3.2 Analyse du packer

Un rapide désassemblage de *badbios.bin* permet de voir qu'il s'agit d'un binaire *packé*. Pour éviter une longue analyse probablement inutile, nous allons directement poser un *breakpoint* à l'endroit où le binaire branche sur le binaire *dépacké*. Pour cela nous allons chercher une occurrence de *Branch with Link* dont l'opérande est un registre :

```
$ aarch64-linux-gnu-objdump -S badbios.bin | grep blr
    102c0:       d63f0040        blr     x2
```

3.3 Récupération du programme

Pour récupérer le programme, il suffit de poser un *breakpoint* à l'adresse trouvée précédemment:

```
$ qemu-aarch64 -g 4000 ./badbios.bin
```

Dans une autre console :

```
$ gdb-aarch64
 [...]
(gdb) target remote :4000
Remote debugging using :4000
0x00000000000102cc in ?? ()
(gdb) b *0x102c0
Breakpoint 1 at 0x102c0
(gdb) c
Continuing.

Breakpoint 1, 0x00000000000102c0 in ?? ()
(gdb) p/x $x2
```

```
$1 = 0x400514
(gdb)
```

Il semble que le point d'entrée du binaire *dépacké* soit en `0x400514`. Voyons ce que **qemu** a réservé comme mémoire:

```
$ cat /proc/`pidof qemu-aarch64`/maps
[...]
00400000-00403000 r-xp 00000000 00:00 0
00500000-00511000 rw-p 00000000 00:00 0
[...]
```

Il y a 3 pages réservées avec les droits d'exécution qui contiennent le point d'entrée à partir de l'adresse `0x400000`, ainsi que 17 autres pages réservées très probablement pour les données du programme, à partir de l'adresse `0x500000`.

Nous allons pouvoir récupérer tout ça grâce à **gdb**. Reprenons la session précédemment lancée:

```
(gdb) dump binary memory text.bin 0x00400000 0x00403000
(gdb) dump binary memory data.bin 0x00500000 0x00511000
(gdb)
```

Le binaire obtenu pour *`text.bin`* contient déjà un en-tête *ELF* :

```
$ file text.bin
text.bin: ELF 64-bit LSB executable, version 1 (SYSV), statically linked,
stripped
```

Un rapide **hexdump** sur le fichier *`data.bin`* obtenu permet de voir qu'il y a 65536 octets qui semblent aléatoires (chiffrement ou compression probablement), suivi de 16 octets correspondants à `0x0badb1050badb1050badb1050badb105`, suivi de `0`s.

3.3.1 Analyse du programme

3.3.1.1 Appels système

L'ELF obtenu ayant été *linké* statiquement, une première étape d'analyse consiste en l'identification des différents appels systèmes faits par le programme. Pour *ARM64 linux*, les appels systèmes sont reconnaissables par l'instruction *SVC* et le numéro de l'appel système se trouve dans le registre *x8*.

La valeur du registre *x8* est bien évidemment obscurcie dans le binaire, mais il n'est pas très difficile de recalculer sa valeur avant l'interruption logicielle. Nous retrouvons ainsi 17 appels système, faisant appel à :

- *TARGET_NR_exit_group*
- *TARGET_NR_openat*
- *TARGET_NR_close*
- *TARGET_NR_mmap*
- *TARGET_NR_read*
- *TARGET_NR_munmap*
- *TARGET_NR_write*

Il y à plusieurs appels à *mmap*, il pourra donc être intéressant de voir ce qui est fait avec cette mémoire.

3.3.1.2 Analyse statique

Muni de nos appels système, nous allons maintenant tenter d'en savoir plus sur le programme avant de l'exécuter. Il n'y a qu'un seul point de sortie, à l'adresse `0x400508` (appel système *exit_group*), le point d'entrée se trouvant en `0x400514`.

```
00000000004004d8 <.Label_12>:
  4004d8:   a9bf7bfd    stp   x29, x30, [sp,#-16]!
  4004dc:   93407c03    sxtw  x3, w0
  4004e0:   910003fd    mov   x29, sp
  4004e4:   91000463    add   x3, x3, #0x1
  4004e8:   90000882    adrp  x2, 510000
  4004ec:   8b030c23    add   x3, x1, x3, lsl #3
  4004f0:   91006042    add   x2, x2, #0x18
  4004f4:   f9000043    str   x3, [x2]
  4004f8:   97fffeee    bl    4000b0
  4004fc:   93407c01    sxtw  x1, w0
  400500:   aa0103e0    mov   x0, x1
  400504:   d2800bc8    mov   x8, #0x5e                   // #94
  400508:   d4000001    svc   #0x0
  40050c:   aa0003e1    mov   x1, x0

0000000000400510 <.Label_13>:
  400510:   14000000    b     400510 <.Label_13>

0000000000400514 <_start>:
  400514:   d280001e    mov   x30, #0x0                   // #0
  400518:   910003fd    mov   x29, sp
  40051c:   f94003e0    ldr   x0, [sp]
  400520:   910023e1    add   x1, sp, #0x8
  400524:   17ffffed    b     4004d8 <.Label_12>
```

Une recherche dans le résultat de **objdump** montre qu'il n'y a pas d'appel direct à *exit_group*. La seule fonction appelée entre le point d'entrée et celui de sortie étant en `0x4000b0`.

```
00000000004000b0 <_init>:
  4000b0:   a9be7bfd    stp   x29, x30, [sp,#-32]!
  4000b4:   910003fd    mov   x29, sp
  4000b8:   90000800    adrp  x0, 500000
  4000bc:   d2a00021    mov   x1, #0x10000                // #65536
  4000c0:   91000000    add   x0, x0, #0x0
  4000c4:   910043a2    add   x2, x29, #0x10
  4000c8:   94000a26    bl    402960
  4000cc:   37f800a0    tbnz  w0, #31, 4000e0 <.Label_01>
  4000d0:   f9400ba0    ldr   x0, [x29,#16]
  4000d4:   94000a10    bl    402914
```

Ici, deux fonctions sont appelées: `0x402960` et `0x402914`. Par convention les arguments sont passés via les registres *x0* à *x3*, il est relativement aisé d'en déduire une version C du code:

```
  uint64_t var;

  if ((prepare_data(data_ptr, 0x10000, &var) & 0x40000000) == 0)
      main(var);

  return;
```

3.3.1.2.1 La fonction d'initialisation

Le *data_ptr* correspond à l'adresse à laquelle le fichier *data.bin* a été précédemment récupéré.

L'analyse de la fonction *prepare_data* s'avère très intéressante:

```
0000000000402960 <prepare_data>:
[...]
  402a84:   97fffbe1    bl    401a08
  402a88:   97fff65f    bl    400404 <ECRYPT_init>
  402a8c:   91004276    add   x22, x19, #0x10
  402a90:   d0000861    adrp  x1, 510000 <data_key>
  402a94:   aa1603e0    mov   x0, x22
  402a98:   91000021    add   x1, x1, #0x0
  402a9c:   52801002    mov   w2, #0x80                    // #128
  402aa0:   52800003    mov   w3, #0x0                     // #0
  402aa4:   97fff659    bl    400408 <ECRYPT_keysetup>
  402aa8:   d0000861    adrp  x1, 510000 <data_key>
  402aac:   aa1603e0    mov   x0, x22
  402ab0:   91008021    add   x1, x1, #0x20
  402ab4:   97fff672    bl    40047c <ECRYPT_ivsetup>
[...]
```

La lecture de la fonction située à l'adresse `0x400408`, qui n'est pas obscurcie, permet de retrouver la chaîne: "*expand 16-byte k*". On reconnaît la fonction de préparation de clef d'un algorithme de chiffrement par flux proposé par Daniel Bernstein. L'étude de cette fonction permet de conclure à une variante du *Salsa20*: le *Chacha8* avec une clef de 128 bits.

Le code du *Chacha8* est identifiable par sa suite d'instruction *ADD / XOR / ROR* que l'on retrouve à l'adresse `0x40015c`. Ce code n'étant pas obscurci, il est possible de replacer les symboles suivants:

- *ECRYPT_init,*
- *ECRYPT_keysetup,*
- *ECRYPT_ivsetup,*
- *ECRYPT_encrypt_bytes,*
- *ECRYPT_decrypt_bytes*
- *ECRYPT_keystream_bytes*

Le dernier symbole n'étant jamais appelé.

3.3.1.2.2 La fonction principale

Avant d'aller plus avant, nous allons nous pencher sur la fonction principale. Seules deux fonctions sont appelées :

```
0000000000402914 <main>:
```

```
  402914:   a9be7bfd    stp   x29, x30, [sp,#-32]!
  402918:   910003fd    mov   x29, sp
  40291c:   39400001    ldrb  w1, [x0]
  402920:   a90153f3    stp   x19, x20, [sp,#16]
  402924:   32000021    orr   w1, w1, #0x1
  402928:   39000001    strb  w1, [x0]
  40292c:   aa0003f3    mov   x19, x0
  402930:   97ffff89    bl    402754
  402934:   2a0003f4    mov   w20, w0
  402938:   340000c0    cbz   w0, 402950 <.Label_79>
  40293c:   b9800661    ldrsw x1, [x19,#4]
  402940:   90000000    adrp  x0, 402000
  402944:   912ce000    add   x0, x0, #0xb38
  402948:   f8617800    ldr   x0, [x0,x1,lsl #3]
  40294c:   97fffdc9    bl    402070

0000000000402950 <.Label_79>:
  402950:   2a1403e0    mov   w0, w20
  402954:   a94153f3    ldp   x19, x20, [sp,#16]
  402958:   a8c27bfd    ldp   x29, x30, [sp],#32
  40295c:   d65f03c0    ret
```

La seconde fonction n'est appelée que dans un cas de valeur de retour non nul de la première fonction. Elle sert très probablement à afficher une erreur, ce qu'il est facile de vérifier avec **gdb**. La première fonction appelée en revanche est composée d'une grande boucle principale, ainsi que des blocs basiques sans intérêt:

```
0000000000402754 <main_loop>:
[...]
  402784:   1400000d    b     4027b8 <.Label74>
[...]

00000000004027b8 <.Label74>:
  4027b8:   aa1303e0    mov   x0, x19
[...]
  40285c:   d63f0040    blr   x2
  402860:   39400260    ldrb  w0, [x19]
  402864:   3707faa0    tbnz  w0, #0, 4027b8 <.Label74>

0000000000402868 <.Label75>:
[...]
  402884:   d65f03c0    ret
```

Il semble que nous ayons identifié la boucle principale du programme. Le *BLR x2* laisse penser qu'il doit exister un tableau de pointeur sur fonction. A chaque tour de boucle une de ces fonctions est appelée. Le choix de la fonction dépendant très probablement du résultat des fonctions appelées en début de boucle.

3.3.2 Déchiffrement des données

Les données récupérées dans *data.bin* semblaient être divisés en 65536 octets chiffrés suivis de 16 octets qui sur la base de notre analyse précédente peuvent être nommés *data_key*. En effet, une

réécriture en C du morceau de code assembleur de la fonction *prepare_data* présenté plus haut donnerait:

```
ECRYPT_ctx x;
unsigned char *data_key =
        "\x0b\xad\xb1\x05\x0b\xad\xb1\x05\x0b\xad\xb1\x05\x0b\xad\xb1\x05";
unsigned char *data_nonce = "\x00\x00\x00\x00\x00\x00\x00\x00";

ECRYPT_keysetup(&x, data_key, 128, 0);
ECRYPT_ivsetup(&x, data_nonce);
```

Ces informations sont suffisantes pour permettre le déchiffrement des 65536 octets de données du fichier *data.bin*. Le code pour déchiffrer les données se trouve en annexe, voir 5.1. Le fichier obtenu est largement composé de 0, mais contient aussi 1024 octets de données et 8192 octets qui semblent être chiffrés.

L'étude des chaînes de caractères présentes nous permet d'en savoir plus sur l'utilisation du programme:

```
$ strings -n 11 data_plain.bin
:: Please enter the decryption key:
:: Trying to decrypt payload...
   Wrong key format.
   Invalid padding.
   Cannot open file payload.bin.
:: Decrypted payload written to payload.bin.
payload.bin
XXXXXXXXXXXXXXXX
$
```

3.4 Reconstruction du binaire ELF

Afin de faciliter la suite de l'analyse il peut être intéressant de reconstruire un *ELF* valide à partir des segments *text.bin* et *data.bin* obtenus avec **gdb**. Pour cela nous allons avoir besoin de reconstruire:

- Le *Program Header* qui semble avoir été effacé et qui comporte 2 entrées.
- Le *Section Header*, qui contenait à l'origine 7 sections
  - Section nulle
  - Section `.text` (text.bin)
  - Section `.rodata` (text.bin)
  - Section `.data` (data.bin)
  - Section `.symtab`
  - Section `.strtab`
  - Section `.shstrtab`
- La section *`strtab`* va devoir être reconstruite avec les noms des symboles devinés
- La section *`symtab`* peut être reconstruite avec les informations obtenues par désassemblage.

3.4.1 Program Header

Les segments utilisés pour charger le binaire en mémoire sont ceux obtenus lors de la récupération des zones mémoires réservées par **qemu**. La reconstruction donne donc:

```
#define EXEC_SIZE    0x3000
#define DATA_SIZE    0x11000

static Elf64_Phdr prog_hdr[] = {
    /* Executable segment */
    {
        .p_type   = PT_LOAD,
        .p_offset = 0,
        .p_vaddr  = 0x400000,
        .p_paddr  = 0x400000,
        .p_filesz = EXEC_SIZE,
        .p_memsz  = EXEC_SIZE,
        .p_flags  = PF_X | PF_R,
        .p_align  = 0x100000
    },
    /* Read/Write segment */
    {
        .p_type   = PT_LOAD,
        .p_offset = EXEC_SIZE,
        .p_vaddr  = 0x500000,
        .p_paddr  = 0x500000,
        .p_filesz = DATA_SIZE,
        .p_memsz  = DATA_SIZE,
        .p_flags  = PF_W | PF_R,
        .p_align  = 0x100000
    }
};
```

3.4.2 Section Header

Nous possédons déjà le contenu de 4 sections sur les 7 originellement définies. Pour la section des données en lecture seule, le résultat du désassemblage par **objdump** permet de constater que la dernière fonction se termine en 0x2b38, le reste correspondant à des données. Il est donc déjà possible de décrire ce que nous avons puis de finir de remplir les sections reconstruites avec leur taille et emplacement lorsqu'ils seront connus:

```
#define EHDR_SIZE    0xb0
#define TEXT_SIZE    (0x2b38 - EHDR_SIZE)
#define EXEC_SIZE    0x3000
#define RODATA_SIZE  (EXEC_SIZE - TEXT_SIZE - EHDR_SIZE)
#define DATA_SIZE    0x11000

static Elf64_Shdr shdr[] = {
    { .sh_name = 0 },
    /* .text */
    {
        .sh_type      = SHT_PROGBITS,
        .sh_flags     = SHF_EXECINSTR | SHF_ALLOC,
        .sh_addr      = 0x400000 + EHDR_SIZE,
        .sh_offset    = EHDR_SIZE,
        .sh_size      = TEXT_SIZE,
        .sh_addralign = 8
    },
```

```
    /* .rodata */
    {
        .sh_type      = SHT_PROGBITS,
        .sh_flags     = SHF_ALLOC,
        .sh_addr      = 0x400000 + TEXT_SIZE + EHDR_SIZE,
        .sh_offset    = TEXT_SIZE + EHDR_SIZE,
        .sh_size      = RODATA_SIZE,
        .sh_addralign = 8
    },
    /* .data */
    {
        .sh_type      = SHT_PROGBITS,
        .sh_flags     = SHF_WRITE | SHF_ALLOC,
        .sh_addr      = 0x500000,
        .sh_offset    = EXEC_SIZE,
        .sh_size      = DATA_SIZE,
        .sh_addralign = 8
    },
    /* .symtab */
    {
        .sh_type      = SHT_SYMTAB,
        .sh_offset    = EXEC_SIZE + DATA_SIZE,
        .sh_addralign = 8,
        .sh_link      = 5,
        .sh_entsize   = 24
    },
    /* .strtab */
    {
        .sh_type      = SHT_STRTAB,
        .sh_addralign = 1
    },
    /* .shstrtab */
    {
        .sh_type      = SHT_STRTAB,
        .sh_addralign = 1
    }
};
```

3.4.3 Table des symboles

3.4.3.1 Les données connues

Tout d'abord, les données connues. Nous en connaissons trois :

```
    /* program data */
    { .st_info = STT_OBJECT, .st_shndx = 3,
      .st_value = 0x500000, .st_size = 65536 },
    { .st_info = STT_OBJECT, .st_shndx = 3,
      .st_value = 0x510000, .st_size = 16 },
    { .st_info = STT_OBJECT, .st_shndx = 3,
      .st_value = 0x510020, .st_size = 8 },
```

Pour les nommer, nous pouvons utiliser (voir 5.2):

```
    "ciphered_data", "data_key", "data_iv",
```

3.4.3.2 Les appels de fonction

Pour les appels de fonction, il suffit de chercher les instructions de *Branch with Link* pour obtenir une liste quasi complète. Ne pas oublier d'y inclure le point d'entrée du programme ainsi que le symbole *ECRYPT_keystream_bytes* pour plus de lisibilité:

```
#define STT_GLOBAL ((STB_GLOBAL << 4) | STT_FUNC)

    /* entry point */
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x400514 },
    /* branch with link */
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x4000b0 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x400404 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x400408 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x40047c },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x400498 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x4004a4 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x4004b0 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x400528 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x401a08 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x402048 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x402070 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x4020c4 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x4022ac },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x402364 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x40243c },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x402450 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x4025ac },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x4025cc },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x4025f4 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x402660 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x4026cc },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x402754 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x402914 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x402960 },
```

Pour le nommage, nous pouvons déjà y mettre les noms trouvés jusqu'à présent, et nommer en *func_XX* les fonctions qui ne sont pas encore connues (voir 5.2):

```
      "_start",
      "_init",
      "ECRYPT_init", "ECRYPT_keysetup", "ECRYPT_ivsetup",
      "ECRYPT_encrypt_bytes", "ECRYPT_decrypt_bytes",
      "ECRYPT_keystream_bytes",
      "func_05", "func_06", "func_07", "func_08", "func_09",
      "func_0a", "func_0b", "func_0c", "func_0d", "func_0e",
      "func_0f", "func_10", "func_11", "func_12", "main_loop",
      "main", "prepare_data",
```

3.4.3.3 Les labels de branchement

Pour faciliter la lecture du code, nous pouvons aussi y rajouter les labels. Le format *ELF* spécifie le nommage des labels locaux. Il est important de bien marquer ces labels comme locaux sans quoi les traces de **gdb** seraient rendus inutilisables car pollués par des labels internes. Nous allons donc préfixer ces labels par *.L* et les déclarer en *NOTYPE LOCAL* dans la table des symboles.

Afin d'en obtenir la liste exhaustive, il faut chercher les instructions de branchement, conditionnels ou non, qui ne renseignent pas le *Link Register*. Ensuite, il faut retirer de la liste les labels correspondant déjà à une adresse de fonction car le standard *ELF* spécifie qu'il ne peut y avoir qu'une seule entrée par symbole. Nous obtenons 131 labels, dont 3 correspondent à des fonctions :

```
    /* labels */
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4000d8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4000e0 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4000e8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4000f8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4000fc },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40015c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40032c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x400334 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x400358 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x400390 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40039c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4003c8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4003d8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4003f4 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4004a0 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4004ac },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4004bc },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4004d4 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4004d8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x400510 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x400548 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x400550 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40055c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x400584 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x400590 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4005a0 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4005a8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4005b4 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4005c0 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4005c8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4005d4 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4005e4 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40060c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4006ec },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x400708 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x400850 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x400e08 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x400f68 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x400ff0 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40100c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4010dc },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4011a8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40125c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x401270 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4013cc },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x401444 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x401458 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40146c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x401508 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40153c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40154c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x401564 },
```

```
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40160c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4016c4 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4016d8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x401778 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4017e0 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4017f0 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x401800 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40183c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x401954 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4019f0 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x401a28 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x401a3c },
    /*{ .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402048 },*/
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402090 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402094 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4020a4 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4020bc },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4020ec },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402110 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40211c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402150 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402168 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402174 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4021a8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4021c4 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4021e0 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402210 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402234 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402250 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4022a4 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4022dc },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402314 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40231c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402330 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402344 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402348 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40235c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402394 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4023cc },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4023d4 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4023e8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40241c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402420 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402434 },
    /*{ .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40243c },*/
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402480 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4024bc },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4024c8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4024e8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40251c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402520 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402538 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402540 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402564 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40257c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4025a4 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40260c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402624 },
```

```
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402634 },
    /*{ .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402660 },*/
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402680 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40268c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4026ac },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4026f8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402704 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40273c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402788 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4027b8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402868 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402888 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4028b8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4028e8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402950 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402a08 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402ae4 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402aec },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402b00 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402b1c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402b20 }
```

Pour le nommage, il suffira de prendre des noms tels que *.Label_00* jusqu'à *.Label_7f*.

Pour encore plus de lisibilité, il peut être utile de renseigner les noms des labels de la boucle principale ainsi que ceux correspondants aux appels système identifiés.

3.4.3.4 Les pointeurs sur fonction

Nous avons vu par l'analyse statique qu'il doit exister une table de pointeur sur fonction. La fonction appelée par *prepare_data* située à l'adresse `0x401a08` n'a pas encore été identifiée. Cette fonction, obscurcie, se déroule en 2 étapes. Tout d'abord une même valeur `0x400604` est écrite 31 fois par incrément de 8:

```
0000000000401800 <.Label_3a>:
[...]
  401824:   f9014443    str   x3, [x2,#648]
  401828:   d1024042    sub   x2, x2, #0x90
  40182c:   d1012042    sub   x2, x2, #0x48
  401830:   f103e03f    cmp   x1, #0xf8
  401834:   54fffe61    b.ne  401800 <.Label_3a>
  401838:   14000081    b     401a3c <.Label_3f>
```

Puis, pour chacune des 31 entrées, un *offset* est rajouté, résultant en une table de 31 entrées contenant des adresses correspondant à des fonctions dans le segment de code du programme. Il est possible de la récupérer avec **gdb** en utilisant la commande:

```
(gdb) x/31gx 0x4000801360
```

Muni de ces nouvelles informations, nous allons pouvoir renseigner ces fonctions dans la table des symboles :

```
    /* decrypted addresses */
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x400d9c },
```

```
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x400dac },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x401580 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x401634 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x4016e4 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x401030 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x4010ec },
    { .st_info = STT_FUNC, .st_shndx = 1,
      .st_value = 0x4011b4, .st_size = 0xbc },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x401794 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x400d58 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x400c90 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x400c20 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x400bd0 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x400b78 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x400b04 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x400a8c },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x400a08 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x400978 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x400918 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x4008c4 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x400864 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x4007ec },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x400d24 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x400ce0 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x401970 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x4018d0 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x40187c },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x4005f4 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x4005fc },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x401490 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x40077c },
```

Certaines de ces fonctions se terminent par un *BR x30* qui est un retour de fonction valide sur ARM. Cependant **gdb** ne semblant pas capable de la détecter come un retour de fonction, il peut être utile de renseigner le champ *st_size* lorsque c'est possible.

Pour les nommer, nous pouvons par exemple utiliser les noms *symbol_00* à *symbol_1e*.

Une analyse prenant en compte les appels système précédemment trouvés montre que la fonction responsable de la grande partie de ces appels est le *symbol_1d* que l'on pourra renommer en *do_syscall* par exemple.

3.4.4 Mise en place

Il ne reste plus qu'à finir de remplir la structure d'en-tête des sections après calcul de la taille des différentes sections générées et à concaténer le tout. Il ne faut pas oublier de mettre à jour l'emplacement de la structure d'en-tête des sections. Le code est disponible dans l'annexe en 5.2.

3.5 L'exécution du programme

Nous pouvons désormais tester l'*ELF* nouvellement reconstruit. Compte tenu de ce qui a été identifié jusqu'à présent, on peut s'attendre à devoir rentrer une clef de 64 bits au format hexadécimal :

```
$ qemu-aarch64 ./sstic.out
```

```
:: Please enter the decryption key: 0000000000000000
:: Trying to decrypt payload...
   Invalid padding.
```

Un petit test qui ne coûte pas cher est de tenter de trouver une clef pour laquelle le *padding* est correct. Cela peut se faire avec le *${RANDOM}* de **bash**.

```
$ while [ true ]; do key=${RANDOM}${RANDOM}${RANDOM}${RANDOM}; echo $key;
echo $key | qemu-aarch64 ./sstic.out; done
[...]
9969156191887612
:: Please enter the decryption key: :: Trying to decrypt payload...
:: Decrypted payload written to payload.bin.
^C
$ file payload.bin
payload.bin: data
$ stat -c "%s" payload.bin
8191
```

Il est donc aisé de trouver une clef disposant d'un *padding* valide, mais il semble sage d'assumer que le fichier résultant doit correspondre à un format connu et pas seulement à un bloc de données arbitraires. Le fichier obtenu faisant 8191 octets alors que les données chiffrées sont sur 8192 permet de déduire que le *padding* fait un octet.

3.6 Conclusion partielle

En se fondant sur l'analyse faite jusqu'à présent, et compte tenu du contexte du challenge, il est très probable que nous soyons en présence d'une machine virtuelle travaillant sur des données chiffrées en *Chacha8* contenant des instructions obscurcies.

Nous avons un bloc de données de 65536 octets précédemment déchiffrés. Des chaînes de caractère y ont été identifiées ainsi qu'un bloc de 8192 octets suspecté d'être les données à déchiffrer pour la suite. Le reste de ces données pourraient bien être les instructions pour la machine virtuelle.

Ce type de protection est normalement couplé à une protection anti-*debug*, ce qui n'est pas le cas ici. Une attaque possible consiste à déterminer la séquence des appels systèmes, ce qui devient assez aisé lorsqu'on a un accès direct à un debugger. Cette séquence, les données déchiffrées et ce qu'on a pu observer du programme permettent d'en déduire sa structure globale:

```
    write(1, ":: Please enter the decryption key: ", 36);
    read(0, key, 16);
    /* loop 16 times */
    ret = key_schedule(&ctx, key);
    if (ret == ERROR) goto end:
    write(1, ":: Trying to decrypt payload...\n", 31);
    /* loop 8192 times */
    len = custom_decrypt(&ctx, cipher, plain, 8192);
    if (plain[len - 1] == get_padding()) {
        fd = open("payload.bin", O_WRONLY|O_CREAT);
        write(fd, plain, len - 1);
        close(fd);
        write(1, ":: Decrypted payload written to payload.bin.\n", 45);
        return SUCCESS;
    }
```

```
    write(1, "   Invalid padding.\n", 20);
    return errno;
end:
    write(1, strerror(errno), strlen(strerror(errno)));
    return ernno;
```

Plutôt que de tenter de convertir le programme dans un code intelligible, il paraît plus efficace de s'appuyer sur les séquences d'appel pour affiner les chemins d'exécution qui nous intéressent. Le but étant de pouvoir se placer avec **gdb** aux endroits clef permettant de remonter au calcul du *keystream*.

3.7 Séquence d'appel

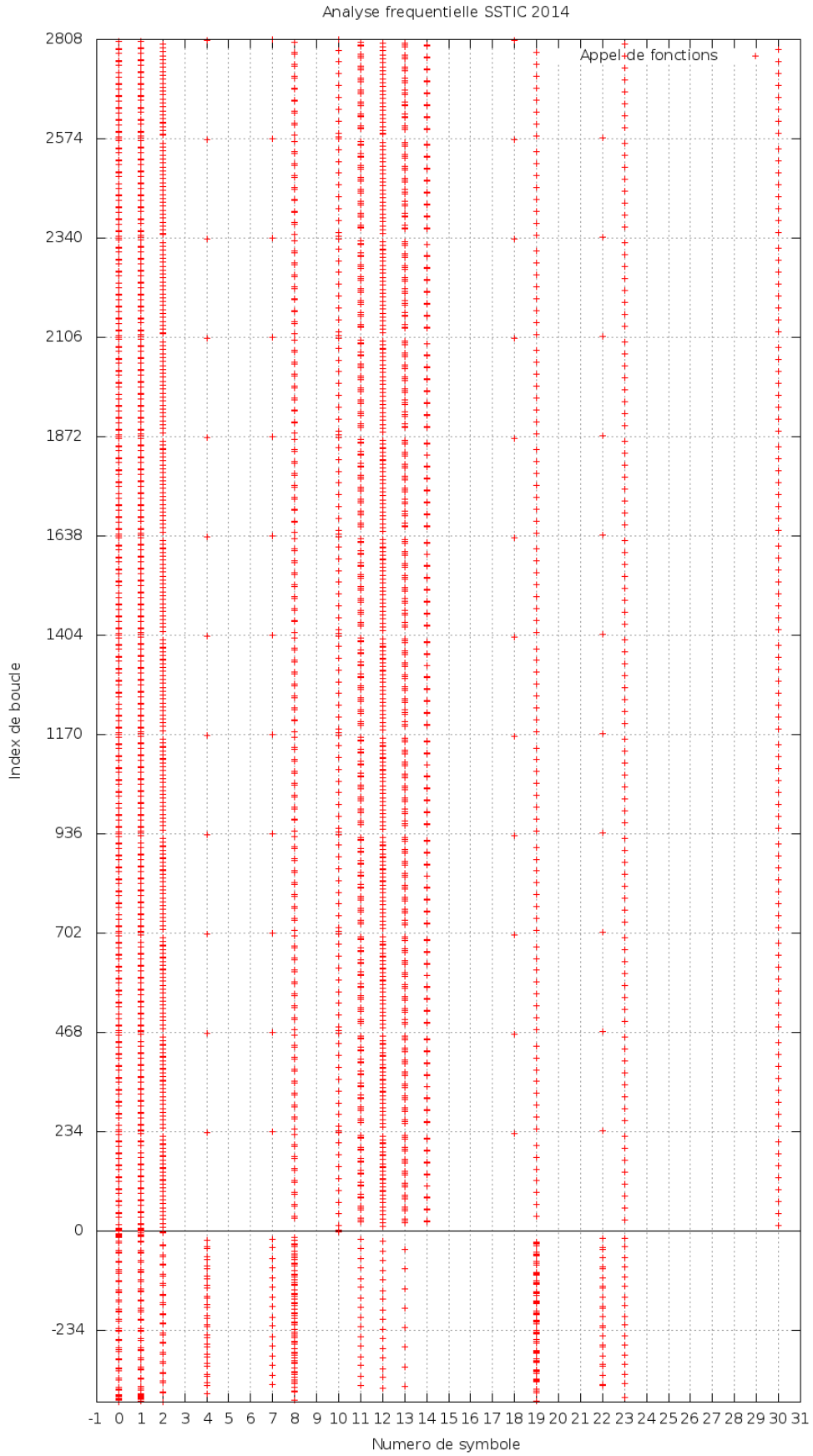
Lors de la reconstruction du binaire *ELF*, nous avons trouvé une table de pointeurs sur fonction. Parmi ces fonctions, celle responsable des appels système a été identifiée. Afin de tenter de déterminer le rôle des autres fonctions, il peut être utile de s'aider de la séquence de leurs appels par rapport à notre idée de la structure du programme.

Pour obtenir les séquences, il faut poser un *breakpoint* dans **gdb** sur le *do_syscall* trouvé précédemment. Le second appel correspondra à la lecture de la clef, le troisième au *write* précédant le déchiffrement des données. Finalement, le quatrième correspondra soit à l'ouverture de *payload.bin*, soit à l'écriture d'un message d'erreur.

En mettant un second *breakpoint* lors du *Branch with Link* sur *x2*, avec affichage de la valeur de *x2*, il sera possible de générer un graphe d'appel exploitable.

A noter que la boucle principale itère près de 2 millions de fois. L'émulation du jeu d'instruction ARM64 par **qemu** avec un si grand nombre d'itération rendrait la génération complète du graphe d'appel extrêmement longue en plus d'être inutile. Il suffit de prendre suffisamment de tours de boucle au début, puis en utilisant la commande *ignore* de **gdb** de prendre les dernières itérations en s'aidant du dernier appel a *write*. Cela permettra de valider la cyclicité des appels de fonction et de déterminer celle responsable du stockage des valeurs intermédiaires lors du calcul du *keystream.*

Le graphe ci-dessous montre les appels des pointeurs sur fonction numérotés de 0 à 30 dans l'ordre de la table. L'index de boucle 0 a été placé sur le second appel à *write*. Le graphe commence après l'appel à *read*:


Analyse frequentielle SSTIC 2014
Appel de fonctions
Index de boucle
Numero de symbole
2808
2574
2340
2106
1872
1638
1404
1170
936
702
468
234
0
-234
-1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

On peut voir qu'il faut 234 tours de boucle principale pour déchiffrer 1 octet du contenu. Une fonction particulièrement intéressante est le *symbol_07*. Elle est appelée une unique fois par caractère de la clef pendant la phase de préparation, puis à la fin de chaque tour du déchiffrement. Une analyse rapide de son contenu montre qu'elle stocke la valeur calculée par la fonction *func_10* par l'instruction *STR* située en 0x401238. Un *breakpoint* posé à cette adresse permet de confirmer qu'il s'agit du résultat du ou-exclusif entre les données et l'octet courant du *keystream*. Le résultat du ou-exclusif est en réalité calculé bit à bit par la fonction *symbol_0a* et se termine par une rotation de 1 vers la gauche.

Une analyse dynamique plus poussée de *symbol_07* permet de s'apercevoir que la clef est stockée en format binaire à l'adresse 0x40008120a6 pendant la phase de préparation, puis le résultat du déchiffrement des 64 premiers octets est stocké en 0x4000812340. Ces adresses peuvent se retrouver par le registre *x2* au retour de cette fonction. Par exemple, pour une clef de 0123456789ABCDEF:

```
(gdb) b symbol_07
Breakpoint 1 at 0x4011c0
(gdb) ignore 1 16
Will ignore next 16 crossings of breakpoint 1.
(gdb) c
Continuing.

Breakpoint 1, 0x00000000004011c0 in symbol_07 ()
(gdb) n
Single stepping until exit from function symbol_07,
which has no line number information.
0x0000000000402860 in main_loop ()
(gdb) p/x $x2
$1 = 0x4000812340
(gdb) disp/8bx 0x40008120a6
1: x/8xb 0x40008120a6
0x40008120a6:   0x01    0x23    0x45    0x67    0x89    0xab    0xcd    0xef
(gdb) disp/8bx 0x4000812340
2: x/8xb 0x4000812340
0x4000812340:   0x00    0xbc    0x68    0x15    0xb5    0x6b    0x1b    0x41
(gdb) ignore 1 8
Will ignore next 8 crossings of breakpoint 1.
(gdb) c
Continuing.

Breakpoint 1, 0x00000000004011c0 in symbol_07 ()
1: x/8xb 0x40008120a6
0x40008120a6:   0x01    0x23    0x45    0x67    0x89    0xab    0xcd    0xef
2: x/8xb 0x4000812340
0x4000812340:   0x23    0x17    0x0f    0xfa    0xb4    0xe2    0x5e    0x8d
(gdb)
```

Seuls les 8 premiers octets nous intéressent dans un premier temps, il n'est donc pas nécessaire de pousser l'analyse plus loin.

3.8 Cryptanalyse

A ce stade le contenu à déchiffrer est connu. Bien que le contenu en clair ne soit pas connu, il paraît cohérent d'assumer qu'il s'agisse d'un type de fichier connu et dont il est possible de deviner les premiers octets.

Le problème peut donc se résumer à trouver une clef qui permette de générer un *keystream* tel qu'il soit égal au résultat d'un ou-exclusif entre la signature de fichier voulue avec le contenu chiffré.

Aidé de notre graphe de séquence d'appel et de **gdb**, on peut voir que la fonction de déchiffrement semble fonctionner avec un registre à décalage. Ce qui pourrait s'écrire de la façon suivante:

$$m_i = c_i \oplus F_i(key)$$

Le test avec une clef de `0000000000000000` permet de voir avec le *breakpoint* trouvé précédemment que le résultat déchiffré est identique au clair. Ce résultat est caractéristique d'un *Linear Feedback Shift Register (LFSR)*.

L'application de l'algorithme de *Berlekamp-Massey* sur le résultat du ou-exclusif entre les données chiffrées et les données récupérés lors du déchiffrement nous permet d'ailleurs d'en déterminer son polynôme de rétroaction:

$$T(x) = x^{64} + x^{63} + x^{61} + x^{60} + 1$$

Partant de là, nous pouvons retrouver la *seed* de ce *LFSR* pour une signature de fichier donnée. L'état précédent du registre à décalage pour la signature *ZIP* donne `0xBC22A7FA52A2B9E1`, et le fichier obtenu en sortie est directement utilisable. Cela peut se vérifier avec le code suivant:

```
uint64_t lfsr = 0xBC22A7FA52A2B9E1;
for (;;) {
    /* feedback polynomial: x^64 + x^63 + x^61 + x^60 + 1 */
    bit = (lfsr ^ (lfsr >> 60lu) ^ (lfsr >> 61lu) ^ (lfsr >> 63lu)) & 1lu;
    lfsr = (lfsr >> 1lu) | (bit << 63lu);
    c <<= 1; c |= bit;
    if (++period == 8) {
        if (fread(&d, 1, 1, fin) <= 0) break;
        d ^= c; fwrite(&d, 1, 1, fout);
        c = 0; period = 0;
    }
}
```

Cependant, nous n'avons pas trouvé la clef attendue par le programme, bien que nous ayons une sortie valide, l'étape de dérivation de clef nous est encore inconnue. Afin d'être le plus exhaustif possible dans notre résolution, nous pouvons tenter de la retrouver. Il est important de noter que si la transformation *F* est une application linéaire, elle a par définition la propriété suivante:

$$F(a \oplus b) = F(a) \oplus F(b)$$

Il sera possible de générer une clef pour une signature de fichier donnée en peu d'itérations:

$$F\left(\sum_i k_i.2^i\right) = \sum_i F\left(k_i.2^i\right) = \sum_i \left(\sum_j f_j^i.2^j\right) = \sum_i (c_i \oplus m_i).2^i$$

Car pour tout bit à 1 de rang *i* de la clef correspondra un ensemble de bits à 1 dans la sortie de l'application linéaire *F* qu'il est possible d'obtenir par un ou exclusif du clair avec le chiffré.

Pour valider cette hypothèse, il faut récupérer le résultat du premier bloc déchiffré pour toutes les clefs ne comportant qu'un seul bit à 1, soit 64 itérations à effectuer.

Une "simple" commande suffira pour obtenir l'impact de chaque bit de la clef:

```
$ for val in `for ((p=0; p<16; p++)); do for i in 1 2 4 8; do for ((b=0;
b<$p; b++)); do echo -n 0; done; echo -n $i; for ((b=1; b<$((16-$p)); b++));
do echo -n 0; done; echo; done; done`; do echo $val | qemu-aarch64 -g 4000
./badbios.bin >/dev/null 2>&1 & echo -e "set height unlimited\ntarget remote
:4000\nb *0x40126c\nignore 1 23\nc\nx/8bx 0x4000812340\nquit\ny\n" | gdb-
aarch64; done | grep 0x4000812340 | cut -d':' -f2 | sed 's/0x//g'
```

Le résultat valide l'hypothèse et il est possible de déduire la valeur de f_j^i pour *i* et *j* entre 0 et 63:

$$f_j^i = k_i \; si \; j = \left(16 * \left\lfloor \frac{i-1}{8} \right\rfloor - \; i + 40\right) [mod\ 64], sinon\ 0$$

Il devient alors aisé de trouver une clef valide pour une signature de fichier donnée.

3.9 Récupération de la clef

L'implémentation du calcul de la clef pour un clair donné est désormais possible. Par exemple avec le code suivant:

```
uint64_t key = 0, magic = strtoll(av[1], NULL, 16) ^ 0x00bc6815b56b1b41;
for (int i = 0; i < 64; i++)
    if (magic & (1lu << i))
        key |= 1lu << ((16 * (int)floor((i - 1) / 8.) - i + 40) % 64);
printf("%016llX\n", key);
```

Il faudra tester une liste de signatures sur 64 bits afin de trouver la clef:

```
$ # ELF32: 7F454C4601010100
$ ./key 7F454C4601010100
5BACB004FD3F4994
$ # ELF64: 7F454C4602010100
$ ./key 7F454C4602010100
DBADB004FD3F4994
$ # GPG  : 2D2D2D2D2D2D2D2D
$ ./key 2D2D2D2D2D2D2D2D
32C4D86C68134539
$ # ZERO : 0000000000000000
$ ./key 0000000000000000
5BADB105017A2C50
$ # PNG  : 89504E470D0A1A0A
$ ./key 89504E470D0A1A0A
3A0C01A5236FC894
$ # 3GPP : 0000001466747970
$ ./key 0000001466747970
97F18D18017A2C00
$ # DOCX : 504B030414000060
$ ./key 504B030414000060
0BADB10915DEAD11
$ # ZIP  : 504B030414000000
$ ./key 504B030414000000
0BADB10515DEAD11
$
```

Il semblerait que la dernière clef soit la bonne. On peut clairement lire **"badbios is dead !!"**, et ça ne semble pas être un hasard:

```
$ echo 0BADB10515DEAD11 | qemu-aarch64 ./badbios.bin
:: Please enter the decryption key: :: Trying to decrypt payload...
:: Decrypted payload written to payload.bin.
$ file payload.bin
payload.bin: Zip archive data, at least v2.0 to extract
$ unzip payload.bin
Archive:  payload.bin
  inflating: mcu/upload.py
  inflating: mcu/fw.hex
```

Et maintenant, la seconde partie…

4 SECONDE PARTIE : LE MICROCONTROLEUR

4.1 Analyse du firmware

Le fichier *fw.hex* obtenu est au format *Intel HEX* et contient visiblement un *firmware* de microcontrôleur.

Lorsqu'on ouvre *upload.py*, on peut y lire le commentaire suivant:

```
# Microcontroller architecture appears to be undocumented.
# No disassembler is available.
```

Une autre information très importante fournie par le script Python concerne l'organisation de la mémoire:

```
#    == MEMORY MAP ==
#
#    [0000-07FF] - Firmware                  \
#    [0800-0FFF] - Unmapped                  | User
#    [1000-F7FF] - RAM                       /
#    [F000-FBFF] - Secret memory area        \
#    [FC00-FCFF] - HW Registers              | Privileged
#    [FD00-FFFF] - ROM (kernel)              /
```

Cela laisse supposer que le microcontrôleur travaille sur des mots de taille 16 bits.

Le script Python permet de charger le *firmware* obtenu sur une machine distante. Commençons par voir ce que ça donne:

```
$ python3.2 upload.py
-------------------------------------------
----- Microcontroller firmware uploader -----
-------------------------------------------

:: Serial port connected.
:: Uploading firmware... done.

System reset.
Firmware v1.33.7 starting.
Execution completed in  8339 CPU cycles.
Halting.
```

La suite de l'épreuve est assez claire. Il va falloir déterminer le jeu d'instruction du microcontrôleur à partir du *firmware* fourni de manière à pouvoir exploiter le code du noyau pour aller lire la zone secrète.

Beaucoup d'outils existent pour faire des conversions de *Intel HEX* vers binaire et vice-versa. Une fois converti au format binaire, on retrouve dans le *firmware* certaines des chaînes de caractères affichées lors du test:

```
$ strings -n6 fw.bin
YeahRiscIsGood!
Firmware v1.33.7 starting.
Halting.
```

4.1.1 Récupération de la ROM

L'organisation de la mémoire donne l'adresse et la taille de la zone réservée pour le code du noyau. Afin d'avoir le plus d'exemple de code possible pour comprendre le jeu d'instruction, mais aussi mieux comprendre le fonctionnement du microcontrôleur, il serait intéressant de pouvoir le récupérer.

La méthode la plus évidente pour afficher une chaîne de caractère sur un microcontrôleur basique n'ayant pas de fonction *printf()* consiste en général à faire:

```
uart_send(message, taille);
```

Ce qui se traduit en assembleur par:

```
mov a0, message
mov a1, taille
call uart_send
```

Le microcontrôleur communique avec l'extérieur par un port série, nous devrions donc pouvoir y envoyer n'importe quelle valeur d'octet puisqu'il n'y a pas de raison a priori pour qu'il n'affiche que du texte.

Parmi les chaînes de caractères, on peut voir un "*YeahRiscIsGood!*" qui semble être un indice sur l'architecture du microcontrôleur. Une spécificité du RISC est d'avoir toutes ses instructions de même taille, qui est la taille des mots du microcontrôleur. Cela se traduit par plus de complexité pour assigner une constante à un registre puisqu'il n'est pas possible pour des raisons évidentes de place de coder sur N bits à la fois l'opération, le registre de destination et un mot de N bits. Le pseudo code précédent pourrait alors s'écrire (un exemple parmi d'autres):

```
mov.hi a0, message_hi
or     a0, message_lo
mov.hi a1, taille_hi
or     a1, taille_lo
call uart_send
```

La chaîne "*Firmware v1.33.7 starting.\n*" présente à l'adresse 0x018c est la première à être affichée. On peut aussi voir que le *firmware* commence par:

```
$ hexdump -vCn10 fw.bin
00000000  21 00 11 1b 20 01 10 8c  c0 d2                    |!... .....|
0000000a
$
```

On y retrouve `01` et `8c`, ainsi que `0x1b = 27` qui correspond à la taille de la chaîne "**Firmware v1.33.7 starting.\n**". On peut ainsi déduire les instructions pour assigner une valeur à un registre qui sont sous la forme :

```
2X HH 1X LL     # mov rX, #0xHHLL
```

Nous avons a priori suffisamment d'information pour afficher le contenu de la mémoire du microcontrôleur pour toutes les zones pour lesquelles les permissions de lecture sont accordées. Celle qui nous intéresse pour le moment est la ROM qui contient le noyau. Evidemment la zone secrète serait encore plus intéressante mais sa lecture nous renvoie:

```
[ERROR] Printing at unallowed address. CPU halted.
```

Pour récupérer la ROM, il faut remplacer l'adresse de la première chaîne de caractère par le début de la zone allouée à la ROM, et la longueur par la longueur de la zone, soit:

```
21 03 11 00    // mov r01, #0x0300
20 fd 10 00    // mov r00, #0xFD00
c0 d2
```

Il faut penser à modifier le script Python pour convertir le résultat de manière affichable.

Après reconversion au format *iHex* et chargement, nous obtenons bien le code du noyau:

```
$ python3.2 upload.py
-------------------------------------------
----- Microcontroller firmware uploader -----
-------------------------------------------

:: Serial port connected.
:: Uploading firmware... done.
System reset.

b'5000a06c210011037210a8122200120281027112220f010006001c094d0002100112b20fe105
ac0be300021fc111022001201f210b3f220fc1022c074550020fc1020c06c5155c09ed80020fc
1020c06026fc1612210011013444e561e264e3647332a7f62301130082234125c056d80021001
10e20fe1086c06c2400140221fd112820f01000c03c600421fd1136c034600421fd114ac02c20
fc1020311122001236c03220fc103a21ef11fec016d80021001101220112006e3017111e401844
24034d00f22001201230113001f10272229113f102d00f230013015222a0067223f102b3f2d00f
5e002dfc1d002cf01c00388859882a001a013bbb5111a01a69e8799ca80869e8799dac02b00e3
999e9e8f9db688a711ab3e2d00f2100113320fe1026c3c2b3025b4552524f525d205072696e74
696e6720617420756e616c6c6f7765642061646472657373732e204350552068616c7465642e0a0
05b4552524f525d20556e646566696e65642073797374656d2063616c6c2e204350552068616c
7465642e0a0053797374656d2072657365742e0a000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000'
Execution completed in 18713 CPU cycles.
Halting.
```

4.1.2 Le jeu d'instruction

Nous connaissons déjà les instructions permettant de charger une valeur immédiate dans un registre. L'analyse de l'instruction *C0* nous amène à penser qu'il s'agit d'un appel de fonction relatif à l'adresse courante.

De plus, une étude des binaires obtenus permet de voir que:

- aucune instruction ne semble commencer par 0.
- les instructions commençant par D semblent être des retours de fonction / interruption

- les instructions commençant par C sont des appels de fonction
- les instructions commençant par 1 ou 2 sont des assignations de valeurs immédiates
- il y a 16 registres numérotés de 0 à F
- les instructions de branchement et d'appel de fonction sont relatives

Il manque encore 3 grandes catégories d'instructions:

- les instructions de l'unité arithmétique et logique
- les chargements / stockages en adressage basé
- les sauts conditionnels

Les instructions RISC se font généralement sous la forme:

```
operateur   r_dest, r_src1, r_src2
load/store  r_value, [r_base, r_idx]
```

Partant de cette constatation, il semblerait que le format des instructions arithmétiques et logiques ainsi que celles de chargement et stockages soient de la forme *AXYZ* avec *A* l'operateur, *X* la destination et *Y* et *Z* les registres sources.

Pour retrouver à quoi correspondent les codes d'opérateur, une possibilité est de:

- charger deux registres sources avec des valeurs différentes
- tenter des instructions avec un registre de destination différent à chaque fois
- afficher le contenu de la zone mémoire contenant les registres
- comparer le résultat de chaque instruction par rapport aux opérations possibles

Le but est de pouvoir deviner par leur résultat les fonctions des opérateurs *OP_x*. Par exemple:

```
20 03 10 01     # r00 = 0x0301
21 02 11 80     # r01 = 0x0280
32 01           # r02 = r00 OP_3 r01
43 01           # r03 = r00 OP_4 r01
54 01           # r04 = r00 OP_5 r01
65 01           # r05 = r00 OP_6 r01
76 01           # r06 = r00 OP_7 r01
87 01           # r07 = r00 OP_8 r01
98 01           # r08 = r00 OP_9 r01
E9 01           # r09 = r00 OP_E r01
FA 01           # r10 = r00 OP_F r01
20 05 10 81     # r00 = 0x0581
21 00 10 01     # r01 = 0x0001
C8 02           # affiche le contenu de [0x0301, 0x0280]
20 FC 10 00     # r00 = 0xFC00
21 01 10 00     # r01 = 0x0100
C8 02           # affiche les registres
```

Malheureusement ce code échouera avec l'erreur suivante :

```
System reset.
-- Exception occurred at FF36: Invalid instruction.
   r0:FF36     r1:0000     r2:0100     r3:0236
```

```
   r4:FD00     r5:0581    r6:0081    r7:8280
   r8:0001     r9:0000   r10:0000   r11:0000
  r12:0000    r13:EFFE   r14:0000   r15:FD1C
   pc:FF36 fault_addr:0000 [S:1 Z:0] Mode:kernel
CLOSING: Invalid instruction.
```

Il semble qu'il y ait un problème lorsqu'on fait un appel système alors que plusieurs registres ont été modifiés. En jouant plus avec le code on se rend compte que les instructions commençant par *E* déréférencent les registres sources. Il suffira donc de déréférencer une adresse de la zone secrète pour déclencher une erreur et afficher les registres:

```
System reset.
-- Exception occurred at 001E: Memory access violation.
   r0:0301     r1:0280    r2:0181    r3:0381
   r4:0200     r5:0581    r6:0081    r7:8280
   r8:0001     r9:0055   r10:0000   r11:F000
  r12:0000    r13:EFFE   r14:0000   r15:0000
   pc:001E fault_addr:F301 [S:1 Z:0] Mode:user
CLOSING: Memory access violation.
```

Le choix des valeurs de *r0* et *r1* est tel que l'on puisse différencier les opérations logiques et arithmétiques. On constate que:

- r2 = r0 XOR r1
- r3 = r0 OR r1
- r4 = r0 AND r1
- r5 = r0 + r1
- r6 = r0 – r1
- r7 = r0 * r1
- r8 = r0 / r1
- *E* correspond à un chargement par adressage basé
- Par élimination, *F* doit être le stockage par adressage basé.

A partir de ces informations, la lecture du code partiellement désassemblé permet de remonter aux formats des instructions et à déterminer les instructions de saut manquantes. Le format est le suivant :

<table>
<tr><td rowspan="3">OPCODE</td><td rowspan="2">Rd</td><td>Rn</td><td>Rm</td></tr>
<tr><td colspan="2">Imm.</td></tr>
<tr><td>Flags</td><td colspan="2">Imm.</td></tr>
</table>

A noter que les instructions *D80F* et *D00F* semblent faire exception car *F* pourrait désigner le *link register r15* tandis que le reste de l'instruction semble être au format des instructions de saut.

Nous pouvons dresser une liste des instructions à partir des déductions précédentes. Le but étant de pouvoir écrire un désassembleur:

```
2X VV 1X VV     MOV  rX, #VVVV (20 pour MSB, 10 pour LSB)
3X YZ           XOR  rX, rY, rZ
```

```
4X YZ           OR   rX, rY, rZ
5X YZ           AND  rX, rY, rZ
6X YZ           ADD  rX, rY, rZ
7X YZ           SUB  rX, rY, rZ
8X YZ           MUL  rX, rY, rZ
9X YZ           DIV  rX, rY, rZ
AV VV           Jxx  +#VVV si xx corresponds aux drapeaux ZS
BV VV           JMP  +#VVV #VVV compris entre -512 et 511
C8 VV           SWI  #VV
CV VV           CALL +#VVV
D0 0F           RET
D8 0F           SYSEXIT
D0 00           SYSENTER
EX YZ           LDR rX, [rY, rZ]
FX YZ           STR rX, [rY, rZ]
```

Le code du désassembleur peut se trouver en annexe, voir 5.3.

4.2 Analyse de la ROM

Muni de notre désassembleur, nous pouvons inspecter les appels système. Pour commencer, regardons l'initialisation de la table contenant leurs adresses:

```
SYS_reset:
FD70: MOV   r01, #0x000E
FD74: MOV   r00, #0xFE86
FD78: CALL  #0xFDE6
FD7A: MOV   r04, #0x0002
FD7E: MOV   r01, #0xFD28    // SYS_shutdown
FD82: MOV   r00, #0xF000
FD86: CALL  #0xFDC4         // write_short
FD88: ADD   r00, r00, r04
FD8A: MOV   r01, #0xFD36    // SYS_uart_send
FD8E: CALL  #0xFDC4         // write_short
FD90: ADD   r00, r00, r04
FD92: MOV   r01, #0xFD4A    // SYS_decrypt
FD96: CALL  #0xFDC4         // write_short
FD98: MOV   r00, #0xFC20
FD9C: XOR   r01, r01, r01
FD9E: MOV   r02, #0x0036
FDA2: CALL  #0xFDD6
FDA4: MOV   r00, #0xFC3A
FDA8: MOV   r01, #0xEFFE
FDAC: CALL  #0xFDC4
FDAE: SYSEXIT
```

Idéalement, s'il est possible de remplacer une des adresses de la table des appels systèmes par une adresse dans l'espace utilisateur, cela permettrait d'exécuter notre propre code avec les privilèges du noyau. L'appel système 3 qui semble être une fonction de déchiffrement écrit son résultat dans une adresse mémoire fournie par l'utilisateur sans la moindre vérification:

```
SYS_decrypt:
FD4A: MOV   r00, #0xFC20
FD4E: CALL  #0xFDB0
[...]
```

```
FD6A: OR    r01, r02, r05
FD6C: CALL  #0xFDC4         // write_short
FD6E: SYSEXIT
[...]
write_short:
FDC4: MOV   r02, #0x0001
FDC8: MOV   r03, #0x0100
FDCC: STR   r01, [r00, r02]
FDCE: SUB   r02, r02, r02
FDD0: DIV   r01, r01, r03
FDD2: STR   r01, [r00, r02]
```

Le contenu écrit en revanche est moins direct: c'est apparemment le résultat d'un déchiffrement. Un test rapide permet de découvrir que lors d'un appel direct, le premier mot de 16 bits écrit est `0x07C0` qui se trouve dans l'espace utilisateur.

4.3 Exploitation

Partant de ces constations, la suite est relativement simple. Une étude de l'appel système responsable de l'affichage des caractères permet de connaître l'adresse à laquelle écrire pour envoyer des caractères sur le port série: `0xFC00`.

La procédure est la suivante:

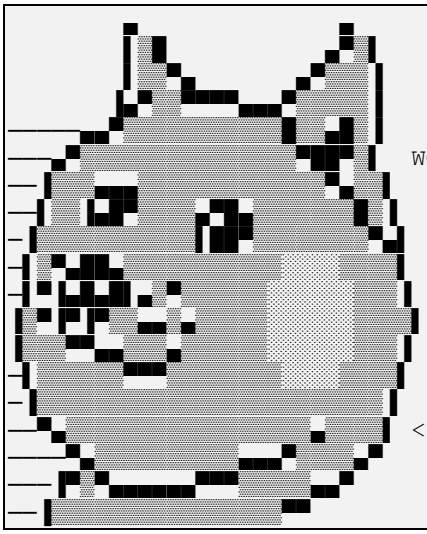
- placer à l'adresse `0x07C0` une boucle qui lit `0xC00` octets à partir de l'adresse de la zone secrète `0xF000` et écrit le contenu dans le registre du port série
- renseigner *`r00`* avec l'adresse de la table des appels systèmes
- appeler *`SYS_decrypt`*
- appeler *SYS_shutdown*

Le *shellcode*:

```
.org 0x0000
20 F0 10 00     // MOV   r00, #0xF000
C8 03           // SWI   #0x03
C8 01           // SWI   #0x01

.org 0x07C0
2D FC 1D 00     // MOV   r13, #0xFC00
2C F0 1C 00     // MOV   r12, #0xF000
2A 00 1A 01     // MOV   r10, #0x0001
2B 0C 1B 00     // MOV   r11, #0x0C00
38 88           // XOR   r08, r08, r08
37 77           // XOR   r07, r07, r07
E9 C7           // LDR   r09, [r12, r07]
F9 D8           // STR   r09, [r13, r08]
67 7A           // ADD   r07, r07, r10
7B BA           // SUB   r11, r11, r10
A7 F6           // JGE   #0x07D4
C8 00           // SWI   #0x00
```

4.4 Résultat



```
WOW

  SUCH EXPLOIT

    VERY CHALLENGING

    SO OPERATIONAL

  MUCH WIN

<66a65dc050ec0c84cf1dd5b3bbb75c8c@challenge.sstic.org>
```

5 ANNEXES

5.1 Chacha8

```
/*
 * Data decrypter for SSTIC Challenge 2014 - part 1
 *
 * Compile with:
 *
 *   gcc -c chacha.c
 *   gcc main.c chacha.o -o ecrypt -Wall -Wextra -pedantic -std=c99
 *
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

#include "ecrypt-sync.h"

#define Q_KEY   "\x0b\xad\xb1\x05"
#define NONCE   "\x00\x00\x00\x00\x00\x00\x00\x00"
#define ENC_LEN 65536

unsigned char cipher[ENC_LEN];
unsigned char plain[ENC_LEN];

int main(int ac, char** av)
{
    ECRYPT_ctx x;
    FILE *fp, *fs;
    struct stat st;
    unsigned char *key = (unsigned char *)Q_KEY Q_KEY Q_KEY Q_KEY;

    if (ac < 3) {
        fprintf(stderr, "Usage: %s <infile> <outfile>\n", av[0]);
        return -1;
    }

    if (stat(av[1], &st) < 0) {
        perror("stat");
        return -1;
    }

    if (st.st_size < ENC_LEN) {
        fprintf(stderr, "<infile> must be at least %d Bytes\n", ENC_LEN);
        return -1;
    }

    fp = fopen(av[1], "r");
    if (fp == NULL) {
        perror("open <infile>");
        return -1;
    }

    fs = fopen(av[2], "w+");
    if (fs == NULL) {
        perror("open <outfile>");
        fclose(fp);
        return -1;
    }

    ECRYPT_keysetup(&x, key, 128, 8);
    ECRYPT_ivsetup(&x, (unsigned char *)NONCE);
```

```
    fread(cipher, 1, ENC_LEN, fp);

    ECRYPT_decrypt_bytes(&x, cipher, plain, ENC_LEN);

    fwrite(plain, 1, ENC_LEN, fs);

    fclose(fp);
    fclose(fs);

    return 0;
}
```

5.2 ELF reconstruction

```
/*
 * SSTIC 2014 ELF rebuilder - Part 1
 *
 * Compile with:
 *   gcc elf_reconstruct.c -o elf_reconstruct -Wall -Wextra -pedantic -std=c99
 *
 * Usage:
 *   ./elf_reconstruct [outfile]
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <elf.h>

#define EXEC_SEGMENT "text.bin"
#define DATA_SEGMENT "data.bin"

#define EHDR_SIZE    0xb0
#define TEXT_SIZE    (0x2b38 - EHDR_SIZE)
#define EXEC_SIZE    0x3000
#define RODATA_SIZE  (EXEC_SIZE - TEXT_SIZE - EHDR_SIZE)
#define DATA_SIZE    0x11000

static Elf64_Phdr prog_hdr[] = {
    /* Executable segment */
    {
        .p_type   = PT_LOAD,
        .p_offset = 0,
        .p_vaddr  = 0x400000,
        .p_paddr  = 0x400000,
        .p_filesz = EXEC_SIZE,
        .p_memsz  = EXEC_SIZE,
        .p_flags  = PF_X | PF_R,
        .p_align  = 0x100000
    },
    /* Read/Write segment */
    {
        .p_type   = PT_LOAD,
        .p_offset = EXEC_SIZE,
        .p_vaddr  = 0x500000,
        .p_paddr  = 0x500000,
        .p_filesz = DATA_SIZE,
        .p_memsz  = DATA_SIZE,
        .p_flags  = PF_W | PF_R,
        .p_align  = 0x100000
    }
```

```
};

static Elf64_Shdr shdr[] = {
    { .sh_name = 0 },
    /* .text */
    {
        .sh_type      = SHT_PROGBITS,
        .sh_flags     = SHF_EXECINSTR | SHF_ALLOC,
        .sh_addr      = 0x400000 + EHDR_SIZE,
        .sh_offset    = EHDR_SIZE,
        .sh_size      = TEXT_SIZE,
        .sh_addralign = 8
    },
    /* .rodata */
    {
        .sh_type      = SHT_PROGBITS,
        .sh_flags     = SHF_ALLOC,
        .sh_addr      = 0x400000 + TEXT_SIZE + EHDR_SIZE,
        .sh_offset    = TEXT_SIZE + EHDR_SIZE,
        .sh_size      = RODATA_SIZE,
        .sh_addralign = 8
    },
    /* .data */
    {
        .sh_type      = SHT_PROGBITS,
        .sh_flags     = SHF_WRITE | SHF_ALLOC,
        .sh_addr      = 0x500000,
        .sh_offset    = EXEC_SIZE,
        .sh_size      = DATA_SIZE,
        .sh_addralign = 8
    },
    /* .symtab */
    {
        .sh_type      = SHT_SYMTAB,
        .sh_offset    = EXEC_SIZE + DATA_SIZE,
        .sh_addralign = 8,
        .sh_link      = 5,
        .sh_entsize   = 24
    },
    /* .strtab */
    {
        .sh_type      = SHT_STRTAB,
        .sh_addralign = 1
    },
    /* .shstrtab */
    {
        .sh_type      = SHT_STRTAB,
        .sh_addralign = 1
    }
};

#define STT_GLOBAL ((STB_GLOBAL << 4) | STT_FUNC)

static Elf64_Sym symtab[] = {
    { .st_name = 0 },
    /* decrypted addresses */
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x400d9c },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x400dac },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x401580 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x401634 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x4016e4 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x401030 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x4010ec },
    { .st_info = STT_FUNC, .st_shndx = 1,
      .st_value = 0x4011b4, .st_size = 0xbc },
```

```
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x401794 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x400d58 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x400c90 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x400c20 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x400bd0 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x400b78 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x400b04 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x400a8c },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x400a08 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x400978 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x400918 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x4008c4 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x400864 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x4007ec },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x400d24 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x400ce0 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x401970 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x4018d0 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x40187c },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x4005f4 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x4005fc },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x401490 },
    { .st_info = STT_FUNC, .st_shndx = 1, .st_value = 0x40077c },
    /* program data */
    { .st_info = STT_OBJECT, .st_shndx = 3,
      .st_value = 0x500000, .st_size = 65536 },
    { .st_info = STT_OBJECT, .st_shndx = 3,
      .st_value = 0x510000, .st_size = 16 },
    { .st_info = STT_OBJECT, .st_shndx = 3,
      .st_value = 0x510020, .st_size = 8 },
    /* entry point */
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x400514 },
    /* branch with link */
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x4000b0 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x400404 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x400408 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x40047c },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x400498 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x4004a4 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x4004b0 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x400528 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x401a08 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x402048 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x402070 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x4020c4 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x4022ac },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x402364 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x40243c },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x402450 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x4025ac },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x4025cc },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x4025f4 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x402660 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x4026cc },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x402754 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x402914 },
    { .st_info = STT_GLOBAL, .st_shndx = 1, .st_value = 0x402960 },
    /* labels */
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4000d8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4000e0 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4000e8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4000f8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4000fc },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40015c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40032c },
```

```
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x400334 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x400358 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x400390 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40039c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4003c8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4003d8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4003f4 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4004a0 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4004ac },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4004bc },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4004d4 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4004d8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x400510 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x400548 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x400550 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40055c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x400584 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x400590 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4005a0 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4005a8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4005b4 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4005c0 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4005c8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4005d4 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4005e4 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40060c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4006ec },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x400708 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x400850 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x400e08 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x400f68 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x400ff0 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40100c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4010dc },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4011a8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40125c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x401270 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4013cc },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x401444 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x401458 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40146c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x401508 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40153c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40154c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x401564 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40160c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4016c4 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4016d8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x401778 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4017e0 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4017f0 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x401800 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40183c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x401954 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4019f0 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x401a28 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x401a3c },
    /*{ .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402048 },*/
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402090 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402094 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4020a4 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4020bc },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4020ec },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402110 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40211c },
```

```
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402150 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402168 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402174 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4021a8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4021c4 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4021e0 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402210 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402234 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402250 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4022a4 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4022dc },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402314 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40231c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402330 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402344 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402348 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40235c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402394 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4023cc },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4023d4 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4023e8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40241c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402420 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402434 },
    /*{ .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40243c },*/
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402480 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4024bc },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4024c8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4024e8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40251c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402520 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402538 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402540 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402564 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40257c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4025a4 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40260c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402624 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402634 },
    /*{ .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402660 },*/
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402680 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40268c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4026ac },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4026f8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402704 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x40273c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402788 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4027b8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402868 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402888 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4028b8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x4028e8 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402950 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402a08 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402ae4 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402aec },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402b00 },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402b1c },
    { .st_other = STV_HIDDEN, .st_shndx = 1, .st_value = 0x402b20 }
};

static char *shstrtab[] = {
    "",
    ".text",
    ".rodata",
```

```
    ".data",
    ".symtab",
    ".strtab",
    ".shstrtab"
};

static char *strtab[] = {
    "",

    "symbol_00", "symbol_01", "symbol_02", "symbol_03", "symbol_04",
    "symbol_05", "symbol_06", "symbol_07", "symbol_08", "symbol_09",
    "symbol_0a", "symbol_0b", "symbol_0c", "symbol_0d", "symbol_0e",
    "symbol_0f", "symbol_10", "symbol_11", "symbol_12", "symbol_13",
    "symbol_14", "symbol_15", "symbol_16", "symbol_17", "symbol_18",
    "symbol_19", "symbol_1a", "symbol_1b", "symbol_1c", "syscall_wrap",
    "symbol_1e",

    "ciphered_data", "data_key", "data_iv",
    "_start",

    "_init",
    "ECRYPT_init", "ECRYPT_keysetup", "ECRYPT_ivsetup",
    "ECRYPT_encrypt_bytes", "ECRYPT_decrypt_bytes",
    "ECRYPT_keystream_bytes",
    "func_05", "write_sym_tbl", "SYS_read", "func_08", "func_09",
    "func_0a", "func_0b", "func_0c", "func_0d", "func_0e",
    "func_0f", "func_10", "func_11", "func_12", "main_loop",
    "main", "prepare_data",

    ".Label_00", ".Label_01", ".Lsalsa20_wordtobyte", ".Label_03", ".Label_04",
    ".Label_05", ".Label_06", ".Label_07", ".Label_08", ".Label_09",
    ".Label_0a", ".Label_0b", ".Label_0c", ".Label_0d", ".Label_0e",
    ".Label_0f", ".Label_10", ".Label_11", ".Label_12", ".Label_13",
    ".Label_14", ".Label_15", ".Label_16", ".Label_17", ".Label_18",
    ".Label_19", ".Label_1a", ".Label_1b", ".Label_1c", ".Label_1d",
    ".Label_1e", ".Label_1f", ".Ldo_sys_openat", ".Label_21",
    ".Ldo_sys_close", ".Label_23", ".Ldo_secure_read", ".Ldummy_bbl_05",
    ".Label_26", ".Lsecure_read", ".Label_28", ".Label_29", ".Label_2a",
    ".Ldo_secure_rw", ".Lsecure_rw_do_write", ".Lsecure_rw_write",
    ".Label_2e", ".Label_2f", ".Label_30", ".Lsys_close",
    ".Label_32", ".Lsecure_rw", ".Label_34", ".Label_35", ".Label_36",
    ".Label_37", ".Label_38", ".Label_39", ".Lsetup_sym_tbl",
    ".Label_3b", ".Label_3c", ".Label_3d", ".Lsys_openat",
    ".Ldo_write_sym_tbl", ".Label_40",
    ".Label_41", ".Label_42", ".Label_43", ".Label_44", ".Label_45",
    ".Label_46", ".Label_47", ".Label_48", ".Label_49", ".Label_4a",
    ".Label_4b", ".Label_4c", ".Label_4d", ".Label_4e", ".Label_4f",
    ".Label_50", ".Label_51", ".Label_52", ".Label_53", ".Label_54",
    ".Label_55", ".Label_56", ".Label_57", ".Label_58", ".Label_59",
    ".Label_5a", ".Label_5b", ".Label_5c", ".Label_5d", ".Label_5e",
    ".Label_5f", ".Label_60", ".Label_61", ".Label_62", ".Label_63",
    ".Label_64", ".Label_65", ".Label_66", ".Label_67", ".Label_68",
    ".Label_69", ".Label_6a", ".Label_6b", ".Label_6c", ".Label_6d",
    ".Label_6e", ".Label_6f", ".Label_70", ".Label_71", ".Label_72",
    ".Ldummy_bbl_00", ".Lmain_while_loop", ".Lmain_while_exit",
    ".Ldummy_bbl_01", ".Ldummy_bbl_02", ".Ldummy_bbl_03", ".Label_79",
    ".Label_7a", ".Label_7b", ".Label_7c", ".Label_7d", ".Label_7e",
    ".Label_7f"
};

#define STRTAB_MAX_SZ   2048

static char rd_buf[DATA_SIZE];
static char real_strtab[STRTAB_MAX_SZ];
static char real_shstrtab[STRTAB_MAX_SZ];
```

```
static int strtab_len;
static int shstrtab_len;

enum {
    SECT_NONE,
    SECT_TEXT,
    SECT_RODATA,
    SECT_DATA,
    SECT_SYMTAB,
    SECT_STRTAB,
    SECT_SHSTRTAB
};

int main(int ac, char **av)
{
    Elf64_Ehdr *ehdr;
    FILE *fp, *fs;
    unsigned int i;
    size_t val;
    Elf32_Off sec_hdr_offt;
    char *name = "sstic.elf";

    if (ac > 1) {
        name = av[1];
    }

    fs = fopen(name, "w+");
    if (fs == NULL) {
        fprintf(stderr, "error: cannot open %s for writing\n", name);
        perror("fopen");
        return -1;
    }

    /* Rebuild symbol table and string table */
    for (i = 0; i < sizeof(strtab) / sizeof(char *); i++) {
        if (i) symtab[i].st_name = strtab_len;
        if (strtab_len + strlen(strtab[i]) + 1 > STRTAB_MAX_SZ) {
            fprintf(stderr, "error: STRTAB too large\n"
                            "You should increase STRTAB_MAX_SZ\n");
            return -1;
        }
        strcpy(&real_strtab[strtab_len], strtab[i]);
        strtab_len += strlen(strtab[i]) + 1;
    }

    /* Rebuild section header and sh string table */
    for (i = 0; i < sizeof(shstrtab) / sizeof(char *); i++) {
        if (i) shdr[i].sh_name = shstrtab_len;
        if (shstrtab_len + strlen(shstrtab[i]) + 1 > STRTAB_MAX_SZ) {
            fprintf(stderr, "error: SHSTRTAB too large\n"
                            "You should increase STRTAB_MAX_SZ\n");
            return -1;
        }
        strcpy(&real_shstrtab[shstrtab_len], shstrtab[i]);
        shstrtab_len += strlen(shstrtab[i]) + 1;
    }

    /* Add size and offset information to section header */
    shdr[SECT_SYMTAB].sh_info = sizeof(symtab) / sizeof(Elf64_Sym);
    shdr[SECT_SYMTAB].sh_size = sizeof(symtab);
    shdr[SECT_STRTAB].sh_offset = shdr[SECT_SYMTAB].sh_offset
                                + shdr[SECT_SYMTAB].sh_size;
    shdr[SECT_STRTAB].sh_size = strtab_len;
    shdr[SECT_SHSTRTAB].sh_offset = shdr[SECT_STRTAB].sh_offset
                                  + shdr[SECT_STRTAB].sh_size;
```

```
    shdr[SECT_SHSTRTAB].sh_size = shstrtab_len;

    /* Read first segment */
    fp = fopen(EXEC_SEGMENT, "r");
    if (fp == NULL) {
        fprintf(stderr, "error: cannot open %s for reading\n"
                "You should modify EXEC_SEGMENT\n", EXEC_SEGMENT);
        perror("fopen");
        fclose(fs);
        return -1;
    }
    val = fread(rd_buf, 1, EXEC_SIZE, fp);
    fclose(fp);

    /* Overwrite section header offset */
    ehdr = (Elf64_Ehdr *)rd_buf;
    sec_hdr_offt = shdr[SECT_SHSTRTAB].sh_offset
                 + shdr[SECT_SHSTRTAB].sh_size;
    ehdr->e_shoff = (sec_hdr_offt + 0xf) & ~0xf;

    /* Write ELF header */
    fwrite(rd_buf, 1, ehdr->e_phoff, fs);

    /* Write reconstructed Program Header */
    fwrite(prog_hdr, 1, sizeof(prog_hdr), fs);

    /* Write first segment */
    fwrite(rd_buf + sizeof(prog_hdr) + ehdr->e_phoff, 1,
           val - sizeof(prog_hdr) - ehdr->e_phoff, fs);

    /* Read second segment */
    fp = fopen(DATA_SEGMENT, "r");
    if (fp == NULL) {
        fprintf(stderr, "error: cannot open %s for reading\n"
                "You should modify DATA_SEGMENT\n", DATA_SEGMENT);
        perror("fopen");
        fclose(fs);
        return -1;
    }
    val = fread(rd_buf, 1, DATA_SIZE, fp);
    fclose(fp);

    /* Write second segment */
    fwrite(rd_buf, 1, val, fs);

    /* Write symbol table */
    fwrite(symtab, 1, sizeof(symtab), fs);

    /* Write string table */
    fwrite(real_strtab, 1, strtab_len, fs);

    /* Write SH string table */
    fwrite(real_shstrtab, 1, shstrtab_len, fs);

    /* Align ELF */
    val = ((sec_hdr_offt + 0xf) & ~0xf) - sec_hdr_offt;
    memset(rd_buf, 0, val);
    fwrite(rd_buf, 1, val, fs);

    /* Write section header */
    fwrite(shdr, 1, sizeof(shdr), fs);

    fclose(fs);

    return 0;
```

```
}
```

5.3 Désassembleur

```
/*
 * Disassembler for SSTIC Challenge 2014 - part 2
 *
 * 2X VV 1X VV      MOV  rX, #VVVV (20 for MSB, 10 for LSB)
 * 3X YZ            XOR  rX, rY, rZ
 * 4X YZ            OR   rX, rY, rZ
 * 5X YZ            AND  rX, rY, rZ
 * 6X YZ            ADD  rX, rY, rZ
 * 7X YZ            SUB  rX, rY, rZ
 * 8X YZ            MUL  rX, rY, rZ
 * 9X YZ            DIV  rX, rY, rZ
 * AV VV            Jxx  +#VVV if xx corresponds to ZS flags
 * BV VV            JMP  +#VVV
 * C8 VV            SWI  #VV
 * CV VV            CALL +#VVV
 * D0 0F            RET
 * D8 0F            SYSEXIT
 * D0 00            SYSENTER
 * EX YZ            LDR rX, [rY, rZ]
 * FX YZ            STR rX, [rY, rZ]
 *
 * Compile with:
 *
 *   gcc disas.c -o disas -Wall -Wextra -pedantic -std=c99
 *
 */
#include <stdio.h>
#include <stdlib.h>

int main(int ac, char** av)
{
    FILE *fp;
    int addr = 0;
    unsigned char c[2];

    if (ac < 2) {
        fprintf(stderr, "Usage: %s file.bin [start_addr]\n", av[0]);
        return -1;
    }

    if (ac == 3)
        addr = strtol(av[2], NULL, 0);

    fp = fopen(av[1], "r");

    if (fp == NULL) {
        perror("open");
        return -1;
    }

    while (fread(c, 1, 2, fp) > 0) {
        unsigned char d[2];
        char *ptr  = NULL;
        size_t tmp = 0;
        int ins    = c[0] & 0xF0;
        int rd     = c[0] & 0xF;
        int ri1    = (c[1] & 0xF0) >> 4;
        int ri2    = c[1] & 0xF;
        int cond   = c[0] & 0xC;
```

```
        int val    = ((c[0] & 0x3) << 8) | c[1];

        if (val & 0x200)
            val = -(0x400 - val);

        switch(ins) {
            case 0x10:
                val = c[1];
                printf("%04X: MOVL\tr%02d, #0x%04X\n", addr, rd, val);
                break;
            case 0x20:
                val = c[1];
                tmp = fread(d, 1, 2, fp);
                if (tmp == 2 && ((d[0] & 0xF0) == 0x10) && ((d[0] & 0xF) == rd)) {
                    val <<= 8;
                    val |= d[1];
                    printf("%04X: MOV \tr%02d, #0x%04X\n", addr, rd, val);
                    addr += 2;
                } else {
                    if (tmp == 2) fseek(fp, -2, SEEK_CUR);
                    printf("%04X: MOVH\tr%02d, #0x%04X\n", addr, rd, val);
                }
                break;
            case 0x30:
                printf("%04X: XOR \tr%02d, r%02d, r%02d\n", addr, rd, ri1, ri2);
                break;
            case 0x40:
                printf("%04X: OR  \tr%02d, r%02d, r%02d\n", addr, rd, ri1, ri2);
                break;
            case 0x50:
                printf("%04X: AND \tr%02d, r%02d, r%02d\n", addr, rd, ri1, ri2);
                break;
            case 0x60:
                printf("%04X: ADD \tr%02d, r%02d, r%02d\n", addr, rd, ri1, ri2);
                break;
            case 0x70:
                printf("%04X: SUB \tr%02d, r%02d, r%02d\n", addr, rd, ri1, ri2);
                break;
            case 0x80:
                printf("%04X: MUL \tr%02d, r%02d, r%02d\n", addr, rd, ri1, ri2);
                break;
            case 0x90:
                printf("%04X: DIV \tr%02d, r%02d, r%02d\n", addr, rd, ri1, ri2);
                break;
            case 0xA0:
                if (cond == 0) ptr = "EQ";
                else if (cond == 0x8) ptr = "L ";
                else if (cond == 0x4) ptr = "G ";
                else if (cond == 0xC) ptr = "GE";
                printf("%04X: J%s \t#0x%04X\n", addr, ptr, addr + 2 + val);
                break;
            case 0xB0:
                printf("%04X: JMP \t#0x%04X\n", addr, addr + 2 + val);
                break;
            case 0xC0:
                if (c[0] & 0x8) printf("%04X: SWI \t#0x%02X\n", addr, c[1]);
                else printf("%04X: CALL\t#0x%04X\n", addr, addr + 2 + val);
                break;
            case 0xD0:
                if (c[1] == 0x0F) printf("%04X: RET\n", addr);
                else if ((c[0] & 0xF) == 0x8) printf("%04X: SYSEXIT\n", addr);
                else if ((c[0] & 0xF) == 0x0 && c[1] == 0x0) printf("%04X:
SYSENTER\n", addr);
                else printf("%04X: .ins %02X%02X\n", addr, c[0], c[1]);
                break;
```

```
            case 0xE0:
                printf("%04X: LDR \tr%02d, [r%02d, r%02d]\n", addr, rd, ri1, ri2);
                break;
            case 0xF0:
                printf("%04X: STR \tr%02d, [r%02d, r%02d]\n", addr, rd, ri1, ri2);
                break;
            default:
                printf("%04X: .ins\t%02X%02X\n", addr, c[0], c[1]);
                break;
        }
        addr += 2;
    }

    fclose(fp);

    return 0;
}
```