Challenge SSTIC 2014 : solution

Julien Perrot
@nieluj

14 juin 2014
version 1.1

Résumé

Ce document présente une démarche possible pour résoudre le challenge SSTIC 2014. Comme pour les années précédentes, la validation du challenge nécessite de pouvoir extraire, depuis un fichier téléchargé sur le site de la conférence, une adresse email de la forme `@sstic.org`.

La première étape du challenge consiste à analyser une trace décrivant des échanges de messages USB. L'étude de ces messages permet d'identifier une session ADB (Android Debug Bridge) au sein de laquelle s'effectue un transfert d'un fichier. La récupération du fichier transféré permet de continuer le challenge.

Le fichier obtenu depuis la trace USB est un binaire ELF et d'architecture ARM64. Lorsque ce fichier est exécuté, une clé de 16 octets est demandée et une tentative de déchiffrement est effectuée.

L'examen du binaire permet de découvrir l'implémentation d'une machine virtuelle et le jeu d'instructions supporté par celle-ci. Il est alors possible de désassembler le programme exécuté par la machine virtuelle pour comprendre l'algorithme de chiffrement utilisé. Il s'agit en fait d'un LFSR (Linear Feedback Shift Register). Des conditions sur la fin des données déchiffrées permettent de déterminer un état intermédiaire de celui-ci. Il suffit alors de l'inverser pour obtenir la clé attendue.

Le fichier déchiffré, avec la clé correcte, par la machine virtuelle est une archive au format Zip contenant un programme et un script pour envoyer et exécuter ce programme sur un micro-contrôleur distant. L'injection de fautes dans le programme déclenche des exceptions qui décrivent l'état du micro-contrôleur. En multipliant de cette façon les tests, le jeu d'instruction utilisé est déterminé et il est possible de désassembler le programme. La découverte d'un appel système `read` permet d'obtenir le code exécuté par le micro-contrôleur en mode kernel. En désassemblant le kernel, une vulnérabilité est découverte qui constitue une primitive d'écriture arbitraire en espace kernel. Un programme exploitant cette vulnérabilité peut alors accéder à la zone mémoire « secrète » du micro-contrôleur qui contient l'adresse email de validation du challenge.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Étude de la trace USB

1.1 Découverte

La première partie du challenge consiste à analyser une trace USB téléchargeable à l'adresse http://static.sstic.org/challenge2014/usbtrace.xz.

Avant tout, il est nécessaire de vérifier l'intégrité du fichier téléchargé :

```
$ wget --quiet http://static.sstic.org/challenge2014/usbtrace.xz
$ md5sum usbtrace.xz
3783cd32d09bda669c189f3f874794bf  usbtrace.xz
$ file usbtrace.xz
usbtrace.xz: XZ compressed data
```

On retrouve bien la même empreinte cryptographique que sur la page du challenge. Le fichier alors obtenu peut être décompressé avec le programme 7-Zip :

```
$ 7z x usbtrace.xz

7-Zip [64] 9.20  Copyright (c) 1999-2010 Igor Pavlov  2010-11-18
p7zip Version 9.20 (locale=fr_FR.UTF-8,Utf16=on,HugeFiles=on,4 CPUs)

Processing archive: usbtrace.xz

Extracting  usbtrace

Everything is Ok

Size:       353537
Compressed: 97192
```

Le fichier décompressé contient quelques indications pour poursuivre la résolution du challenge :

```
$ head -n 10 usbtrace
Date: Thu, 17 Apr 2015 00:40:34 +0200
To: <challenge2014@sstic.org>
Subject: Trace USB

Bonjour,

voici une trace USB enregistrée en branchant mon nouveau téléphone Android sur mon ordinateur
personnel air-gapped. Je suspecte un malware de transiter sur mon téléphone. Pouvez-vous voir
de quoi il en retourne ?
```

```
--
```

La suite du fichier est constituée de lignes similaires à celles présentées ci-dessous :

```
$ tail -n +12 usbtrace | head -n 12
ffff8804ff109d80 1765779215 C Ii:2:005:1 0:8 8 = 00000000 00000000
ffff8804ff109d80 1765779244 S Ii:2:005:1 -115:8 8 <
ffff88043ac600c0 1765809097 S Bo:2:008:3 -115 24 = 4f50454e fd010000 00000000 09000000 1f030000
    b0afbab1
ffff88043ac600c0 1765809154 C Bo:2:008:3 0 24 >
ffff88043ac60300 1765809224 S Bo:2:008:3 -115 9 = 7368656c 6c3a6964 00
ffff88043ac60300 1765809279 C Bo:2:008:3 0 9 >
ffff8804e285ec00 1765810255 C Bi:2:008:5 0 24 = 4f4b4159 fb000000 fd010000 00000000 00000000
    b0b4bea6
ffff8800d0fbf180 1765810282 S Bi:2:008:5 -115 24 <
ffff8800d0fbf180 1765815007 C Bi:2:008:5 0 24 = 57525445 fb000000 fd010000 d3000000 05410000
    a8adabba
ffff8800d0fbf180 1765815053 S Bi:2:008:5 -115 211 <
ffff8800d0fbf180 1765815140 C Bi:2:008:5 0 211 = 7569643d 32303030 28736865 6c6c2920 6769643d
    32303030 28736865 6c6c2920 67726f75 70733d31 30303328 67726170 68696373 292c3130 30342869
    6e707574 292c3130 3037286c 6f67292c 31303039 286d6f75 6e74292c 31303131 28616462 292c3130
    31352873 64636172 645f7277 292c3130 32382873 64636172 645f7229 2c333030 31286e65 745f6274
    5f61646d 696e292c 33303032 286e6574 5f627429 2c333030 3328696e 6574292c 33303036 286e6574
    5f62775f 73746174 73292063 6f6e7465 78743d75 3a723a73 68656c6c 3a7330
ffff8800d0fbf180 1765815196 S Bi:2:008:5 -115 24 <
```

1.2 Analyse des évènements USB

Une recherche Google avec les mots clés « usb trace `Ii:2:005:1` » retourne une trace USB au format similaire à l'adresse http://permalink.gmane.org/gmane.linux.usb.general/61635. Le message posté à cette liste de diffusion permet d'identifier le format de la trace USB, à savoir le format usbmon.

Ce format est documenté à l'adresse https://www.kernel.org/doc/Documentation/usb/usbmon.txt. Le terme usbmon désigne un mécanisme implémenté en standard sous Linux permettant d'obtenir des traces USB, sous forme binaire ou textuelle. Dans notre cas, il s'agit bien évidemment d'une trace au format textuel.

D'après le document cité précédemment, une ligne d'une trace est composée des champs suivants :

- un tag URB (USB Request Block), généralement l'adresse dans le noyau de la structure URB correspondante ;
- une information d'horodatage ;
- le type d'évènement (S pour « submission », C pour « callback », E pour « submission error ») ;
- une adresse composée de quatre champs :
  - le type et la direction de l'URB (`C` pour « control », `Z` pour les échanges isochrones, `I` pour les interruptions et `B` pour les échanges « bulk »),
  - le numéro du bus,
  - l'adresse du périphérique,
  - le numéro du correspondant ;
- le status de l'URB, soit une lettre ou plusieurs nombres séparés par le caractère `:` ;
- la taille des données échangées ;
- le caractère `=` lors que des données échangées ;
- enfin les données sous forme hexadécimale.

A partir de ces informations, il est possible de développer un script Ruby pour instancier un objet pour chaque ligne :

```ruby
class Event
  def initialize(urb_tag, timestamp, event_type, address, urb_status,
```

```
    data_length, data_tag)
    [...]
  end

  def data_words=(words)
    @data = words.map {|x| [x].pack("H*")}.join
  end

  def is_data?
    @data_tag == "="
  end

  def is_bulk?
    @urb_type == "B"
  end
end

class EventParser
  class << self
    def parse_line(line)
      raise unless line =~ /^ffff/
      comps = line.strip.split(/\s+/)
      urb_tag = comps.shift
      timestamp = comps.shift
      event_type = comps.shift
      address = comps.shift
      urb_status = comps.shift
      data_length = comps.shift
      data_tag = comps.shift

      e = Event.new(urb_tag, timestamp, event_type, address, urb_status,
                    data_length, data_tag)

      if e.is_data? then
        e.data_words = comps
      end
      return e
    end
  end
end
```

Sur un bus USB, seuls les évènements de type « bulk » contiennent les données utiles transférées entre deux périphériques. L'analyseur de la trace ne va donc sélectionner que les évènements de ce type.

```
input = ARGV.shift

File.open(input, "r").each_line do |line|
  next unless line =~ /^ffff/

  e = USB::EventParser.parse_line(line)
  next unless e.is_data? and e.is_bulk?

  puts e.data.inspect
end
```

Le résultat pour les premiers évènements USB donne :

```
$ ./parse-usbmon.rb usbtrace
"OPEN\xFD\x01\x00\x00\x00\x00\x00\x00\t\x00\x00\x00\x1F\x03\x00\x00\xB0\xAF\xBA\xB1"
"shell:id\x00"
"OKAY\xFB\x00\x00\x00\xFD\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\xB0\xB4\xBE\xA6"
"WRTE\xFB\x00\x00\x00\xFD\x01\x00\x00\xD3\x00\x00\x00\x05A\x00\x00\xA8\xAD\xAB\xBA"
"uid=2000(shell) gid=2000(shell) groups=1003(graphics),1004(input),1007(log),1009(mount),
  1011(adb),1015(sdcard_rw),1028(sdcard_r),3001(net_bt_admin),3002(net_bt),3003(inet),
  3006(net_bw_stats) context=u:r:shell:s0"
```

1.3 Analyse des messages ADB

Certaines lignes obtenues précédemment semblent composées d'une commande sur quatre caractères suivi d'un ensemble de données. Une recherche Google sur les termes `OPEN OKAY WRTE` retourne de nombreuses références à ADB (Android Debug Bridge), un protocole permettant de communiquer avec un ordiphone Android.

La documentation de ce protocole est présente dans les sources d'Android disponibles sur Github [1]. En particulier, la structure d'un message `adb` est précisée :

```
struct message {
    unsigned command;       /* command identifier constant      */
    unsigned arg0;          /* first argument                   */
    unsigned arg1;          /* second argument                  */
    unsigned data_length;   /* length of payload (0 is allowed) */
    unsigned data_crc32;    /* crc32 of data payload            */
    unsigned magic;         /* command ^ 0xffffffff             */
};
```

Cela permet d'ajouter une nouvelle méthode `to_adb_message` à la classe `USB::Event` de l'analyseur de trace :

```
class AdbMessage

  def initialize(command, arg0, arg1, data_length, data_check, magic, data)
    [...]
  end

  def got_data?
    @data_length == @data.size
  end

  def to_s
    s = "[ADB] #@command #@arg0 #@arg1 #@data_length"
    case @command
    when "OPEN"
      s << " = " << @data
    when "WRTE"
      s << " = " << @data[0, 32].inspect
    end
    return s
  end
end

class Event
  [...]

  def to_adb_message
    return nil unless is_data? and is_bulk?

    command = @data[0, 4]
    arg0, arg1, data_length, data_crc32, magic = *@data[4, 20].unpack('L5')
    data = @data[24..-1]
    message = AdbMessage.new(command, arg0, arg1, data_length, data_crc32, magic, data)
  end
end
```

La documentation précise également les commandes possibles :

1. https://github.com/android/platform_system_core/blob/master/adb/protocol.txt

```
#define A_SYNC 0x434e5953
#define A_CNXN 0x4e584e43
#define A_AUTH 0x48545541
#define A_OPEN 0x4e45504f
#define A_OKAY 0x59414b4f
#define A_CLSE 0x45534c43
#define A_WRTE 0x45545257
```

La boucle principale de l'analyseur est alors mise à jour pour instancier des messages ADB et reconstituer les données des messages à partir des évènements USB :

```
valid_adb_commands = %w{ SYNC CNXN AUTH OPEN OKAY CLSE WRTE }

input = ARGV.shift

adb_messages = []
current_message = nil
File.open(input, "r").each_line do |line|
  next unless line =~ /^ffff/

  e = USB::EventParser.parse_line(line)
  next unless e.is_data? and e.is_bulk?

  if current_message then
    current_message.data << e.data
  else
    if e.data_length >= 4 then
      if valid_adb_commands.include?(e.data[0, 4]) then
        current_message = e.to_adb_message
      end
    end
  end

  if current_message and current_message.got_data? then
    adb_messages << current_message
    puts current_message
    current_message = nil
  end
end
```

Ce traitement permet d'obtenir la séquence des messages ADB échangés. L'exécution de l'analyseur retourne le résultat suivant :

```
$ ./parse-usbmon.rb usbtrace
[ADB] OPEN 509 0 9 = shell:id
[ADB] OKAY 251 509 0
[ADB] WRTE 251 509 211 = "uid=2000(shell) gid=2000(shell) "
[ADB] OKAY 509 251 0
[ADB] WRTE 251 509 2 = "\r\n"
[ADB] CLSE 251 509 0
[ADB] OKAY 509 251 0
[ADB] CLSE 0 251 0
[ADB] OPEN 511 0 15 = shell:uname -a
[ADB] OKAY 252 511 0
[...]
```

1.4 Reconstitution de la session ADB

L'analyse des messages ADB permet d'identifier une séquence curieuse :

```
$ ./parse-usbmon.rb usbtrace
[...]
[ADB] OPEN 519 0 6 = sync:
[ADB] OKAY 256 519 0
[ADB] WRTE 519 256 8 = "STAT\e\x00\x00\x00"
[ADB] OKAY 256 519 0
[ADB] WRTE 519 256 27 = "/data/local/tmp/badbios.bin"
[ADB] OKAY 256 519 0
[ADB] WRTE 256 519 16 = "STAT\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
[ADB] OKAY 519 256 0
[ADB] WRTE 519 256 8 = "SEND!\x00\x00\x00"
[ADB] OKAY 256 519 0
[ADB] WRTE 519 256 4096 = "/data/local/tmp/badbios.bin,3326" ...
[ADB] OKAY 256 519 0
[ADB] WRTE 519 256 4096 = "\x00\x00\x00\x00\x00\x00\x00[...]x00\x00\x00\x00" ...
[ADB] OKAY 256 519 0
[ADB] WRTE 519 256 4096 = "\xDA\xF7\xE4\x94b]v\x8F\x12g[...]93\x84`\xF9\xF6;\xE1=" ...
[ADB] OKAY 256 519 0
[ADB] WRTE 519 256 4096 = "\x01\x9C\xC0\xD2]\x06\xABm\xDD[...]p/1\xA0+7\xA4t" ...
```

Ces messages semblent correspondre à un transfert d'un fichier entre l'ordinateur et l'ordiphone. Au sein même des messages ADB, on retrouve de nouvelles commandes, telles que `STAT` et `SEND`.

Ces commandes correspondent au protocole `sync` décrit dans le fichier `SYNC.TXT`[2] présent dans les sources d'Android. Le fichier est également disponible à l'annexe A.1.

Ce protocole permet d'effectuer les opérations suivantes :

— `LIST` : obtenir la liste des fichiers dans un répertoire ;
— `SEND` : envoyer un fichier vers un périphérique ;
— `RECV` : récupérer un fichier depuis un périphérique ;
— `STAT` : obtenir des informations sur un fichier.

La documentation spécifie que la commande `SEND` fonctionne de la manière suivante :

— le nom du fichier est envoyé, suivi d'une valeur décrivant les permissions du fichier ;
— une série de commandes `DATA` est envoyée, chaque commande précisant la taille des données transférées ;
— quand toutes les données sont transférées, une commande `DONE` est envoyée.

Cela permet de reconstituer le fichier envoyé à l'aide de la méthode Ruby ci-dessous :

```ruby
def handle_data(binary)
  send_result, send_fname = nil, nil
  scanner = BinaryScanner.new(binary)

  while not scanner.empty?
    case cmd = scanner.scan_data(4)
      when "STAT"
        [...]
      when "DONE"
        dw = scanner.scan_dword
        if dw != 0 and send_fname then
          puts "=> DONE, writing #{send_result.size} bytes to #{send_fname}, mtime = #{Time.at(dw)}"
          File.open(send_fname, "wb") { |f| f.write send_result }
          send_fname, send_result = nil, nil
        end
      when "LIST"
        [...]
      when "SEND"
        send_result = ""
        ssize = scanner.scan_dword
```

2. https://github.com/android/platform_system_core/blob/master/adb/SYNC.TXT

```
        s = scanner.scan_data(ssize)
        puts "=> SEND #{s}"
        send_fname = File.basename(s.split(',').first)
      when "DATA"
        size = scanner.scan_dword
        puts "=> DATA #{size}"
        send_result << scanner.scan_data(size)
      when "OKAY", "QUIT"
        [...]
      else
        raise "Unknown command #{cmd}"
      end
    end # case cmd
  end # while

end # method handle_data
```

Le script `parse-usbmon.rb` est modifié pour appeler cette méthode à la fin d'une session `sync` :

```
binary, shell_cmd = "", false
adb_messages.each do |m|

  case m.command
  when "OPEN"
    if m.data[0,5] == "sync:" then
      binary = ""
    elsif m.data[0,5] == "shell" then
      binary = ""
      puts "=> #{m.data.strip}"
      shell_cmd = true
    end
  when "CLSE"
    if shell_cmd then
      puts binary.strip
      shell_cmd = false
    else
      USB::AdbMessage.handle_data(binary) unless binary.empty?
    end
    binary = ""
  when "WRTE"
    binary << m.data
  end
end
```

Le script final est disponible à l'annexe A.1. Le résultat obtenu est alors le suivant :

```
$ ./parse-usbmon.rb usbtrace
=> shell:id
uid=2000(shell) gid=2000(shell) groups=1003(graphics),1004(input),1007(log),1009(mount),1011(adb),1015(sdcard_rw),
  1028(sdcard_r),3001(net_bt_admin),3002(net_bt),3003(inet),3006(net_bw_stats) context=u:r:shell:s0
=> shell:uname -a
Linux localhost 4.1.0-g4e972ee #1 SMP PREEMPT Mon Feb 24 21:16:40 PST 2015 armv8l GNU/Linux

=> LIST "/sdcard/"
40771   4096    2014-04-17 11:53:13 +0200 .
40771   4096    2013-01-01 01:01:18 +0100 ..
40770   4096    2014-01-30 17:08:22 +0100 Samsung
40771   4096    2013-01-01 01:01:20 +0100 Android
40770   4096    2014-02-20 13:08:54 +0100 .face
40770   4096    2013-01-01 01:01:19 +0100 Music
40770   4096    2013-01-01 01:01:19 +0100 Podcasts
40770   4096    2013-01-01 01:01:33 +0100 Ringtones
40770   4096    2013-01-01 01:01:19 +0100 Alarms
40770   4096    2013-01-01 01:01:19 +0100 Notifications
40770   4096    2014-02-20 13:06:52 +0100 Pictures
```

```
40770   4096    2013-01-01 01:01:19 +0100 Movies
40770   4096    2014-04-17 11:58:18 +0200 Download
40770   4096    2013-01-11 08:50:03 +0100 DCIM
40770   4096    2014-04-17 11:58:18 +0200 Documents
40770   4096    2013-01-01 01:01:24 +0100 .SPenSDK30
100660  15      2013-01-01 01:01:30 +0100 .enref
40770   4096    2014-01-29 14:23:15 +0100 Nearby
40770   4096    2014-01-29 14:25:03 +0100 Playlists
100660  0       2014-01-29 16:40:48 +0100 .pla
40770   4096    2014-02-21 10:12:52 +0100 .estrongs
40770   4096    2014-02-20 13:05:23 +0100 backups
40770   4096    2014-02-25 17:39:49 +0100 clockworkmod
40770   4096    2014-02-25 17:42:00 +0100 CyanogenMod
40770   4096    2013-01-06 22:12:45 +0100 mmc1

=> LIST "/sdcard/Documents/"
40770   4096    2014-04-17 11:58:18 +0200 .
40771   4096    2014-04-17 11:53:13 +0200 ..
100660  229376  2014-03-12 16:42:15 +0100 CSW-2014-Hacking-9.11_uncensored.pdf
100660  44032   2014-03-12 16:51:01 +0100 NATO_Cosmic_Top_Secret.gpg

=> LIST "/data/local/tmp"
40771   16384   2014-04-17 13:11:23 +0200 .
40751   4096    1970-01-30 00:55:29 +0100 ..

=> STAT "/data/local/tmp/badbios.bin": mode = 0, size = 0, time = 1970-01-01 01:00:00 +0100
=> SEND /data/local/tmp/badbios.bin,33261
=> DATA 65536
=> DATA 12464
=> DONE, writing 78000 bytes to badbios.bin, mtime = 2014-04-17 13:01:02 +0200
=> shell:chmod 777 /data/local/tmp/badbios.bin

=> LIST "/data/local/tmp"
40771   16384   2014-04-17 13:11:25 +0200 .
40751   4096    1970-01-30 00:55:29 +0100 ..
100777  78000   2014-04-17 13:01:02 +0200 badbios.bin
```

L'exécution du script a permis d'extraire le fichier `badbios.bin` de la trace USB.

```
$ md5sum badbios.bin
b6097e562cb80a20dfb67a4833b1988a  badbios.bin
```

L'analyse de ce fichier fait l'objet du chapitre suivant.

Chapitre 2

Analyse du fichier `badbios.bin`

2.1 Découverte

Le fichier obtenu est un binaire aarch64, comme le montre la commande ci-dessous :

```
$ file badbios.bin
badbios.bin: ELF 64-bit LSB  executable, ARM aarch64, version 1 (SYSV), statically linked,
  stripped
```

Les symboles de débogage ont été supprimés du binaire, ce qui va complexifier l'analyse. Les sections du binaire sont présentées ci-dessous :

```
$ objdump -h badbios.bin
badbios.bin:     format de fichier elf64-little

Sections:
Idx Nom          Taille    VMA               LMA               Fich off  Algn
  0 .text         0000048c  000000000001010c  000000000001010c  0000010c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00000040  0000000000010598  0000000000010598  00000598  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data         00011f50  0000000000021000  0000000000021000  00001000  2**3
                  CONTENTS, ALLOC, LOAD, DATA
```

Il est possible de remarquer que la section `.text`, qui contient le code exécutable, est relativement petite (`0x48c = 1164` octets) par rapport à la section `.data` (`0x11f50 = 73552` octets).

2.2 Analyse du programme

L'émulation en mode utilisateur de QEMU [1] permet de lancer le binaire, comme présenté ci-dessous.

```
$ sudo apt-get install qemu-user
$ chmod +x badbios.bin
$ qemu-aarch64 badbios.bin
:: Please enter the decryption key: AAAA
   Wrong key format.
```

Le programme semble attendre une clé de déchiffrement dans un certain format. Pour obtenir plus d'information sur le binaire, il est possible de le lancer sous QEMU en demandant à obtenir la liste des appels systèmes.

1. à condition de disposer de la version 2.0.0 de QEMU

```
$ qemu-aarch64 -strace badbios.bin 2> /tmp/strace.txt
:: Please enter the decryption key: AAAA
$ cat /tmp/strace.txt
2836 mmap(0x0000000000400000,12288,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS|MAP_FIXED,0,0)
  = 0x0000000000400000
2836 mprotect(0x0000000000400000,12288,PROT_EXEC|PROT_READ) = 0
2836 mmap(0x0000000000500000,69632,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS|MAP_FIXED,0,0)
  = 0x0000000000500000
2836 mprotect(0x0000000000500000,69632,PROT_READ|PROT_WRITE) = 0
2836 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) = 0x0000004000801000
2836 mmap(NULL,65536,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) = 0x0000004000802000
2836 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) = 0x0000004000812000
2836 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) = 0x0000004000813000
2836 write(1,0x813000,36) = 36
2836 munmap(0x0000004000813000,36) = 0
2836 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) = 0x0000004000814000
2836 read(0,0x814000,16) = 5
2836 munmap(0x0000004000814000,16) = 0
2836 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) = 0x0000004000815000
2836 write(2,0x815000,21)   Wrong key format.
 = 21
2836 munmap(0x0000004000815000,21) = 0
2836 exit_group(0)
```

On remarque qu'avant d'afficher la chaîne `Wrong key format`, le programme exécute l'appel système `read` avec une longueur de 16. On peut supposer à ce stade que le programme attends alors une clé de 16 octets. Un second test est alors effectué avec la longueur attendue.

```
$ qemu-aarch64 -strace badbios.bin 2> /tmp/strace.txt
:: Please enter the decryption key: AAAAAAAAAAAAAAAA
:: Trying to decrypt payload...
$ cat /tmp/strace
3541 mmap(0x0000000000400000,12288,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS|MAP_FIXED,0,0)
  = 0x0000000000400000
3541 mprotect(0x0000000000400000,12288,PROT_EXEC|PROT_READ) = 0
3541 mmap(0x0000000000500000,69632,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS|MAP_FIXED,0,0)
  = 0x0000000000500000
3541 mprotect(0x0000000000500000,69632,PROT_READ|PROT_WRITE) = 0
3541 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) = 0x0000004000801000
3541 mmap(NULL,65536,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) = 0x0000004000802000
3541 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) = 0x0000004000812000
3541 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) = 0x0000004000813000
3541 write(1,0x813000,36) = 36
3541 munmap(0x0000004000813000,36) = 0
3541 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) = 0x0000004000814000
3541 read(0,0x814000,16) = 16
3541 munmap(0x0000004000814000,16) = 0
3541 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) = 0x0000004000815000
3541 write(1,0x815000,32) = 32
3541 munmap(0x0000004000815000,32) = 0
3541 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) = 0x0000004000816000
3541 write(2,0x816000,20)   Invalid padding.
 = 20
3541 munmap(0x0000004000816000,20) = 0
3541 exit_group(0)
```

Le comportement du programme est cette fois différent. La chaîne `:: Trying to decrypt payload...` est affichée, puis le programme s'arrête avec le message d'erreur `Invalid padding.`.

2.2.1 Analyse du point d'entrée

Pour aller plus loin dans l'analyse, il va être nécessaire de désassembler le binaire.

```
$ sudo apt-get install binutils-aarch64-linux-gnu
$ /usr/bin/aarch64-linux-gnu-objdump -d badbios.bin

badbios:     file format elf64-littleaarch64

Disassembly of section .text:

000000000001010c <.text>:
   1010c:       a9b97bfd        stp     x29, x30, [sp,#-112]!
   10110:       910003fd        mov     x29, sp
   10114:       b0000089        adrp    x9, 0x21000
   10118:       91000122        add     x2, x9, #0x0
   1011c:       b9006ba0        str     w0, [x29,#104]
   10120:       f9400440        ldr     x0, [x2,#8]
   10124:       b9400043        ldr     w3, [x2]
   10128:       a9046bf9        stp     x25, x26, [sp,#64]
   1012c:       a90153f3        stp     x19, x20, [sp,#16]
   10130:       a9025bf5        stp     x21, x22, [sp,#32]
   10134:       a90363f7        stp     x23, x24, [sp,#48]
   10138:       f9002bfb        str     x27, [sp,#80]
   1013c:       f90033a0        str     x0, [x29,#96]
[...]
$ readelf -h badbios.bin | grep "Entry point"
  Entry point address:               0x102cc
```

Pour identifier les appels de fonction, il est possible de rechercher les instructions `bl` :

```
$ /usr/bin/aarch64-linux-gnu-objdump -d badbios.bin|grep bl
Disassembly of section .text:
   1029c:       9400001a        bl      0x10304
   102c0:       d63f0040        blr     x2
   102e8:       97ffff89        bl      0x1010c
```

Curieusement, l'instruction à l'adresse `0x102c0` appelle la fonction à l'adresse contenue dans le registre `r2`.

Le point d'entrée étant `0x102cc`, l'analyse commence donc à cette adresse. Le code correspondant est présenté ci-dessous :

```
   102cc:    d280001e     mov    x30, #0x0                      // #0
   102d0:    910003fd     mov    x29, sp
   102d4:    f94003e0     ldr    x0, [sp]
   102d8:    910023e1     add    x1, sp, #0x8
   102dc:    14000001     b    0x102e0
   102e0:    a9bf7bfd     stp    x29, x30, [sp,#-16]!
   102e4:    910003fd     mov    x29, sp
   102e8:    97ffff89     bl    0x1010c
   102ec:    93407c01     sxtw    x1, w0
   102f0:    aa0103e0     mov    x0, x1
   102f4:    d2800bc8     mov    x8, #0x5e                      // #94
   102f8:    d4000001     svc    #0x0
   102fc:    aa0003e1     mov    x1, x0
   10300:    14000000     b    0x10300
```

Le code appelle rapidement la fonction à l'adresse `0x1010c`. Avant de poursuivre l'analyse en s'intéressant à cette fonction, l'instruction `svc #0x0` à l'adresse `0x102f8` mérite d'être étudiée.

Une recherche Google sur les mots clés « ARM linux svc » permet d'obtenir le document `lcna_co2012_marinas.pdf` [2] décrivant l'implémentation de Linux sur les architectures ARM 64-bit.

On peut y apprendre que l'instruction `svc` est utilisée par Linux pour la gestion des appels systèmes. Le numéro de l'appel système est alors stocké dans le registre `x8` et les paramètres sont passés dans les registres `x0`, `x1`, `x2`, etc.

Dans le cas présent, le numéro de l'appel système est 94. La correspondance peut être trouvée dans le fichier `unistd.h`.

```
$ grep 94 /usr/include/asm-generic/unistd.h
#define __NR_exit_group 94
#define __NR_shmget 194
```

Le programme va donc simplement appeler la fonction à l'adresse `0x1010c` puis appeler l'appel système `exit_group` en passant en paramètre la valeur de retour de l'appel de la fonction.

L'analyse du programme continue ensuite à l'adresse `0x1010c`.

2.2.2 Analyse de la fonction `sub_1010c`

La fonction `sub_1010c` commence par le code ci-dessous qui se contente d'appeler l'appel système `mmap` :

```
000000000001010c <.text>:
   1010c:    a9b97bfd     stp    x29, x30, [sp,#-112]!
   10110:    910003fd     mov    x29, sp
   10114:    b0000089     adrp    x9, 0x21000                 // x9 = 0x21000 (section .data)
   10118:    91000122     add    x2, x9, #0x0                // x2 = 0x21000
   1011c:    b9006ba0     str    w0, [x29,#104]              // sauvegarde argc sur la pile
   10120:    f9400440     ldr    x0, [x2,#8]                 // x0 = *(0x21008) = 0x400514
   10124:    b9400043     ldr    w3, [x2]                    // w2 = *(0x21000) = 0x02
   [...]
   1013c:    f90033a0     str    x0, [x29,#96]               // sauvegarde 0x400514 sur la pile
   10140:    aa0103f9     mov    x25, x1                     // x25 = argv
   10144:    34000b23     cbz    w3, 0x102a8                 //
   10148:    f9400c55     ldr    x21, [x2,#24]               // x21 = *(0x21018) = 0x2c08
   1014c:    f9400846     ldr    x6, [x2,#16]                // x6 = *(0x21000) = 0x400000
   10150:    913ffeb5     add    x21, x21, #0xfff            //
   10154:    9274ceb5     and    x21, x21, #0xfffffffffffff000 // x21 = 0x3000 (arrondit au multiple de 4096 supérieur)
   10158:    d2800007     mov    x7, #0x0                    // x7 = 0
   1015c:    d2800654     mov    x20, #0x32                  // x20 = 50
   10160:    d280006a     mov    x10, #0x3                   // x10 = 3
   10164:    aa0703e5     mov    x5, x7                      // x5 = 0
   10168:    aa0703e4     mov    x4, x7                      // x4 = 0
   1016c:    aa1403e3     mov    x3, x20                     // x3 = 50
   10170:    aa0a03e2     mov    x2, x10                     // x2 = 3
   10174:    aa1503e1     mov    x1, x21                     // x1 = 0x3000
   10178:    aa0603e0     mov    x0, x6                      // x0 = 0x400000
   1017c:    d2801bc8     mov    x8, #0xde                   // #222
   10180:    d4000001     svc    #0x0                        // mmap(0x400000, 0x3000, 3, 50, 0, 0)
   10184:    aa0003f4     mov    x20, x0                     //
   10188:    eb06029f     cmp    x20, x6                     // compare le résultat du mmap à l'adresse demandée
   1018c:    aa1403e1     mov    x1, x20                     //
   10190:    54000140     b.eq    0x101b8                    // saute si égal
   10194:    52800037     mov    w23, #0x1                   // sinon, sort en retournant 1
```

Certaines valeurs sont directement lues depuis la section `.data` dont voici un extrait :

```
$ /usr/bin/aarch64-linux-gnu-objdump -s -j .data badbios.bin|head -n 15
```

2. http://events.linuxfoundation.org/images/stories/pdf/lcna_co2012_marinas.pdf

```
badbios.bin:     file format elf64-littleaarch64

Contents of section .data:
 21000 02000000 00000000 14054000 00000000  ..........@.....
 21010 00004000 00000000 082c0000 00000000  ..@......,......
 21020 c8100300 00000000 841e0000 00000000  ................
 21030 05000000 01000000 00005000 00000000  ..........P.....
 21040 40000100 00000000 b0100200 00000000  @...............
 21050 18000100 00000000 03000000 00000000  ................
 21060 00000000 00000000 00000000 00000000  ................
 21070 00000000 00000000 00000000 00000000  ................
 21080 00000000 00000000 00000000 00000000  ................
 21090 00000000 00000000 00000000 00000000  ................
 210a0 00000000 00000000 00000000 00000000  ................
```

L'appel à `mmap` correspond à celui observé lors de l'exécution du programme avec QEMU :

— l'adresse demandée, `0x400000`, est la même ;
— la taille demandée, `0x3000`, est également identique ;
— `PROT_READ|PROT_WRITE` correspond à 3 ;
— `MAP_PRIVATE|MAP_ANONYMOUS|MAP_FIXED` correspond à 50 ;
— enfin les deux derniers paramètres valent 0.

Si le résultat de `mmap` correspond à l'adresse demandée, l'exécution continue alors à l'adresse `0x101b8` dont le code est présenté ci-dessous :

```
   101b8:    9100012a     add    x10, x9, #0x0                 // x10 = 0x21000 (section .data)
   101bc:    91004156     add    x22, x10, #0x10               // x22 = 0x21010
   101c0:    b94026c0     ldr    w0, [x22,#36]                 // w0 = *(0x21034) = 1
   101c4:    9100e153     add    x19, x10, #0x38               // x19 = 0x21038
   101c8:    52800018     mov    w24, #0x0                     // w24 = #0
   101cc:    aa0a03f7     mov    x23, x10                      // x23 = 0x21000
   101d0:    d280007b     mov    x27, #0x3                     // x27 = 3
   101d4:    d280065a     mov    x26, #0x32                    // x26 = 50
   101d8:    350005c0     cbnz    w0, 0x10290                  // saute à 0x10290 si w0 != 0
   [...]
   10290:    f9400ac0     ldr    x0, [x22,#16]                 // x0 = *(0x21020) = 0x310c8
   10294:    b9401ac2     ldr    w2, [x22,#24]                 // w2 = *(0x21028) = 0x1e84
   10298:    2a1503e3     mov    w3, w21                       // w3 = 0x3000
   1029c:    9400001a     bl    0x10304                        // w0 = sub_10304(0x310c8, 0x400000, 0x1e84, 0x3000)
   102a0:    35fffba0     cbnz    w0, 0x10214                  // saute à 0x10214 si w0 != 0
   102a4:    17ffffbc     b    0x10194                         // retourne 1
```

Le code précédent va simplement lire certaines valeurs dans la section `.data` puis appeler la fonction `sub_10304` avec les paramètres suivants :

— la valeur `0x310c8`, lue à l'adresse `0x21020` ;
— l'adresse `0x400000`, pointant vers une zone mémoire de `0x3000` octets obtenue à l'aide de `mmap` ;
— la valeur `0x1e84`, lue à l'adresse `0x21028` ;
— enfin la valeur `0x3000` qui correspond à la taille de la zone mémoire à l'adresse `0x400000`.

Il est possible de remarquer que l'adresse `0x310c8` appartient à la section `.data` du binaire. Les données à cette adresse sont présentées ci-dessous :

```
$ /usr/bin/aarch64-linux-gnu-objdump -s --start-address=0x310c8 -j .data badbios.bin

badbios.bin:     file format elf64-littleaarch64

Contents of section .data:
 310c8 847f454c 46020101 00010040 0200b700  ..ELF......@....
 310d8 0e003114 05401000 13401800 35a80002  ..1..@...@..5...
 310e8 2400cf40 00380002 00400007 00060001  $..@.8...@......
```

```
310f8 005dfeff 3dfd7bbe a9fd0300 91000800  .]..=.{.........
[...]
32ed8 e0080013 f80800a2 4e6f2065 72726f72  ........No error
32ee8 2e0ad02a f5074261 6420696e 73747275  ...*..Bad instru
32ef8 6374696f 6e20706f 696e7465 1f007900  ction pointe..y.
32f08 496e7661 6c692400 011c00c0 4d656d6f  Invali$.....Memo
32f18 72792066 61756c74 11001049 36003a6e  ry fault...I6.:n
32f28 616c5e00 06400084 61726775 6d656e74  al^..@..argument
32f38 1a00f001 4f757420 6f66206d 656d6f72  ....Out of memor
32f48 792e0a00 00000000                    y.......
```

Le début du bloc de données semble contenir l'entête d'un programme ELF. La fin du bloc contient des chaînes de caractères mais qui semblent corrompues. On peut aussi noter que `0x310c8 + 0x1e84 = 0x32f4c`, qui correspond à la fin de la section `.data`.

L'analyse de la fonction `sub_10304` sera abordée dans la section suivante. Pour l'instant, on peut considérer que le retour de la fonction est différent de 0 et que l'exécution continue à l'adresse `0x10214` dont le code est présenté ci-dessous :

```
   10214:    b98022c3    ldrsw    x3, [x22,#32]              // x3 = 5
   10218:    aa0303e2    mov    x2, x3                       // x2 = 5
   1021c:    aa1503e1    mov    x1, x21                      // x1 = 0x3000
   10220:    aa1403e0    mov    x0, x20                      // x0 = retour de mmap = 0x400000
   10224:    d2801c48    mov    x8, #0xe2                    // #226
   10228:    d4000001    svc    #0x0                         // mprotect(0x400000, 0x3000, 5)
   1022c:    aa0003f4    mov    x20, x0
   10230:    b5fffb34    cbnz    x20, 0x10194                // retourne 1 si le retour de mprotect != 0
   10234:    b94002e0    ldr    w0, [x23]                    // w0 = *(0x21000) = 2
   10238:    11000718    add    w24, w24, #0x1               // w24++
   1023c:    6b00031f    cmp    w24, w0
   10240:    54000342    b.cs    0x102a8                     // saute à 0x102a8 si w24 >= 2
   10244:    f9400675    ldr    x21, [x19,#8]               // x21 = *(0x21040) = 0x10040
   10248:    aa1303f6    mov    x22, x19                    // x22 = 0x21038
   1024c:    913ffeb5    add    x21, x21, #0xfff
   10250:    9274ceb5    and    x21, x21, #0xfffffffffffff000 // x21 = 0x11000 (arrondit au multiple de 4096 supérieur)
   10254:    f8428666    ldr    x6, [x19],#40               // x6 = *(0x21038) = 0x500000, x19 = 0x21060
   10258:    aa1403e5    mov    x5, x20                     // x5 = 0 (retour de mprotect)
   1025c:    aa1403e4    mov    x4, x20                     // x4 = 0
   10260:    aa1a03e3    mov    x3, x26                     // x3 = 50
   10264:    aa1b03e2    mov    x2, x27                     // x2 = 3
   10268:    aa1503e1    mov    x1, x21                     // x1 = 0x11000
   1026c:    aa0603e0    mov    x0, x6                      // x0 = 0x500000
   10270:    d2801bc8    mov    x8, #0xde                   // #222
   10274:    d4000001    svc    #0x0                        // x0 = mmap(0x500000, 0x11000, 3, 50, 0, 0)
   10278:    aa0003f4    mov    x20, x0
   1027c:    eb06029f    cmp    x20, x6                     // compare le résultat de mmap à 0x500000
   10280:    aa1403e1    mov    x1, x20                     // x1 = 0
   10284:    54fff881    b.ne    0x10194                    // retourne 1 si différent
   10288:    b94026c0    ldr    w0, [x22,#36]               // w0 = *(2105c) = 0
   1028c:    34fffa80    cbz    w0, 0x101dc                 // saute à 0x101dc
```

Le code va tout d'abord appeler l'appel système `mprotect` sur la valeur de `mmap`. Pour connaître cette valeur avant l'appel à `mprotect`, il est possible de poser un point d'arrêt à l'adresse `0x10228` avec GDB et de consulter la valeur du registre `x0`.

Pour cela, QEMU doit être lancé avec le paramètre `-g` pour activer un stub GDB.

```
$ qemu-aarch64 -strace -g 1234 badbios.bin
```

Ensuite, il suffit de lancer GDB et de se connecter au stub GDB :

```
$ gdb-multiarch -q badbios.bin
Reading symbols from badbios.bin...(no debugging symbols found)...done.
(gdb) break *0x10228
Breakpoint 1 at 0x10228
(gdb) target remote 127.1:1234
Remote debugging using 127.1:1234
0x00000000000102cc in ?? ()
(gdb) cont
Continuing.

Breakpoint 1, 0x0000000000010228 in ?? ()
(gdb) p $x0
$1 = 0x400000
(gdb) x/i $pc
=> 0x10228:     svc     #0x0
(gdb) si
0x0000000000010230 in ?? ()
```

Sans surprise, on retrouve bien la valeur `0x400000`, à savoir l'adresse demandée à `mmap`. L'exécution de l'appel système à l'adresse `0x10228` déclenche l'affichage suivant au niveau de QEMU :

```
$ qemu-aarch64 -strace -g 1234 badbios.bin
14930 mmap(0x0000000000400000,12288,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS|MAP_FIXED,0,0) = 0x0000000000400000
14930 mprotect(0x0000000000400000,12288,PROT_EXEC|PROT_READ) = 0
```

L'appel à `mprotect` a donc modifié les permissions sur la zone mémoire à l'adresse `0x400000` : la permission `PROT_WRITE` a été supprimée, tandis que la permission `PROT_EXEC` a été rajoutée.

Le programme va ensuite incrémenter la valeur du registre `w24` et la comparer à la valeur stockée à l'adresse `0x21000`, c'est-à-dire 2. Les deux valeurs sont ensuite comparées et le programme saute à l'adresse `0x102a8` si `w24 >= 2`. Sinon, l'exécution continue à l'adresse `0x10244`.

Les instructions entre `0x10244` et `0x10274` vont effectuer un second appel à `mmap` avec les paramètres suivants :

— l'adresse demandée est `0x500000` ;
— la taille demandée est `0x11000` ;
— les permissions sont `PROT_READ|PROT_WRITE`, c'est-à-dire 3 ;
— les drapeaux sont `MAP_PRIVATE|MAP_ANONYMOUS|MAP_FIXED`, c'est-à-dire 50 ;
— les deux derniers paramètres sont nuls.

L'exécution continue à l'adresse `0x101dc` si `mmap` retourne l'adresse demandée.

```
   101dc:    f9400ec0    ldr    x0, [x22,#24]    // x0 = *(0x21050) = 0x10018
   101e0:    f9400ac7    ldr    x7, [x22,#16]    // x7 = *(0x21048) = 0x210b0
   101e4:    b4000180    cbz    x0, 0x10214      // saute à 0x10214 si x0 vaut 0
   101e8:    91000400    add    x0, x0, #0x1     // x0 = 0x10019
   101ec:    d2800024    mov    x4, #0x1         // x4 = 1
   101f0:    d2800002    mov    x2, #0x0         // x2 = 0
   101f4:    14000003    b    0x10200            // saute à 0x10200

   101f8:    aa0403e2    mov    x2, x4           // x2 = x4
   101fc:    aa0303e4    mov    x4, x3           // x4 = x3

   10200:    386268e5    ldrb    w5, [x7,x2]     // w5 = *(x7 + x2)
   10204:    91000483    add    x3, x4, #0x1     // x3 = x4 + 1
   10208:    38226825    strb    w5, [x1,x2]     // *(x1 + x2) = w5
   1020c:    eb00007f    cmp    x3, x0
   10210:    54ffff41    b.ne    0x101f8         // saute à 0x101f8 si x3 != 0

   10214:    b98022c3    ldrsw    x3, [x22,#32]  // x3 = *(0x21058) = 3
   10218:    aa0303e2    mov    x2, x3           // x2 = 3
   1021c:    aa1503e1    mov    x1, x21          // x1 = 0x11000
```

```
   10220:    aa1403e0    mov    x0, x20          // x0 = retour de mmap = 0x500000
   10224:    d2801c48    mov    x8, #0xe2        // #226
   10228:    d4000001    svc    #0x0             // mprotect(0x500000, 0x11000, 3)
   1022c:    aa0003f4    mov    x20, x0
   10230:    b5fffb34    cbnz    x20, 0x10194    // retourne 1 si le retour de mprotect != 0
   10234:    b94002e0    ldr    w0, [x23]        // w0 = *(0x21000) = 2
   10238:    11000718    add    w24, w24, #0x1   // w24++
   1023c:    6b00031f    cmp    w24, w0
   10240:    54000342    b.cs    0x102a8         // saute à 0x102a8 si w24 >= 2
```

Les instructions entre `0x101dc` et `0x10210` vont simplement copier `0x10018` octets de données à l'adresse `0x210b0` vers l'adresse obtenue précédemment par `mmap`, à savoir `0x500000`.

Ensuite le programme continue à l'adresse `0x10214` qui a déjà été étudiée précédemment. Un second appel à `mprotect` est effectué avec les paramètres suivants :

— l'adresse `0x500000` ;
— la taille `0x11000` ;
— les protections `PROT_READ|PROT_WRITE`, c'est-à-dire 3.

Cette fois, la valeur du registre `w24` contient 2, le programme continue à l'adresse `0x102a8`.

```
   102a8:    b9806bb8    ldrsw    x24, [x29,#104] // x24 = argc
   102ac:    f94033a1    ldr    x1, [x29,#96]     // x1 = 0x400514 (précédemment sauvegardé sur la pile)
   102b0:    9100033f    mov    sp, x25           // la pile pointe sur argv
   102b4:    d10023ff    sub    sp, sp, #0x8      // décale la pile de 8 octets
   102b8:    f90003f8    str    x24, [sp]         // stocke argc sur la pile
   102bc:    aa0103e2    mov    x2, x1
   102c0:    d63f0040    blr    x2                // appelle la fonction à l'adresse 0x400514
   102c4:    aa0003f7    mov    x23, x0
   102c8:    17ffffb4    b    0x10198             // retourne x0
```

Le programme charge dans le registre `x1` la valeur `0x400514` qui a été précédemment sauvegardée sur la pile par l'instruction `str x0, [x29,#96]` à l'adresse `0x1011c`. Ensuite, le programme repositionne `argc` et `argv` sur la pile avant de brancher à l'adresse contenue dans le registre `r2`, c'est-à-dire `0x400514`.

En synthèse, la fonction `sub_1010c` effectue les opérations suivantes :

```
void sub_1010c(int argc, char **argv) {
    mmap(0x400000, 0x3000, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_FIXED, 0, 0);
    sub_10304(0x310c8, 0x400000, 0x1e84, 0x3000);
    mprotect(0x0400000, 0x3000, PROT_EXEC|PROT_READ);
    mmap(0x500000, 0x11000, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_FIXED, 0, 0);
    memcpy(0x500000, 0x201b0, 0x10018);
    mprotect(0x500000, 0x11000, PROT_READ|PROT_WRITE);
    sub_400514(argc, argv);
}
```

Il faut maintenant s'intéresser à la fonction `sub_10304`.

2.2.3 Analyse de la fonction `sub_10304`

Le fonctionnement de la fonction `sub_10304` est assez complexe et sa compréhension n'est pas nécessaire pour continuer le challenge. Cependant, le résultat de la rétro-conception de cette fonction est disponible à l'annexe A.2.

Le prototype de la fonction est :

```
int sub_10304(char *src, char *dst, uint32_t slen, uint32_t dlen);
```

La fonction effectue des traitements sur les données à l'adresse `src` puis les copie à l'adresse `dst`. La fonction retourne le nombre d'octets qui ont été copiés dans la zone mémoire de destination. Pour connaître ce résultat, il est possible placer un point d'arrêt à l'adresse `0x10464`.

```
(gdb) break *0x10464
Breakpoint 2 at 0x10464
(gdb) cont
Continuing.

Breakpoint 2, 0x0000000000010464 in ?? ()
(gdb) p $x0
$2 = 0x2c08
```

On remarque le nombre d'octets copiés est supérieur au nombre d'octets source (`0x1e84`) : on peut alors supposer que `sub_10304` est une fonction de décompression de données.

2.2.4 Analyse de la fonction `sub_400514`

Avant d'appeler la fonction `sub_400514`, il est intéressant de sauvegarder le contenu des deux zones mémoires aux adresses `0x400000` et `0x500000`.

```
(gdb) break *0x102c0
Breakpoint 4 at 0x102c0
(gdb) cont
Continuing.

Breakpoint 4, 0x00000000000102c0 in ?? ()
(gdb) dump binary memory 400000.bin 0x400000 (0x400000+0x3000)
(gdb) dump binary memory 500000.bin 0x500000 (0x500000+0x11000)
```

La commande `file` permet d'obtenir des informations sur les deux fichiers obtenus.

```
$ file 400000.bin
400000.bin: ELF 64-bit LSB  executable, ARM aarch64, version 1 (SYSV), statically linked, stripped
$ readelf -a 400000.bin
readelf: Error: Unable to read in 0x40 bytes of section headers
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:               0x400514
  Start of program headers:          64 (bytes into file)
  Start of section headers:          131240 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         2
  Size of section headers:           64 (bytes)
  Number of section headers:         7
  Section header string table index: 6
readelf: Error: Unable to read in 0x1c0 bytes of section headers
readelf: Error: Section headers are not available!

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
```

```
                FileSiz            MemSiz              Flags  Align
  NULL          0x0000000000000000 0x0000000000000000 0x0000000000000000
                0x0000000000000000 0x0000000000000000         0
  NULL          0x0000000000000000 0x0000000000000000 0x0000000000000000
                0x0000000000000000 0x0000000000000000         0

There is no dynamic section in this file.
$ file 500000.bin
500000.bin: data
$ hexdump -C 500000.bin | head -n 20
00000000  a2 db e7 41 f2 76 3b 28  08 77 53 e2 39 39 c2 b0  |...A.v;(.wS.99..|
00000010  6f c3 10 cf f9 e1 29 9d  5c 4e 33 b4 15 7a 41 20  |o.....).\N3..zA |
00000020  99 5e c9 8a 4d 15 55 f2  e8 88 c9 98 03 d7 29 ba  |.^..M.U.......).|
00000030  00 2d 1e b5 46 72 22 0d  0b 01 d4 6d 6a e9 9a 8d  |.-..Fr"....mj...|
00000040  96 dc e2 8b 3b e1 25 46  ea 00 fe 98 f3 15 22 58  |....;.%F......"X|
00000050  1e 21 6c a7 54 fb a6 91  d8 cc e0 bd 7f 41 cb 76  |.!l.T........A.v|
00000060  7c ef 21 25 b7 d8 53 8c  ae 50 a6 0c fa d6 05 23  ||.!%..S..P.....#|
00000070  52 78 b9 e1 12 01 82 01  e9 31 92 e6 e8 72 cb b5  |Rx.......1...r..|
00000080  63 0e 0c 25 c0 ee 1d a0  f3 2a 66 a6 66 db 97 bc  |c..%.....*f.f...|
00000090  20 5c c6 32 2c 3c 2e 66  76 2c 7e e8 3e 20 e9 d3  | \.2,<.fv,~.> ..|
```

Le fichier `400000.bin` semble être un second exécutable au format ELF mais pour lequel les entêtes de section semblent corrompus. Les entêtes de programme semblent également avoir été modifiés, tous les champs étant à 0. On retrouve cependant un point d'entrée à `0x400514`, qui correspond au branchement effectué par la fonction `sub_1010c`.

Le format du fichier `500000.bin` est inconnu : les données semblent correspondre à du contenu chiffré ou compressé.

La première idée qui vient alors à l'esprit est d'essayer de restaurer les entêtes de section et de programme pour obtenir un binaire fonctionnel.

Le code C ci-dessous recrée les entêtes des sections `NULL`, `.text` et `.shstrtab`.

```c
#define NUM_SECTIONS 3
void restore_program(char *output, char *stext, int stext_size) {
    Elf64_Ehdr *ehdr;
    Elf64_Phdr *phdr;
    Elf64_Shdr shdrs[NUM_SECTIONS]; // NULL section + text + strtab
    char *strtab;
    int fd, strtab_size = 0;

    memset(shdrs, 0, NUM_SECTIONS * sizeof(Elf64_Shdr));

    /* Fix program header */
    phdr = (Elf64_Phdr *) (stext + sizeof(Elf64_Ehdr));
    phdr->p_type = PT_LOAD;
    phdr->p_flags = PF_X | PF_R;
    phdr->p_offset = 0;
    phdr->p_vaddr = phdr->p_paddr = 0x400000;
    phdr->p_filesz = phdr->p_memsz = stext_size;
    phdr->p_align = 0x10000;

    strtab = malloc(1024);

    /* section NULL */
    shdrs[0].sh_name = strtab_size;
    memcpy(strtab, "", 1); strtab_size += 1;

    /* section .text */
    shdrs[1].sh_name = strtab_size;
    shdrs[1].sh_type = SHT_PROGBITS;
    shdrs[1].sh_flags = SHF_ALLOC | SHF_EXECINSTR;
    shdrs[1].sh_addr = 0x400000 + sizeof(Elf64_Ehdr) + 2 * sizeof(Elf64_Phdr);
```

```
    shdrs[1].sh_offset = sizeof(Elf64_Ehdr) + 2 * sizeof(Elf64_Phdr);
    shdrs[1].sh_size = stext_size;
    shdrs[1].sh_addralign = 4;
    memcpy(strtab + strtab_size, ".text", strlen(".text") + 1);
    strtab_size += strlen(".text") + 1;

    /* section .shstrab */
    shdrs[2].sh_name = strtab_size;
    shdrs[2].sh_type = SHT_STRTAB;
    shdrs[2].sh_offset = stext_size;
    memcpy(strtab + strtab_size, ".shstrtab", strlen(".shstrtab") + 1);
    strtab_size += strlen(".shstrtab") + 1;
    shdrs[2].sh_size =  strtab_size;
    shdrs[2].sh_addralign = 1;

    ehdr = (Elf64_Ehdr *) stext;
    ehdr->e_shoff = stext_size + strtab_size;
    ehdr->e_shnum = NUM_SECTIONS;
    ehdr->e_shstrndx = NUM_SECTIONS - 1;

    fd = creat(output, S_IRWXU);
    if (fd == -1) {
        perror("creat");
        exit(EXIT_FAILURE);
    }
    write(fd, stext, stext_size);
    write(fd, strtab, strtab_size);
    write(fd, shdrs, NUM_SECTIONS * sizeof(Elf64_Shdr));
    close(fd);
}
```

En passant le contenu du fichier `400000.bin` comme paramètre `stext` et la valeur `0x3000` pour `stext_size`, on obtient un nouveau fichier `badbios2.bin`.

```
$ readelf -S badbios2.bin
There are 3 section headers, starting at offset 0x3011:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS         00000000004000b0  000000b0
       0000000000003000  0000000000000000  AX       0     0     4
  [ 2] .shstrtab         STRTAB           0000000000000000  00003000
       0000000000000011  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
$ readelf -l badbios2.bin

Elf file type is EXEC (Executable file)
Entry point 0x400514
There are 2 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000003000 0x0000000000003000  R E    10000
  NULL           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000         0

 Section to Segment mapping:
```

```
  Segment Sections...
   00
   01
```

Pour autant, le binaire n'est pas fonctionnel :

```
$ qemu-aarch64 badbios2.bin
qemu: uncaught target signal 11 (Segmentation fault) - core dumped
```

L'utilisation de GDB permet d'investiguer plus en amont :

```
$ gdb-multiarch -q badbios2.bin
Reading symbols from badbios2.bin...(no debugging symbols found)...done.
(gdb) target remote 127.1:1234
Remote debugging using 127.1:1234
0x0000000000400514 in ?? ()
(gdb) cont
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x00000000004004f4 in ?? ()
(gdb) x/i $pc
=> 0x4004f4:    str     x3, [x2]
(gdb) (gdb) p $x2
$1 = 0x510018
(gdb) maintenance info sections
Exec file:
    `badbios2.bin', file type elf64-littleaarch64.
 [0]     0x004000b0->0x004030b0 at 0x000000b0: .text ALLOC LOAD READONLY CODE HAS_CONTENTS
```

L'adresse `0x510018` est déférencée mais aucune section chargée par le binaire ne contient cette adresse. Le fichier `500000.bin` obtenu précédemment doit certainement correspondre à la section `.data` du programme : les entêtes de programme et de section doivent également être corrigés, de la même manière que précédemment.

Enfin, en voulant désassembler la section `.text` avec la commande `objdump`, on remarque que certaines instructions ne peuvent pas être décodées.

```
$ /usr/bin/aarch64-linux-gnu-objdump -d badbios2.bin

badbios2.bin:     file format elf64-littleaarch64

Disassembly of section .text:

00000000004000b0 <.text>:
  4000b0:       a9be7bfd        stp     x29, x30, [sp,#-32]!
  4000b4:       910003fd        mov     x29, sp
  4000b8:       90000800        adrp    x0, 0x500000
  4000bc:       d2a00021        mov     x1, #0x10000                    // #65536
  4000c0:       91000000        add     x0, x0, #0x0
  4000c4:       910043a2        add     x2, x29, #0x10
  4000c8:       94000a26        bl      0x402960
[...]
  402b1c:       f901ae7f        str     xzr, [x19,#856]
  402b20:       12800000        mov     w0, #0xffffffff                 // #-1
  402b24:       a94153f3        ldp     x19, x20, [sp,#16]
  402b28:       a9425bf5        ldp     x21, x22, [sp,#32]
  402b2c:       f9401bf7        ldr     x23, [sp,#48]
  402b30:       a8c47bfd        ldp     x29, x30, [sp],#64
  402b34:       d65f03c0        ret
  402b38:       00402b70        .inst   0x00402b70 ; undefined
  402b3c:       00000000        .inst   0x00000000 ; undefined
  402b40:       00402b80        .inst   0x00402b80 ; undefined
  402b44:       00000000        .inst   0x00000000 ; undefined
```

```
[...]
```

Les données à partir de l'adresse `0x402b38` ne semblent effectivement pas correspondre à du code ARM64. Par contre, on retrouve des adresses appartenant à la section `.text`, par exemple `0x402b70` et `0x402b80`. Ces données correspondent en fait à la section `.rodata` du programme : il faut donc également créer un entête de section pour cette dernière.

Le programme `unpack.c`, disponible à l'annexe A.2, implémente la fonction `sub_400514` et permet de produire un binaire fonctionnel à partir des zones mémoires aux adresses `0x400000` et `0x500000`, en restaurant les entêtes de programmes et de sections.

```
$ gcc -Wall -o unpack unpack.c
$ ./unpack badbios.bin badbios2.bin
$ md5sum badbios2.bin
8021a12f55603445a331212a6fd907aa  badbios2.bin
$ readelf -S badbios2.bin
There are 3 section headers, starting at offset 0x3011:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS         00000000004000b0  000000b0
       0000000000003000  0000000000000000  AX       0     0     4
  [ 2] .shstrtab         STRTAB           0000000000000000  00003000
       0000000000000011
$ readelf -l badbios2.bin

Elf file type is EXEC (Executable file)
Entry point 0x400514
There are 2 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000003000 0x0000000000003000  R E    10000
  LOAD           0x0000000000003000 0x0000000000500000 0x0000000000500000
                 0x0000000000011000 0x0000000000011000  RW     10000

 Section to Segment mapping:
  Segment Sections...
   00     .text .rodata
   01     .data
```

Cette fois le binaire obtenu semble fonctionnel :

```
$ qemu-aarch64 badbios2.bin
:: Please enter the decryption key: AAAAAAAAAAAAAAAA
:: Trying to decrypt payload...
   Invalid padding.
```

Pour poursuivre le challenge, il faut maintenant analyser ce binaire, en partant du point d'entrée `0x400514`.

2.3 Analyse du second programme

Le programme commence par le code suivant :

```
4004d8:    a9bf7bfd     stp    x29, x30, [sp,#-16]!
4004dc:    93407c03     sxtw    x3, w0             // x3 = argc
4004e0:    910003fd     mov    x29, sp
4004e4:    91000463     add    x3, x3, #0x1        // x3 = 2
4004e8:    90000882     adrp    x2, 0x510000       // x2 = 0x510000
4004ec:    8b030c23     add    x3, x1, x3, lsl #3  // x3 = argv + 16
4004f0:    91006042     add    x2, x2, #0x18       // x2 = 0x510018
4004f4:    f9000043     str    x3, [x2]            // *(0x510018) = argv + 16
4004f8:    97fffeee     bl    0x4000b0
4004fc:    93407c01     sxtw    x1, w0
400500:    aa0103e0     mov    x0, x1
400504:    d2800bc8     mov    x8, #0x5e           // #94
400508:    d4000001     svc    #0x0                // exit_group
40050c:    aa0003e1     mov    x1, x0
400510:    14000000     b    0x400510

400514:    d280001e     mov    x30, #0x0           // #0
400518:    910003fd     mov    x29, sp
40051c:    f94003e0     ldr    x0, [sp]            // x0 = argc
400520:    910023e1     add    x1, sp, #0x8        // x1 = argv
400524:    17ffffed     b    0x4004d8
```

Le programme va directement appeler la fonction `sub_4000b0` dont le code est présenté ci-dessous :

```
4000b0:    a9be7bfd     stp    x29, x30, [sp,#-32]!
4000b4:    910003fd     mov    x29, sp
4000b8:    90000800     adrp    x0, 0x500000          // x0 = 0x500000
4000bc:    d2a00021     mov    x1, #0x10000          // x1 = 0x10000
4000c0:    91000000     add    x0, x0, #0x0          //
4000c4:    910043a2     add    x2, x29, #0x10        // x2 = sp + 16
4000c8:    94000a26     bl    0x402960               // sub_402960(0x500000, 0x10000, sp + 16)
4000cc:    37f800a0     tbnz    w0, #31, 0x4000e0    // teste le bit de poids faible de w0 (!= 0)
4000d0:    f9400ba0     ldr    x0, [x29,#16]         // x0 = *(sp + 16)
4000d4:    94000a10     bl    0x402914               // x0 = sub_402914(x0)
4000d8:    a8c27bfd     ldp    x29, x30, [sp],#32
4000dc:    d65f03c0     ret                         // retourne w0
4000e0:    12800000     mov    w0, #0xffffffff       // #-1
4000e4:    17fffffd     b    0x4000d8                // retourne -1
```

La fonction `sub_4000b0` va simplement appeler les fonctions `sub_402960` et `sub_402914` dans l'ordre.

Avant d'aller plus loin dans l'analyse, il est utile de représenter le graphe d'appels des fonctions afin de comprendre la logique du programme. Ce graphe est présenté à la figure 2.1.

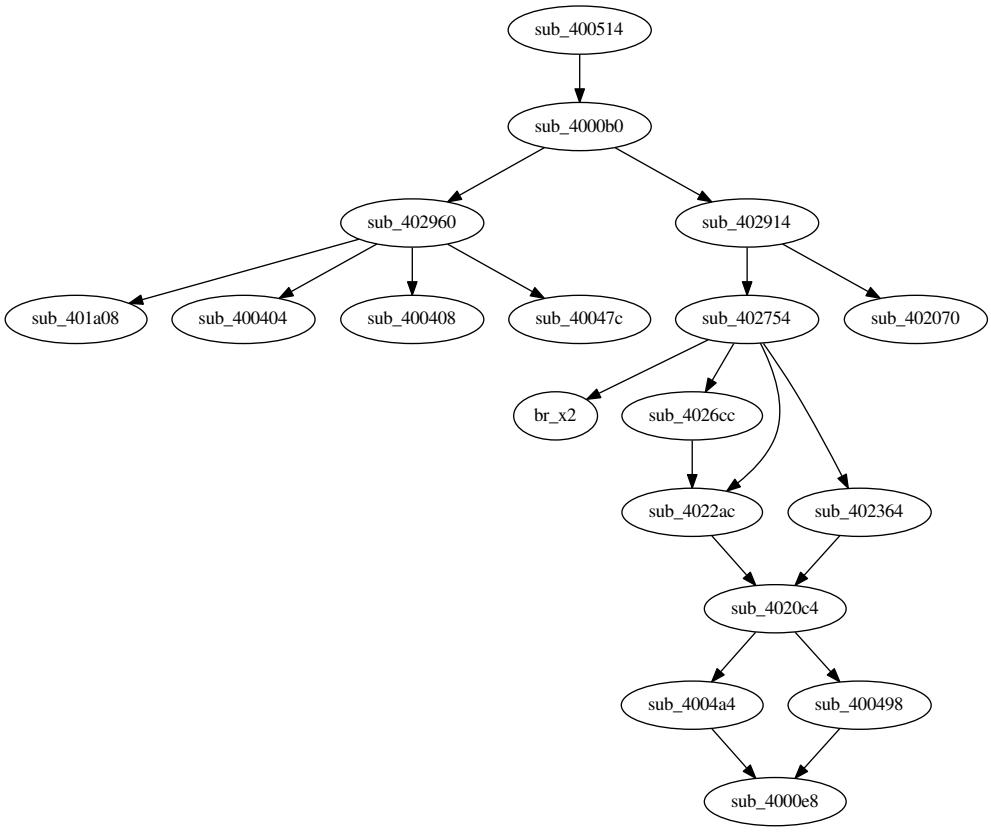

FIGURE 2.1 – Graphe d'appels de `badbios2.bin`

2.3.1 Analyse de la fonction `sub_402960`

La fonction `sub_402960` commence par les instructions suivantes :

```
402960:    a9bc7bfd    stp    x29, x30, [sp,#-64]!
402964:    910003fd    mov    x29, sp
402968:    a90153f3    stp    x19, x20, [sp,#16]
40296c:    a9025bf5    stp    x21, x22, [sp,#32]
402970:    f9001bf7    str    x23, [sp,#48]
402974:    aa0003f4    mov    x20, x0                  // x20 = 0x500000
402978:    aa0103f5    mov    x21, x1                  // x21 = 0x10000
40297c:    aa0203f7    mov    x23, x2                  // x23 = adresse sur la pile
402980:    d2800006    mov    x6, #0x0                 // x6 = 0
402984:    d2800453    mov    x19, #0x22               // x19 = 34
402988:    d2800069    mov    x9, #0x3                 // x9 = 3
40298c:    d2820007    mov    x7, #0x1000              // x7 = 4096
402990:    aa0603e5    mov    x5, x6                   // x5 = 0
402994:    aa0603e4    mov    x4, x6                   // x4 = 0
402998:    aa1303e3    mov    x3, x19                  // x3 = 34
40299c:    aa0903e2    mov    x2, x9                   // x2 = 3
4029a0:    aa0703e1    mov    x1, x7                   // x1 = 4096
4029a4:    aa0603e0    mov    x0, x6                   // x0 = 0
4029a8:    d2801bc8    mov    x8, #0xde                // x8 = 222
4029ac:    d4000001    svc    #0x0                     // addr0 = mmap(0, 4096, 3, 34, 0, 0)
4029b0:    aa0003f3    mov    x19, x0                  // x19 = addr0
4029b4:    b100067f    cmn    x19, #0x1                // compare x19 à -1
4029b8:    54000b40    b.eq    0x402b20                // saute si égal (erreur mmap)
```

Ce code réalise simplement un appel à `mmap`. Le décodage des paramètres peut être obtenu en surveillant la sortie de QEMU :

```
$ qemu-aarch64 -strace -g 1234 badbios2.bin
12846 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) = 0x0000004000801000
```

Il s'agit a priori d'une simple zone de mémoire de 4096 octets, avec des permissions en lecture / écriture.

La fonction continue avec les instructions suivantes :

```
4029bc:    913ffea0    add    x0, x21, #0xfff              // x0 = 0x10fff
4029c0:    d2800006    mov    x6, #0x0                     // x6 = 0
4029c4:    9274cc07    and    x7, x0, #0xfffffffffffff000  // x7 = arrondit 0x10000 au multiple de 4096 supérieur
4029c8:    d280044a    mov    x10, #0x22                   // x10 = 34
4029cc:    d2800069    mov    x9, #0x3                     // x9 = 3
4029d0:    aa0603e5    mov    x5, x6                       // x5 = 0
4029d4:    aa0603e4    mov    x4, x6                       // x4 = 0
4029d8:    aa0a03e3    mov    x3, x10                      // x3 = 34
4029dc:    aa0903e2    mov    x2, x9                       // x2 = 3
4029e0:    aa0703e1    mov    x1, x7                       // x1 = 0x10000
4029e4:    aa0603e0    mov    x0, x6                       // x0 = 0
4029e8:    d2801bc8    mov    x8, #0xde                    // #222
4029ec:    d4000001    svc    #0x0                         // addr1 = mmap(0, 0x10000, 3, 34, 0, 0)
4029f0:    aa0003e6    mov    x6, x0                       // x6 = addr1
4029f4:    f9002a66    str    x6, [x19,#80]                // *(addr0 + 0x50) = addr1
4029f8:    b10004df    cmn    x6, #0x1                     // compare x6 à -1
4029fc:    54000920    b.eq    0x402b20                    // saute si égal (erreur mmap)
```

Un second appel à `mmap` est réalisé pour cette fois obtenir une zone mémoire de 65536 octets. Les permissions sont également `PROT_READ|PROT_WRITE`.

Les instructions suivantes sont :

```
402a00:    91016264    add    x4, x19, #0x58               // x4 = addr0 + 0x58
402a04:    52800003    mov    w3, #0x0                     // w3 = 0

402a08:    93407c65    sxtw    x5, w3                      // x5 = w3
402a0c:    8b0504a5    add    x5, x5, x5, lsl #1           // x5 = 2 * x5
402a10:    8b050e65    add    x5, x19, x5, lsl #3          // x5 = addr0 + 8 * x5 = addr0 + 16 * w3
402a14:    3941a0a6    ldrb    w6, [x5,#104]               // w6 = *(addr0 + 0x68 + 16 * w3)
402a18:    11000463    add    w3, w3, #0x1                 // w3 += 1
402a1c:    321f00c6    orr    w6, w6, #0x2                 // w6 |= 2
402a20:    121f78c6    and    w6, w6, #0xfffffffe          // w2 &= 0xfffffffe (mets le dernier bit à 0)
402a24:    3901a0a6    strb    w6, [x5,#104]               // *(addr0 + 0x68 + 16 * w3) = w6
402a28:    f900009f    str    xzr, [x4]                    // *(addr0 + 0x58 + 24 * (w3 - 1)) = 0
402a2c:    f900049f    str    xzr, [x4,#8]                 // *(addr0 + 0x60 + 24 * (w3 - 1)) = 0
402a30:    7100807f    cmp    w3, #0x20                    // compare w3 à 32
402a34:    91006084    add    x4, x4, #0x18                // x4 += 24
402a38:    54fffe81    b.ne    0x402a08                    // boucle si w3 != 32
```

Ce code initialise un tableau de 32 structures de 24 octets contenant 3 champs de 8 octets. Les deux premiers champs sont mis à 0. Pour le troisième champ, le bit de poids faible est mis à 0 et le second bit de poids faible est mis à 1.

La fonction continue avec les instructions suivantes :

```
402a3c:    d2800006    mov    x6, #0x0                     // x6 = 0
402a40:    d280044a    mov    x10, #0x22                   // x10 = 34
402a44:    d2800069    mov    x9, #0x3                     // x9 = 3
402a48:    d2820007    mov    x7, #0x1000                  // x7 = 4096
402a4c:    aa0603e5    mov    x5, x6                       // x5 = 0
402a50:    aa0603e4    mov    x4, x6                       // x4 = 0
```

```
402a54:    aa0a03e3     mov    x3, x10                  // x3 = 34
402a58:    aa0903e2     mov    x2, x9                   // x2 = 3
402a5c:    aa0703e1     mov    x1, x7                   // x1 = 4096
402a60:    aa0603e0     mov    x0, x6                   // x0 = 0
402a64:    d2801bc8     mov    x8, #0xde                // #222
402a68:    d4000001     svc    #0x0                     // addr2 = mmap(0, 4096, 9, 34, 0, 0)
402a6c:    aa0003e6     mov    x6, x0                   // x6 = addr2
402a70:    b10004df     cmn    x6, #0x1                 // compare addr à -1
402a74:    54000540     b.eq    0x402b1c                // saute si égal (erreur mmap)
```

Un troisième appel à `mmap` est effectué pour obtenir une seconde zone mémoire de 4096 octets.

Les instructions suivantes sont :

```
402a78:    f901ae66     str    x6, [x19,#856]           // *(addr0 + 0x358) = addr2
402a7c:    b4000526     cbz    x6, 0x402b20             // saute si addr2 == 0
402a80:    aa1303e0     mov    x0, x19                  // x0 = addr0
402a84:    97fffbe1     bl    0x401a08                  // x0 = sub_401a08(addr0)
402a88:    97fff65f     bl    0x400404                  // sub_400404(x0)
402a8c:    91004276     add    x22, x19, #0x10          // x22 = addr0 + 0x10
402a90:    d0000861     adrp    x1, 0x510000            // x1 = 0x510000
402a94:    aa1603e0     mov    x0, x22                  // x0 = addr0 + 0x10
402a98:    91000021     add    x1, x1, #0x0             //
402a9c:    52801002     mov    w2, #0x80                // w2 = 128
402aa0:    52800003     mov    w3, #0x0                 // w3 = 0
402aa4:    97fff659     bl    0x400408                  // sub_400408(addr0 + 0x10, 0x510000)
402aa8:    d0000861     adrp    x1, 0x510000            // x1 = 0x510000
402aac:    aa1603e0     mov    x0, x22                  // x0 = addr0 + 0x10
402ab0:    91008021     add    x1, x1, #0x20            // x1 = 0x510020
402ab4:    97fff672     bl    0x40047c                  // sub_40047c(addr0 + 0x10, 0x510020)
402ab8:    39400260     ldrb    w0, [x19]               // w0 = *(addr0)
402abc:    b900067f     str    wzr, [x19,#4]            // *(addr0 + 4) = 0
402ac0:    121f7800     and    w0, w0, #0xfffffffe      // w0 &= 0xfffffffe (bit de poids faible à 0)
402ac4:    39000260     strb    w0, [x19]               // (*addr0) = w0
402ac8:    f900067f     str    xzr, [x19,#8]            // *(addr0 + 8) = 0
402acc:    f9402a60     ldr    x0, [x19,#80]            // x0 = *(addr0 + 0x50) = addr1
402ad0:    b4000195     cbz    x21, 0x402b00            // saute si x21 vaut 0
```

Ce code se contente d'appeler les sous-fonctions suivantes :

- `x0 = sub_401a08(addr0)`;
- `sub_400404(x0)`;
- `sub_400408(addr0 + 0x10, 0x510000)`;
- `sub_40047c(addr0 + 0x10, 0x510020)`.

Ces quatre sous-fonctions seront analysées par la suite.

Les données aux adresses `addr0`, `addr0 + 4` et `addr0 + 8` sont également initialisées.

La fonction continue avec les instructions suivantes :

```
402ad4:    910006a1     add    x1, x21, #0x1            // x1 = 0x10001
402ad8:    d2800024     mov    x4, #0x1                 // x4 = 1
402adc:    d2800003     mov    x3, #0x0                 // x3 = 0
402ae0:    14000003     b    0x402aec

402ae4:    aa0403e3     mov    x3, x4
402ae8:    aa0503e4     mov    x4, x5

402aec:    38636a86     ldrb    w6, [x20,x3]            // w6 = *(0x500000 + x3)
402af0:    91000485     add    x5, x4, #0x1             // x5 = x4 + 1
402af4:    38236806     strb    w6, [x0,x3]             // *(addr1 + x3) = w6
402af8:    eb0100bf     cmp    x5, x1                   // compare x5 à 0x10001
```

```
  402afc:    54ffff41     b.ne    0x402ae4
```

Ce code copie 65536 octets de données de l'adresse `0x500000` vers `addr0 + 0x10`.

La fonction finit stocker la valeur de `addr0` à l'adresse sur la pile stockée dans le registre `x23` puis retourne la valeur 0 avec les instructions suivantes :

```
  402b00:    f90002f3     str    x19, [x23]
  402b04:    52800000     mov    w0, #0x0                    // #0
  402b08:    a94153f3     ldp    x19, x20, [sp,#16]
  402b0c:    a9425bf5     ldp    x21, x22, [sp,#32]
  402b10:    f9401bf7     ldr    x23, [sp,#48]
  402b14:    a8c47bfd     ldp    x29, x30, [sp],#64
  402b18:    d65f03c0     ret
```

Pour résumer, la fonction `sub_402960` réalise les opérations suivantes :

```
int sub_402960(char *addr, int count, char **res) {
    addr0 = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, 0, 0);
    addr1 = mmap(NULL, count, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, 0, 0);
    *(addr0 + 0x50) = addr1;
    for (i = 0; i < 32; i++) {
        *(addr0 + 0x58 + 24 * i) = 0;
        *(addr0 + 0x58 + 8 + 24 * i) = 0;
        *(addr0 + 0x58 + 16 + 24 * i) |= 2;
        *(addr0 + 0x58 + 16 + 24 * i) &= 0xfffffffe;
    }
    addr2 = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, 0, 0);
    *(addr0 + 0x358) = addr2;
    x0 = sub_401a08(addr0);
    sub_400404(x0);
    sub_400408(addr0 + 16, 0x510000);
    sub_40047c(addr0 + 16, 0x510020);
    *(addr0) &= 0xfffffffe;
    *(addr0 + 4) = 0;
    *(addr0 + 8) = 0;
    memcpy(addr1, addr, count);
    *res = addr0;
    return 0;
}
```

Analyse de la fonction `sub_401a08`

La fonction `sub_401a08` commence par les instructions suivantes :

```
  401a08:    9b1f1826     madd    x6, x1, xzr, x6   // x6 = x1 * 0 + x6 = x6
  401a0c:    f0ffffe3     adrp    x3, 0x400000      // x3 = 0x400000
  401a10:    9a81b3e1     csel    x1, xzr, x1, lt   // x1 = x1
  401a14:    9a81a3e1     csel    x1, xzr, x1, ge   // x1 = 0
  401a18:    9118b063     add    x3, x3, #0x62c     // x3 += 0x62c
  401a1c:    d1003063     sub    x3, x3, #0xc       // x3 -= 0xc
  401a20:    d1007063     sub    x3, x3, #0x1c      // x3 -= 0x1c = 0x400604
  401a24:    17ffff77     b    0x401800
```

La compréhension du code est rendue plus complexe par l'utilisation d'instructions inutiles, comme `madd` ou `csel`. L'exécution continue ensuite à l'adresse `0x401800`.

```
  401800:    8b010002     add    x2, x0, x1      // x2 = addr0 + x1
  401804:    d10ecc21     sub    x1, x1, #0x3b3
  401808:    d11bc821     sub    x1, x1, #0x6f2
  40180c:    d1108421     sub    x1, x1, #0x421
```

```
401810:    913b3821     add    x1, x1, #0xece  // x1 += 8
401814:    d1225442     sub    x2, x2, #0x895
401818:    d1193442     sub    x2, x2, #0x64d
40181c:    913ca842     add    x2, x2, #0xf2a
401820:    91024042     add    x2, x2, #0x90   // x2 += 0xd8
401824:    f9014443     str    x3, [x2,#648]   // *(addr0 + 0x360) = x3 = 0x400604
401828:    d1024042     sub    x2, x2, #0x90
40182c:    d1012042     sub    x2, x2, #0x48   // x2 -= 0xd8 (addr0 + x1)
401830:    f103e03f     cmp    x1, #0xf8       // compare x1 à 248
401834:    54fffe61     b.ne    0x401800
```

Ce code initialise un tableau de 31 pointeurs (`248/8`) à l'adresse `addr0 + 0x360` avec la valeur `0x400604`. On peut s'assurer du résultat sous GDB :

```
(gdb) break *0x401838
Breakpoint 13 at 0x401838
(gdb) cont
Continuing.

Breakpoint 13, 0x0000000000401838 in ?? ()
(gdb) x/32gx ($x0+0x360)
0x4000801360:   0x0000000000400604      0x0000000000400604
0x4000801370:   0x0000000000400604      0x0000000000400604
0x4000801380:   0x0000000000400604      0x0000000000400604
0x4000801390:   0x0000000000400604      0x0000000000400604
0x40008013a0:   0x0000000000400604      0x0000000000400604
0x40008013b0:   0x0000000000400604      0x0000000000400604
0x40008013c0:   0x0000000000400604      0x0000000000400604
0x40008013d0:   0x0000000000400604      0x0000000000400604
0x40008013e0:   0x0000000000400604      0x0000000000400604
0x40008013f0:   0x0000000000400604      0x0000000000400604
0x4000801400:   0x0000000000400604      0x0000000000400604
0x4000801410:   0x0000000000400604      0x0000000000400604
0x4000801420:   0x0000000000400604      0x0000000000400604
0x4000801430:   0x0000000000400604      0x0000000000400604
0x4000801440:   0x0000000000400604      0x0000000000400604
0x4000801450:   0x0000000000400604      0x0000000000000000
```

La fonction continue avec les instructions suivantes :

```
401838:    14000081     b    0x401a3c
[...]
401a3c:    f0ffffe1     adrp    x1, 0x400000
401a40:    91193021     add    x1, x1, #0x64c
401a44:    d1008021     sub    x1, x1, #0x20
401a48:    d100c021     sub    x1, x1, #0x30
401a4c:    d1269000     sub    x0, x0, #0x9a4
401a50:    912bd000     add    x0, x0, #0xaf4
401a54:    9100e000     add    x0, x0, #0x38
401a58:    91034000     add    x0, x0, #0xd0
401a5c:    f900f401     str    x1, [x0,#488]
401a60:    d1034000     sub    x0, x0, #0xd0
401a64:    d100e000     sub    x0, x0, #0x38
401a68:    f0ffffe1     adrp    x1, 0x400000
401a6c:    91193021     add    x1, x1, #0x64c
401a70:    d1001021     sub    x1, x1, #0x4
401a74:    d100b021     sub    x1, x1, #0x2c
401a78:    d1054000     sub    x0, x0, #0x150
401a7c:    d100a021     sub    x1, x1, #0x28
401a80:    d1281000     sub    x0, x0, #0xa04
401a84:    d1036400     sub    x0, x0, #0xd9
401a88:    912fd400     add    x0, x0, #0xbf5
401a8c:    910b0000     add    x0, x0, #0x2c0
401a90:    91006000     add    x0, x0, #0x18
401a94:    f9002401     str    x1, [x0,#72]
```

```
[...]
40200c:    f0ffffe1     adrp    x1, 0x401000
402010:    d101a000     sub    x0, x0, #0x68
402014:    91137021     add    x1, x1, #0x4dc
402018:    d100a021     sub    x1, x1, #0x28
40201c:    d1009021     sub    x1, x1, #0x24
402020:    d1311800     sub    x0, x0, #0xc46
402024:    91325800     add    x0, x0, #0xc96
402028:    d131fc00     sub    x0, x0, #0xc7f
40202c:    9138fc00     add    x0, x0, #0xe3f
402030:    9102c000     add    x0, x0, #0xb0
402034:    f900c401     str    x1, [x0,#392]
402038:    d102c000     sub    x0, x0, #0xb0
40203c:    d1070000     sub    x0, x0, #0x1c0
402040:    d1014000     sub    x0, x0, #0x50
402044:    d61f03c0     br    x30
```

Ce code est composé de 31 séquences similaires : les valeurs des registres `x0` et `x1` sont mises à jour à l'aide d'opérations arithmétiques puis la valeur du registre `x1` est écrite en mémoire.

Plutôt que d'effectuer tous les calculs à la main, il est plus simple de poser des points d'arrêts avant chaque instruction `str`. Ces instructions peuvent être identifiées à l'aide de la commande suivante :

```
$ /usr/bin/aarch64-linux-gnu-objdump -d --start-address=0x401a3c --stop-address=0x402044  badbios2.bin|grep str
  401a5c:       f900f401        str     x1, [x0,#488]
  401a94:       f9002401        str     x1, [x0,#72]
  401ac8:       f9000401        str     x1, [x0,#8]
  401af0:       f9014801        str     x1, [x0,#656]
  401b24:       f9001801        str     x1, [x0,#48]
  401b60:       f9006401        str     x1, [x0,#200]
  401b98:       f900b001        str     x1, [x0,#352]
  401bcc:       f9001401        str     x1, [x0,#40]
  401bf4:       f9009c01        str     x1, [x0,#312]
  401c28:       f9013401        str     x1, [x0,#616]
  401c4c:       f900d801        str     x1, [x0,#432]
  401c78:       f9000001        str     x1, [x0]
  401ca8:       f9004401        str     x1, [x0,#136]
  401ce0:       f9003001        str     x1, [x0,#96]
  401d10:       f9003001        str     x1, [x0,#96]
  401d3c:       f9002401        str     x1, [x0,#72]
  401d74:       f9000001        str     x1, [x0]
  401da8:       f9006001        str     x1, [x0,#192]
  401dc8:       f900e001        str     x1, [x0,#448]
  401dec:       f9016401        str     x1, [x0,#712]
  401e20:       f9000401        str     x1, [x0,#8]
  401e58:       f9003401        str     x1, [x0,#104]
  401e9c:       f9003401        str     x1, [x0,#104]
  401ed4:       f9002801        str     x1, [x0,#80]
  401f08:       f9007401        str     x1, [x0,#232]
  401f40:       f9007401        str     x1, [x0,#232]
  401f68:       f9002001        str     x1, [x0,#64]
  401f9c:       f900d401        str     x1, [x0,#424]
  401fcc:       f9003801        str     x1, [x0,#112]
  402000:       f9000001        str     x1, [x0]
  402034:       f900c401        str     x1, [x0,#392]
```

Il ne reste plus qu'à poser 31 points d'arrêts et examiner l'état des registres à chaque fois :

```
(gdb) break *0x401a5c
Breakpoint 14 at 0x401a5c
(gdb) break *0x401a94
Breakpoint 15 at 0x401a94
(gdb) break *0x401ac8
Breakpoint 16 at 0x401ac8
```

```
[.. .]
(gdb) cont
Continuing.

Breakpoint 14, 0x0000000000401a5c in ?? ()
(gdb) p $x1
$4 = 0x4005fc
(gdb) p $x0 + 488
$5 = 0x4000801440
(gdb) cont
Continuing.

Breakpoint 15, 0x0000000000401a94 in ?? ()
(gdb) p $x1
$6 = 0x4005f4
(gdb) p $x0 + 72
$7 = 0x4000801438
(gdb) cont
Continuing.

Breakpoint 16, 0x0000000000401ac8 in ?? ()
(gdb) p $x1
$8 = 0x400dac
(gdb) p $x0 + 8
$9 = 0x4000801368
[...]
```

En passant les 31 points d'arrêt, il est possible de se rendre compte que l'adresse de destination est toujours comprise entre `0x4000801360` et `0x4000801450`. Il est alors aussi simple d'afficher le contenu de cette zone mémoire à la fin de la fonction.

```
(gdb) break *0x402044
Breakpoint 47 at 0x402044
(gdb) cont
Continuing.

Breakpoint 47, 0x0000000000402044 in ?? ()
(gdb) x/32gx ($x0+0x360)
0x4000801360:   0x0000000000400d9c      0x0000000000400dac
0x4000801370:   0x0000000000401580      0x0000000000401634
0x4000801380:   0x00000000004016e4      0x0000000000401030
0x4000801390:   0x00000000004010ec      0x00000000004011b4
0x40008013a0:   0x0000000000401794      0x0000000000400d58
0x40008013b0:   0x0000000000400c90      0x0000000000400c20
0x40008013c0:   0x0000000000400bd0      0x0000000000400b78
0x40008013d0:   0x0000000000400b04      0x0000000000400a8c
0x40008013e0:   0x0000000000400a08      0x0000000000400978
0x40008013f0:   0x0000000000400918      0x00000000004008c4
0x4000801400:   0x0000000000400864      0x00000000004007ec
0x4000801410:   0x0000000000400d24      0x0000000000400ce0
0x4000801420:   0x0000000000401970      0x00000000004018d0
0x4000801430:   0x000000000040187c      0x00000000004005f4
0x4000801440:   0x00000000004005fc      0x0000000000401490
0x4000801450:   0x000000000040077c      0x0000000000000000
```

Le rôle de la fonction `sub_401a08` est donc d'initialiser ce tableau à l'adresse `arg0 + 0x360`. La fonction retourne ensuite dans la fonction parente, `sub_402960`. On peut remarquer que toutes les valeurs du tableau correspondent à des adresses appartenant à la section `.text`, donc probablement des pointeurs de fonction.

Analyse de la fonction `sub_400404`

La fonction `sub_400404` est composée d'une seule instruction :

```
400404:    d65f03c0    ret
```

Cette fonction ne fait donc que retourner vers la fonction parente.

Analyse de la fonction `sub_400408`

Le code de la fonction `sub_400408` est présenté ci-dessous :

```
400408:    b9400022    ldr    w2, [x1]              // w2 = *(0x510000) = 0x5b1ad0b
40040c:    528f0ca5    mov    w5, #0x7865           // w5 = 0x7865
400410:    b9001002    str    w2, [x0,#16]          // *(addr0 + 32) = 0x5b1ad0b
400414:    b9400422    ldr    w2, [x1,#4]           // w2 = *(0x510004) = 0x5b1ad0b
400418:    528c8dc4    mov    w4, #0x646e           // w4 = 0x646e
40041c:    b9001402    str    w2, [x0,#20]          // *(addr0 + 36) = 0x5b1ad0b
400420:    b9400822    ldr    w2, [x1,#8]           // w2 = *(0x510008) = 0x5b1ad0b
400424:    5285a6c3    mov    w3, #0x2d36           // w3 = 0x2d36
400428:    b9001802    str    w2, [x0,#24]          // *(addr0 + 40) = 0x5b1ad0b
40042c:    b9400c22    ldr    w2, [x1,#12]          // w2 = *(0x51000c) = 0x5b1ad0b
400430:    72ac2e05    movk    w5, #0x6170, lsl #16 // x5 = 0x61707865
400434:    b9001c02    str    w2, [x0,#28]          // *(addr0 + 44) = 0x5b1ad0b
400438:    b9400026    ldr    w6, [x1]              // w6 = *(0x510000) = 0x5b1ad0b
40043c:    528cae82    mov    w2, #0x6574           // w2 = 25972
400440:    b9002006    str    w6, [x0,#32]          // *(addr0 + 48) = 0x5b1ad0b
400444:    b9400426    ldr    w6, [x1,#4]           // w6 = *(0x510004) = 0x5b1ad0b
400448:    72a62404    movk    w4, #0x3120, lsl #16 // x4 = 0x3120646e
40044c:    b9002406    str    w6, [x0,#36]          // *(addr0 + 52) = 0x5b1ad0b
400450:    b9400826    ldr    w6, [x1,#8]           // w6 = *(0x510008) = 0x5b1ad0b
400454:    72af2c43    movk    w3, #0x7962, lsl #16 // x3 = 0x79622d36
400458:    b9002806    str    w6, [x0,#40]          // *(addr0 + 56) = 0x5b1ad0b
40045c:    b9400c21    ldr    w1, [x1,#12]          // w1 = *(0x51000c) = 0x5b1ad0b
400460:    72ad6402    movk    w2, #0x6b20, lsl #16 // x2 = 0x6b206574
400464:    b9002c01    str    w1, [x0,#44]          // *(addr0 + 60) = 0x5b1ad0b
400468:    b9000005    str    w5, [x0]              // *(addr0 + 16) = 0x61707865
40046c:    b9000404    str    w4, [x0,#4]           // *(addr0 + 20) = 0x3120646e
400470:    b9000803    str    w3, [x0,#8]           // *(addr0 + 24) = 0x79622d36
400474:    b9000c02    str    w2, [x0,#12]          // *(addr0 + 28) = 0x6b206574
400478:    d65f03c0    ret
```

Cette fonction initialise un bloc mémoire à l'adresse `addr0 + 16`. Il est possible d'afficher le résultat final avec GDB :

```
$ (gdb) break *0x400478
Breakpoint 52 at 0x400478
(gdb) cont
Continuing.

Breakpoint 52, 0x0000000000400478 in ?? ()
(gdb) x/12wx $x0
0x4000801010:   0x61707865      0x3120646e      0x79622d36      0x6b206574
0x4000801020:   0x05b1ad0b      0x05b1ad0b      0x05b1ad0b      0x05b1ad0b
0x4000801030:   0x05b1ad0b      0x05b1ad0b      0x05b1ad0b      0x05b1ad0b
(gdb) x/s $x0
0x4000801010:   "expand 16-byte k\v\255\261\[...]"
```

On peut remarquer que le début du bloc est constitué de la chaîne de caractères « expand 16-byte k ».

Analyse de la fonction `sub_40047c`

Le code de la fonction `sub_40047c` est constitué des instructions ci-dessous :

```
  40047c:    b900301f     str    wzr, [x0,#48]  // *(addr0 + 64) = 0
  400480:    b900341f     str    wzr, [x0,#52]  // *(addr0 + 68) = 0
  400484:    b9400022     ldr    w2, [x1]       // w2 = *(0x510020) = 0
  400488:    b9003802     str    w2, [x0,#56]   // *(addr0 + 72) = 0
  40048c:    b9400421     ldr    w1, [x1,#4]    // w1 = *(0x510024) = 0
  400490:    b9003c01     str    w1, [x0,#60]   // *(addr0 + 76) = 0
  400494:    d65f03c0     ret
```

Cette fonction initialise à zéro le bloc de données à l'adresse `addr0 + 64` Le résultat est alors visible sous GDB :

```
(gdb) x/16wx 0x4000801010
0x4000801010:   0x61707865      0x3120646e      0x79622d36      0x6b206574
0x4000801020:   0x05b1ad0b      0x05b1ad0b      0x05b1ad0b      0x05b1ad0b
0x4000801030:   0x05b1ad0b      0x05b1ad0b      0x05b1ad0b      0x05b1ad0b
0x4000801040:   0x00000000      0x00000000      0x00000000      0x00000000
```

2.3.2 Analyse de la fonction `sub_402914`

Le code de la fonction `sub_402914` est présenté ci-dessous :

```
  402914:    a9be7bfd     stp    x29, x30, [sp,#-32]!
  402918:    910003fd     mov    x29, sp
  40291c:    39400001     ldrb    w1, [x0]             // w1 = *(addr0) = 0
  402920:    a90153f3     stp    x19, x20, [sp,#16]
  402924:    32000021     orr    w1, w1, #0x1         // w1 = 1
  402928:    39000001     strb    w1, [x0]            // *(addr0) = 1
  40292c:    aa0003f3     mov    x19, x0              // x19 = addr0
  402930:    97ffff89     bl    0x402754             // w0 = sub_402754(addr0)
  402934:    2a0003f4     mov    w20, w0              // w20 = w0
  402938:    340000c0     cbz    w0, 0x402950        // saute si w0 == 0
  40293c:    b9800661     ldrsw    x1, [x19,#4]
  402940:    90000000     adrp    x0, 0x402000
  402944:    912ce000     add    x0, x0, #0xb38
  402948:    f8617800     ldr    x0, [x0,x1,lsl #3]
  40294c:    97fffdc9     bl    0x402070
  402950:    2a1403e0     mov    w0, w20             // retourne la valeur de w20
  402954:    a94153f3     ldp    x19, x20, [sp,#16]
  402958:    a8c27bfd     ldp    x29, x30, [sp],#32
  40295c:    d65f03c0     ret
```

La fonction va simplement initialiser à 1 la valeur à l'adresse `addr0` puis appeler la fonction `sub_402754`.

2.3.3 Analyse de la fonction `sub_402754`

Le début du code de la fonction `sub_402754` est présenté ci-dessous :

```
  402754:    a9ba7bfd     stp    x29, x30, [sp,#-96]!
  402758:    910003fd     mov    x29, sp
  40275c:    a90153f3     stp    x19, x20, [sp,#16]
  402760:    aa0003f3     mov    x19, x0               // x19 = addr0
  402764:    39400000     ldrb    w0, [x0]             // w0 = *(addr0) = 1
  402768:    a9025bf5     stp    x21, x22, [sp,#32]
  40276c:    a90363f7     stp    x23, x24, [sp,#48]
  402770:    f90023f9     str    x25, [sp,#64]
  402774:    360007a0     tbz    w0, #0, 0x402868       // saute si le premier bit de w0 vaut 0
  402778:    d2800059     mov    x25, #0x2              // x25 = 2
  40277c:    d2800098     mov    x24, #0x4              // x24 = 4
  402780:    529ffff7     mov    w23, #0xffff           // w23 = 65535
  402784:    1400000d     b    0x4027b8
```

```
[...]
4027b8:    aa1303e0    mov    x0, x19
4027bc:    97ffffc4    bl     0x4026cc                // x0 = sub_4026cc(addr0)
4027c0:    aa0003f4    mov    x20, x0                 // x20 = x0
4027c4:    b100069f    cmn    x20, #0x1               // compare x20 à -1
4027c8:    aa0003e1    mov    x1, x0                  // x1 = x0
4027cc:    910173a2    add    x2, x29, #0x5c          // x2 = x29 + 92 (adresse sur la pile)
4027d0:    aa1303e0    mov    x0, x19                 // x0 = addr0
4027d4:    d2800023    mov    x3, #0x1                // x3 = 1
4027d8:    54000700    b.eq    0x4028b8               // saute si x20 == -1

4027dc:    97fffeb4    bl     0x4022ac                // w0 = sub_4022ac(addr0, x1, x29 + 92, 1)
4027e0:    2a0003f5    mov    w21, w0                 // w21 = w0
4027e4:    710006bf    cmp    w21, #0x1               // compare w21 à 1
4027e8:    aa1403e1    mov    x1, x20                 // x1 = x20
4027ec:    910163a2    add    x2, x29, #0x58          // x2 = x29 + 88 (adresse sur la pile)
4027f0:    aa1303e0    mov    x0, x19                 // x0 = addr0
4027f4:    54000621    b.ne    0x4028b8               // saute si w21 != 1
4027f8:    394173a3    ldrb    w3, [x29,#92]          // w3 = *(x29 + 92)
4027fc:    71007c7f    cmp    w3, #0x1f               // compare w3 à 0x1f
402800:    54000448    b.hi    0x402888               // saute si w3 > 0x1f
```

La fonction appelle successivement les sous-fonctions `sub_4026cc` et `sub_4022ac`.

L'exécution continue à l'adresse `0x402804` :

```
402804:    7100207f    cmp    w3, #0x8
402808:    9a988336    csel    x22, x25, x24, hi // x22 = (w3 > 8) ? 2 : 4
40280c:    aa1603e3    mov    x3, x22            // x3 = x22
402810:    b9005bbf    str    wzr, [x29,#88]     // *(x29 + 88) = 0
402814:    97fffea6    bl     0x4022ac           // w0 = sub_4022ac(addr0, x1, x29 + 88, x3)
402818:    93407c00    sxtw    x0, w0            // x0 = w0 (avec extension du signe)
40281c:    eb16001f    cmp    x0, x22            // compare x0 à x22 (2 ou 4)
402820:    8b140014    add    x20, x0, x20       // x20 += x0
402824:    54000621    b.ne    0x4028e8          // saute si x0 != x22
402828:    b90053b4    str    w20, [x29,#80]     // *(x29 + 80) = w20
40282c:    6b17029f    cmp    w20, w23           // compare w20 à  0xffff
402830:    54fffac8    b.hi    0x402788          // saute si w20 > 0xffff
402834:    d2800781    mov    x1, #0x3c          // x1 = 0x3c
402838:    910143a2    add    x2, x29, #0x50     // x2 = x29 + 80
40283c:    d2800083    mov    x3, #0x4           // x3 = 4
402840:    aa1303e0    mov    x0, x19            // x0 = addr0
402844:    97fffec8    bl     0x402364           // sub_402364(addr0, 0x3c, x29 + 80, 4)
```

La fonction `sub_4022ac` est appelée de nouveau, cette fois avec comme dernier paramètre 2 ou 4. La valeur de retour est comparée à ce dernier paramètre, l'exécution continue si les deux valeurs sont identiques. Ensuite, la fonction `sub_402364` est appelée.

L'exécution continue avec les instructions suivantes :

```
402848:    394173a0    ldrb    w0, [x29,#92]       // w0 = *(x29 + 92)
40284c:    b9405ba1    ldr    w1, [x29,#88]        // w1 = *(x29 + 88)
402850:    9101b000    add    x0, x0, #0x6c        // x0 = w0 + 0x6c
402854:    f8607a62    ldr    x2, [x19,x0,lsl #3]  // x2 = *(x19 + 8 * x0)
402858:    aa1303e0    mov    x0, x19              // x0 = addr0
40285c:    d63f0040    blr    x2                   // branche sur la valeur de x2
402860:    39400260    ldrb    w0, [x19]           // w0 = *(addr0)
402864:    3707faa0    tbnz    w0, #0, 0x4027b8    // teste le premier bit de w0, saute si différent de 0
402868:    b9400675    ldr    w21, [x19,#4]        // w21 = *(addr0 + 4)
40286c:    a94363f7    ldp    x23, x24, [sp,#48]
402870:    2a1503e0    mov    w0, w21              // retourne w21
402874:    a94153f3    ldp    x19, x20, [sp,#16]
402878:    a9425bf5    ldp    x21, x22, [sp,#32]
40287c:    f94023f9    ldr    x25, [sp,#64]
```

```
  402880:    a8c67bfd     ldp    x29, x30, [sp],#96
  402884:    d65f03c0     ret
```

Le programme continue en appelant une fonction dont l'adresse est contenue dans le registre `x2`. Cette adresse est déterminée par rapport à la valeur stockée à l'adresse `0x29 + 92`.

Le code de la fonction `sub_402754` est équivalent au pseudo-code C ci-dessous :

```
int sub_402754(char *addr0) {
    while ((*addr0 & 1) != 0) {
        x20 = sub_4026cc(addr0);
        if (x1 == -1) {
            /* loc_4028b8 */
        }

        ret = sub_4022ac(addr0, x20, &var_92, 1);
        if (ret != 1) {
            /* loc_4028b8 */
        }

        if (var_92 > 0x1f) {
            /* loc_402888 */
        }
        w3 = (var_92 > 8) ? 2 : 4;
        ret = sub_4022ac(addr0, x20, &var_88, w3);
        if (ret != w3) {
            /* loc_4028e8 */
        }

        x20 += ret;
        var_80 = x20;
        if (x20 > 0xffff) {
            /* loc_402788 */
        }

        sub_402364(addr0, 0x3c, &var_80, 4);
        x2 = *(addr0 + 0x360 + 8 * var_92);
        (*x2)(addr0, var_88);
    }

    return *(addr0 + 4);
}
```

Ce code fait penser au fonctionnement d'une machine virtuelle :

— la fonction `sub_4022ac` stocke le numéro de la fonction à appeler dans la variable `var_92` ;
— de même, le paramètre de la fonction est stocké dans la variable `var_88` ;
— l'exécution continue tant que la condition `*addr0 & 1 != 0` est vraie.

La variable `var_92` correspondrait alors à l'opcode de l'instruction et la variable `var_88` à l'opérande.

2.4 Analyse de la machine virtuelle

2.4.1 Analyse de la fonction `sub_4026cc`

Le code de la fonction `sub_4026cc` est présenté ci-dessous :

```
  4026cc:    a9be7bfd     stp    x29, x30, [sp,#-32]!
  4026d0:    910003fd     mov    x29, sp
  4026d4:    d2800781     mov    x1, #0x3c                 // x1 = 0x3c
```

```
4026d8:    910043a2    add    x2, x29, #0x10             // x2 = x29 + 16
4026dc:    d2800083    mov    x3, #0x4                   // x3 = 4
4026e0:    97fffef3    bl    0x4022ac                    // sub_4022ac(addr0, 0x3c, x29 + 16, 4)
4026e4:    7100101f    cmp    w0, #0x4
4026e8:    92800000    mov    x0, #0xffffffffffffffff    // retourne -1 si w0 != 4
4026ec:    54000060    b.eq    0x4026f8
4026f0:    a8c27bfd    ldp    x29, x30, [sp],#32
4026f4:    d65f03c0    ret

4026f8:    b94013a0    ldr    w0, [x29,#16]              // retourne la valeur *(x29 + 16)
4026fc:    a8c27bfd    ldp    x29, x30, [sp],#32
402700:    d65f03c0    ret
```

La fonction va appeler la fonction `sub_4022ac` et retourner la valeur stockée sur la pile à l'adresse `x29 + 16`.

2.4.2 Analyse de la fonction `sub_4022ac`

Le code de la fonction `sub_4022ac` est présenté ci-dessous :

```
4022ac:    a9bc7bfd    stp    x29, x30, [sp,#-64]!
4022b0:    910003fd    mov    x29, sp
4022b4:    a90153f3    stp    x19, x20, [sp,#16]
4022b8:    a9025bf5    stp    x21, x22, [sp,#32]
4022bc:    a90363f7    stp    x23, x24, [sp,#48]
4022c0:    aa0303f4    mov    x20, x3                 // x20 = arg3
4022c4:    aa0003f7    mov    x23, x0                 // x23 = arg0 = addr0
4022c8:    aa0103f5    mov    x21, x1                 // x21 = arg1
4022cc:    aa0203f3    mov    x19, x2                 // x19 = arg2
4022d0:    b4000463    cbz    x3, 0x40235c            // retourne 0 si x3 == 0
4022d4:    d2800016    mov    x22, #0x0               // x22 = 0
4022d8:    d2800818    mov    x24, #0x40              // x24 = 64

4022dc:    aa1703e0    mov    x0, x23                 // x0 = addr0
4022e0:    aa1503e1    mov    x1, x21                 // x1 = x21
4022e4:    97ffff78    bl    0x4020c4                 // x0 = sub_4020c4(addr0, x21)
4022e8:    b40002e0    cbz    x0, 0x402344            // retourne w22 si x0 == 0
4022ec:    924016a2    and    x2, x21, #0x3f          // x2 = x21 & 0x3f (reste de la division par 64)
4022f0:    cb020309    sub    x9, x24, x2             // x9 = 64 - (x21 & 0x3f)
4022f4:    eb14013f    cmp    x9, x20                 // compare x9 et x20
4022f8:    9a949129    csel    x9, x9, x20, ls        // x9 = (x9 <= x20) ? x9 : x20
4022fc:    8b020002    add    x2, x0, x2              // x2 += x0
402300:    b4000189    cbz    x9, 0x402330            // saute si x9 == 0
402304:    91000528    add    x8, x9, #0x1            // x8 = x9 + 1
402308:    d2800026    mov    x6, #0x1                // x6 = 1
40230c:    d2800004    mov    x4, #0x0                // x4 = 0
402310:    14000003    b    0x40231c

402314:    aa0603e4    mov    x4, x6                  // x4 = x6
402318:    aa0503e6    mov    x6, x5                  // x6 = x5

40231c:    38646847    ldrb    w7, [x2,x4]            // w7 = *(x2 + x4)
402320:    910004c5    add    x5, x6, #0x1            // x5 = x6 + 1
402324:    38246a67    strb    w7, [x19,x4]           // *(x19 + x4) = w7
402328:    eb0800bf    cmp    x5, x8                  // compare x5 à x8
40232c:    54ffff41    b.ne    0x402314               // saute si différent

402330:    eb090294    subs    x20, x20, x9           // x20 -= x9
402334:    8b0902d6    add    x22, x22, x9            // x22 += x9
402338:    8b090273    add    x19, x19, x9            // x19 += x9
40233c:    8b0902b5    add    x21, x21, x9            // x21 += x9
402340:    54fffce1    b.ne    0x4022dc               // saute si x20 != 0

402344:    2a1603e0    mov    w0, w22                 // retourne w22
```

```
402348:    a94153f3    ldp    x19, x20, [sp,#16]
40234c:    a9425bf5    ldp    x21, x22, [sp,#32]
402350:    a94363f7    ldp    x23, x24, [sp,#48]
402354:    a8c47bfd    ldp    x29, x30, [sp],#64
402358:    d65f03c0    ret

40235c:    2a0303e0    mov    w0, w3
402360:    17fffffa    b    0x402348
```

Cette fonction est construite autour de deux boucles imbriquées :

- une première qui réalise des appels successifs à la fonction `sub_4020c4`. Cette fonction retourne une valeur en fonction du paramètre `x21`. ;
- la seconde boucle copie le nombre d'octets spécifiés par le registre `x9` depuis l'adresse contenue dans le registre `x2` vers l'adresse du registre `x19` ;
- les registres `x19` jusqu'à `x20` sont mis à jour en fonction du nombre d'octets copiés précédemment ;
- la fonction refait une itération si le nombre total d'octets copiés est inférieur au quatrième paramètre de la fonction.

2.4.3 Analyse de la fonction `sub_402364`

Le code de la fonction `sub_402364` est présenté ci-dessous.

```
402364:    a9bc7bfd    stp    x29, x30, [sp,#-64]!
402368:    910003fd    mov    x29, sp
40236c:    a90153f3    stp    x19, x20, [sp,#16]
402370:    a9025bf5    stp    x21, x22, [sp,#32]
402374:    a90363f7    stp    x23, x24, [sp,#48]
402378:    aa0303f4    mov    x20, x3              // x20 = arg3
40237c:    aa0003f6    mov    x22, x0              // x22 = arg0
402380:    aa0103f5    mov    x21, x1              // x21 = arg1
402384:    aa0203f3    mov    x19, x2              // x19 = arg2
402388:    b4000563    cbz    x3, 0x402434         // saute si arg3 == 0
40238c:    d2800018    mov    x24, #0x0            // x24 = 0
402390:    d2800817    mov    x23, #0x40           // x23 = 64

402394:    aa1603e0    mov    x0, x22              // x0 = arg0
402398:    aa1503e1    mov    x1, x21              // x1 = arg1
40239c:    97ffff4a    bl    0x4020c4              // x0 = sub_4020c4(arg0, arg1)
4023a0:    b40003e0    cbz    x0, 0x40241c        // saute si x0 == 0
4023a4:    924016a2    and    x2, x21, #0x3f      // x2 = x21 & 0x3f
4023a8:    cb0202e9    sub    x9, x23, x2         // x9 = 64 - (x21 & 0x3f)
4023ac:    eb14013f    cmp    x9, x20             // compare x9 et x20
4023b0:    9a949129    csel    x9, x9, x20, ls    // x9 = (x9 <= x20) ? x9 : x20
4023b4:    8b020002    add    x2, x0, x2          // x2 += x0
4023b8:    b4000189    cbz    x9, 0x4023e8        // saute si x9 == 0
4023bc:    91000528    add    x8, x9, #0x1        // x8 = x9 + 1
4023c0:    d2800026    mov    x6, #0x1            // x6 = 1
4023c4:    d2800004    mov    x4, #0x0            // x4 = 0
4023c8:    14000003    b    0x4023d4

4023cc:    aa0603e4    mov    x4, x6              // x4 = x6
4023d0:    aa0503e6    mov    x6, x5              // x6 = x5

4023d4:    38646a67    ldrb    w7, [x19,x4]       // w7 = *(x19 + x4)
4023d8:    910004c5    add    x5, x6, #0x1        // x5 = x6 + 1
4023dc:    38246847    strb    w7, [x2,x4]        // *(x2 + x4) = w7
4023e0:    eb0800bf    cmp    x5, x8              // compare x5 à x8
4023e4:    54ffff41    b.ne    0x4023cc           // saute si différent

4023e8:    f941aec1    ldr    x1, [x22,#856]      // x1 = *(addr0 + 0x358) = addr2
4023ec:    eb090294    subs    x20, x20, x9       // x20 -= x9
```

```
4023f0:    cb010000     sub    x0, x0, x1           // x0 -= x1
4023f4:    9346fc00     asr    x0, x0, #6           // x0 = x0 / 64 (index)
4023f8:    8b000400     add    x0, x0, x0, lsl #1   // x0 = 3 * x0
4023fc:    8b000ec1     add    x1, x22, x0, lsl #3  // x1 = addr0 + 8 * x0
402400:    3941a020     ldrb    w0, [x1,#104]       // w0 = *(x1 + 0x68)
402404:    8b090318     add    x24, x24, x9         // x24 += x9
402408:    32000000     orr    w0, w0, #0x1         // w0 |= 1
40240c:    3901a020     strb    w0, [x1,#104]       // *(x1 + 0x68) = w0
402410:    8b090273     add    x19, x19, x9         // x19 += x9
402414:    8b0902b5     add    x21, x21, x9         // x21 += x9
402418:    54fffbe1     b.ne    0x402394            // saute si x20 != 0

40241c:    2a1803e0     mov    w0, w24
402420:    a94153f3     ldp    x19, x20, [sp,#16]
402424:    a9425bf5     ldp    x21, x22, [sp,#32]
402428:    a94363f7     ldp    x23, x24, [sp,#48]
40242c:    a8c47bfd     ldp    x29, x30, [sp],#64
402430:    d65f03c0     ret
```

Cette fonction est très similaire à la fonction `sub_4022ac`. Cependant, on remarque que le sens de la copie de données est inversé, c'est-à-dire qu'il s'effectue depuis l'adresse sur la pile passée par le registre `x2` vers l'adresse obtenue via la fonction `sub_4020c4`.

De plus, le tableau de structures à l'adresse `addr0 + 0x358` est mis à jour en fonction de l'adresse retournée par la fonction `sub_4020c4` qui détermine l'index dans le tableau

2.4.4 Analyse de la fonction `sub_4020c4`

La fonction `sub_4020c4` est assez complexe et ne sera pas présentée dans le détail. Cependant, le résultat de sa rétroconception est présenté ci-dessous :

```
char *mmu_handle(char *addr, uint32_t vm_addr) {
    uint64_t u1, u9, block_start, b3, b8;
    int i;
    char *src, *dst;

    for (i = 0; i < 32; i++) {
        if (((mmu[i].flags >> 1) & 1) == 0) {
            if (mmu[i].idx == (vm_addr / 64)) {
                dst = addr2 + 64 * i;
                mmu[i].addr = *((uint64_t *) (addr + 8));
                if (dst != 0)
                    return dst;
                else
                    break;
            }
        }
    }

    block_start = (vm_addr / 64) * 64;
    if (block_start > 0xffff)
        return NULL;

    b8 = -1;
    u9 = 0;
    for (i = 0; i < 32; i++) {
        if (((mmu[i].flags >> 1) & 1) == 0) {
            if (mmu[i].addr >= b8) {
                u9 = i;
                b8 = mmu[i].addr;
            }
        } else {
```

```
            dst = addr2 + 64 * i;
            mmu[i].flags &= 0xFFFFFFFC;
            mmu[i].idx = vm_addr / 64;
            mmu[i].addr = *((uint64_t *) (addr + 8));

            *((uint32_t *) (addr + 0x40)) = mmu[i].idx;
            *((uint32_t *) (addr + 0x44)) = 0;
            sub_4004a4(addr + 0x10, addr1 + block_start, dst, 64);
            return dst;
        }
    }

    dst = addr2 + 64 * u9;
    if ((mmu[u9].flags & 1) != 0) {
        u1 = mmu[u9].idx * 64;
        if (u1 <= 0xffff) {
            src = addr1 + (int32_t) u1;
            if (src != 0) {
                *((uint32_t *) (addr + 0x40)) = mmu[u9].idx;
                *((uint32_t *) (addr + 0x44)) = 0;

                sub_400498(addr + 0x10, dst, src, 64);
            }
        }
    }

    mmu[u9].flags &= 0xFFFFFFFC;
    mmu[u9].idx = vm_addr / 64;
    mmu[u9].addr = *((uint64_t *) (addr + 8));

    *((uint32_t *) (addr + 0x40)) = mmu[u9].idx;
    *((uint32_t *) (addr + 0x44)) = 0;
    sub_4004a4(addr + 0x10, addr1 + block_start, dst, 64);
    return dst;
}
```

Cette fonction implémente la gestion de la mémoire de la machine virtuelle. Elle prend en argument une adresse `vm_addr`, va charger le bloc mémoire qui contient l'adresse demandée et retourne l'adresse du bloc.

Un tableau de 32 structures (initialisé précédemment par la fonction `sub_402960`) permet de savoir les blocs qui ont déjà été précédemment chargés, en fonction du champ `flags` de la structure. Ce champ peut être interprété de la façon suivante :

- si le second bit de poids faible est nul, alors le bloc correspondant a été chargé en mémoire ;
- si le second bit de poids faible est non nul, alors aucun bloc n'a été chargé en mémoire pour cette structure ;
- si le premier bit de poids faible est nul, alors le bloc doit être sauvegardé.

L'opération de chargement d'un bloc est assuré par la fonction `sub_4004a4`, tandis que la fonction `sub_400498` se charge de sauvegarder un bloc.

2.4.5 Analyse des fonctions `sub_4004a4` et `sub_400498`

Le code de ces deux fonctions est présenté ci-dessous :

```
  400498:    34000043     cbz    w3, 0x4004a0
  40049c:    17ffff13     b    0x4000e8
  4004a0:    d65f03c0     ret

  4004a4:    35000043     cbnz    w3, 0x4004ac
  4004a8:    d65f03c0     ret
```

```
4004ac:    17ffff0f     b    0x4000e8
```

Ces deux fonctions vont simplement tester la valeur du paramètre `w3` et sauter à l'adresse `0x4000e8` si `w3 != 0`.

2.4.6 Analyse de `0x4000e8`

Le code à l'adresse `0x4000e8` ne sera pas analysé en détail dans cette solution car trop complexe. Néanmoins, il est possible d'examiner l'état des registres à cette adresse.

```
$ qemu-aarch64 -strace -g 1234 badbios2.bin
$ gdb-multiarch -q badbios2.bin
Reading symbols from badbios2.bin...(no debugging symbols found)...done.
(gdb) target remote 127.1:1234
Remote debugging using 127.1:1234
0x0000000000400514 in ?? ()
(gdb) break *0x4000e8
Breakpoint 1 at 0x4000e8
(gdb) cont
Continuing.

Breakpoint 1, 0x00000000004000e8 in ?? ()
(gdb) x/18x $x0
0x4000801010:   0x61707865      0x3120646e      0x79622d36      0x6b206574
0x4000801020:   0x05b1ad0b      0x05b1ad0b      0x05b1ad0b      0x05b1ad0b
0x4000801030:   0x05b1ad0b      0x05b1ad0b      0x05b1ad0b      0x05b1ad0b
0x4000801040:   0x00000000      0x00000000      0x00000000      0x00000000
0x4000801050:   0x00802000      0x00000040
(gdb) x/s $x0
0x4000801010:   "expand 16-byte k\v\25[...]"
(gdb) x/64bx $x1
0x4000802000:   0xa2    0xdb    0xe7    0x41    0xf2    0x76    0x3b    0x28
0x4000802008:   0x08    0x77    0x53    0xe2    0x39    0x39    0xc2    0xb0
0x4000802010:   0x6f    0xc3    0x10    0xcf    0xf9    0xe1    0x29    0x9d
0x4000802018:   0x5c    0x4e    0x33    0xb4    0x15    0x7a    0x41    0x20
0x4000802020:   0x99    0x5e    0xc9    0x8a    0x4d    0x15    0x55    0xf2
0x4000802028:   0xe8    0x88    0xc9    0x98    0x03    0xd7    0x29    0xba
0x4000802030:   0x00    0x2d    0x1e    0xb5    0x46    0x72    0x22    0x0d
0x4000802038:   0x0b    0x01    0xd4    0x6d    0x6a    0xe9    0x9a    0x8d
(gdb) x/64bx $x2
0x4000812000:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x4000812008:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x4000812010:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x4000812018:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x4000812020:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x4000812028:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x4000812030:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x4000812038:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb) p $x3
$1 = 0x40
```

A l'entrée de la fonction, les registres contiennent :

- `x0` : une adresse vers une zone mémoire initialisée par la fonction `sub_400408` ;
- `x1` : une adresse vers une zone mémoire appartenant à `addr1` (second appel à `mmap`) ;
- `x2` : une adresse vers une zone mémoire appartenant à `addr2` (troisième appel à `mmap`) ;
- `x3` : la valeur 64.

Une recherche Google sur la chaîne de caractères `expand 16-byte k` permet d'identifier l'algorithme de chiffrement `salsa20`, dont l'implémentation de référence peut être téléchargée à l'adresse http://cr.yp.to/snuffle/salsa20/merged/salsa20.c.

On peut alors tenter de déchiffrer le bloc de données pointé par le registre `x1` avec l'implémentation de référence. Le code C ci-dessous effectue cette opération :

```c
#include <stdio.h>
#include <stdlib.h>

#include "ecrypt-sync.h"

#ifndef HEXDUMP_COLS
#define HEXDUMP_COLS 8
#endif

u8 ciphertext[64];
u8 output[64];

int main(int argc, char **argv) {
  ECRYPT_ctx ctx;
  u64 block_count = 0;
  u8 key[16] = "\x0b\xad\xb1\x05\x0b\xad\xb1\x05\x0b\xad\xb1\x05\x0b\xad\xb1\x05";
  u8 iv[8];

  U64TO8_LITTLE(ciphertext,      0x283b76f241e7dba2);
  U64TO8_LITTLE(ciphertext +  8, 0xb0c23939e2537708);
  U64TO8_LITTLE(ciphertext + 16, 0x9d29e1f9cf10c36f);
  U64TO8_LITTLE(ciphertext + 24, 0x20417a15b4334e5c);
  U64TO8_LITTLE(ciphertext + 32, 0xf255154d8ac95e99);
  U64TO8_LITTLE(ciphertext + 40, 0xba29d70398c988e8);
  U64TO8_LITTLE(ciphertext + 48, 0x0d227246b51e2d00);
  U64TO8_LITTLE(ciphertext + 56, 0x8d9ae96a6dd4010b);

  ECRYPT_keysetup(&ctx, key, 128, 0);
  U64TO8_LITTLE(iv, block_count);
  ECRYPT_ivsetup(&ctx, iv);

  ECRYPT_decrypt_bytes(&ctx, ciphertext, output, 64);
  printf("ctx:\n");
  hexdump(&ctx, sizeof(ctx));

  printf("ciphertext:\n");
  hexdump(ciphertext, 64);

  printf("output:\n");
  hexdump(output, 64);

  exit(EXIT_SUCCESS);
}
```

Le résultat obtenu est alors :

```
$ gcc -o test_salsa test_salsa20.c salsa20.c
$ ./test_salsa
ctx:
0x000000: 65 78 70 61 0b ad b1 05 expa....
0x000008: 0b ad b1 05 0b ad b1 05 ........
0x000010: 0b ad b1 05 6e 64 20 31 ....nd 1
0x000018: 00 00 00 00 00 00 00 00 ........
0x000020: 01 00 00 00 00 00 00 00 ........
0x000028: 36 2d 62 79 0b ad b1 05 6-by....
0x000030: 0b ad b1 05 0b ad b1 05 ........
0x000038: 0b ad b1 05 74 65 20 6b ....te k
ciphertext:
0x000000: a2 db e7 41 f2 76 3b 28 ...A.v;(
0x000008: 08 77 53 e2 39 39 c2 b0 .wS.99..
0x000010: 6f c3 10 cf f9 e1 29 9d o.....).
0x000018: 5c 4e 33 b4 15 7a 41 20 \N3..zA
```

```
0x000020: 99 5e c9 8a 4d 15 55 f2 .^..M.U.
0x000028: e8 88 c9 98 03 d7 29 ba ......).
0x000030: 00 2d 1e b5 46 72 22 0d .-..Fr".
0x000038: 0b 01 d4 6d 6a e9 9a 8d ...mj...
output:
0x000000: ef 51 c3 f6 58 00 3f 6d .Q..X.?m
0x000008: 5b a3 bf e4 da 21 80 99 [....!..
0x000010: 61 43 a7 bc 81 dc 2f 29 aC..../)
0x000018: 40 57 1d ce 97 d7 19 10 @W......
0x000020: a9 e6 16 6e de f9 0d 93 ...n....
0x000028: 28 1c 65 08 21 c6 bb 29 (.e.!..)
0x000030: 61 cf e6 33 ca 7d b6 5b a..3.}.[
0x000038: 91 08 f1 ba bb 2e 9c 51 .......Q
```

Pour comparer avec l'implémentation du challenge, il est possible d'utiliser GDB et de poser un point d'arrêt à la fin de la fonction :

```
(gdb) break *0x400478
Breakpoint 2 at 0x400478
(gdb) cont
Continuing.

Breakpoint 1, 0x00000000004000e8 in ?? ()
(gdb) x/64bx 0x4000812000
0x4000812000:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x4000812008:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x4000812010:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x4000812018:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x4000812020:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x4000812028:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x4000812030:   0x00    0x00    0x00    0x00    0x00    0x20    0x00    0x00
0x4000812038:   0x00    0x00    0x00    0x00    0x40    0x00    0x00    0x00
```

On constate que les deux résultats n'ont rien à voir. De plus, le contexte cryptographique obtenu avec l'implémentation de référence de `salsa20` n'est semblable à celui observé via GDB : en particulier, la chaîne `expand 16-byte k` n'est pas stockée de manière contigüe dans l'implémentation de référence.

L'examen de la fonction `ECRYPT_keysetup` de `salsa20`, présentée ci-dessous, permet de confirmer cette hypothèse.

```
static const char sigma[16] = "expand 32-byte k";
static const char tau[16] = "expand 16-byte k";

void ECRYPT_keysetup(ECRYPT_ctx *x,const u8 *k,u32 kbits,u32 ivbits)
{
  const char *constants;

  x->input[1] = U8TO32_LITTLE(k + 0);
  x->input[2] = U8TO32_LITTLE(k + 4);
  x->input[3] = U8TO32_LITTLE(k + 8);
  x->input[4] = U8TO32_LITTLE(k + 12);
  if (kbits == 256) { /* recommended */
    k += 16;
    constants = sigma;
  } else { /* kbits == 128 */
    constants = tau;
  }
  x->input[11] = U8TO32_LITTLE(k + 0);
  x->input[12] = U8TO32_LITTLE(k + 4);
  x->input[13] = U8TO32_LITTLE(k + 8);
  x->input[14] = U8TO32_LITTLE(k + 12);
  x->input[0] = U8TO32_LITTLE(constants + 0);
  x->input[5] = U8TO32_LITTLE(constants + 4);
  x->input[10] = U8TO32_LITTLE(constants + 8);
```

```
  x->input[15] = U8TO32_LITTLE(constants + 12);
}
```

Le troisième résultat de la recherche Google permet d'identifier un autre algorithme nommé `chacha`, comme présenté sur la figure 2.2.

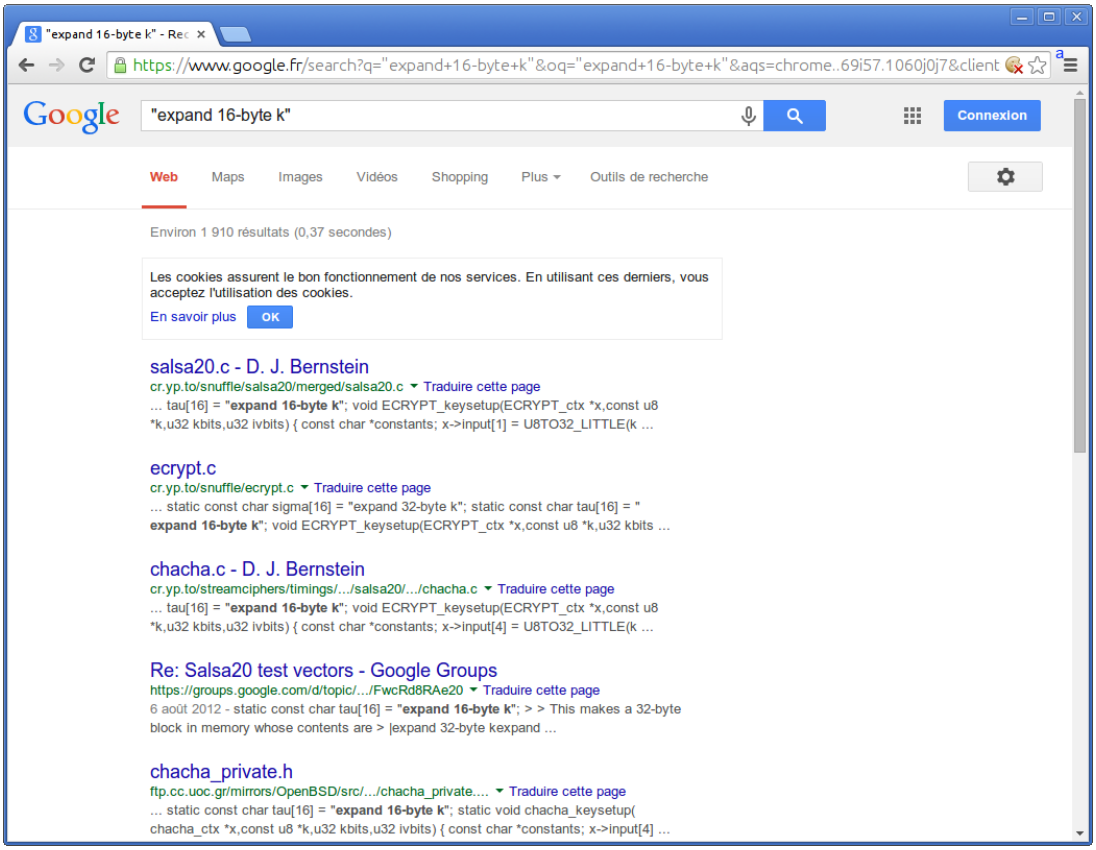

FIGURE 2.2 – Résultats de la recherche Google

Le fichier `chacha.c`[3] présente la même API que le fichier `salsa20.c` : il suffit de recompiler le programme `test_salsa20.c` pour effectuer un test.

```
$ gcc -o test_salsa test_salsa20.c chacha.c
$ ./test_salsa
ctx:
0x000000: 65 78 70 61 6e 64 20 31 expand 1
0x000008: 36 2d 62 79 74 65 20 6b 6-byte k
0x000010: 0b ad b1 05 0b ad b1 05 ........
0x000018: 0b ad b1 05 0b ad b1 05 ........
0x000020: 0b ad b1 05 0b ad b1 05 ........
0x000028: 0b ad b1 05 0b ad b1 05 ........
0x000030: 01 00 00 00 00 00 00 00 ........
0x000038: 00 00 00 00 00 00 00 00 ........
ciphertext:
0x000000: a2 db e7 41 f2 76 3b 28 ...A.v;(
0x000008: 08 77 53 e2 39 39 c2 b0 .wS.99..
0x000010: 6f c3 10 cf f9 e1 29 9d o.....).
0x000018: 5c 4e 33 b4 15 7a 41 20 \N3..zA
0x000020: 99 5e c9 8a 4d 15 55 f2 .^..M.U.
0x000028: e8 88 c9 98 03 d7 29 ba ......).
0x000030: 00 2d 1e b5 46 72 22 0d .-..Fr".
0x000038: 0b 01 d4 6d 6a e9 9a 8d ...mj...
output:
```

3. http://cr.yp.to/streamciphers/timings/estreambench/submissions/salsa20/chacha8/merged/chacha.c

```
0x000000: 00 00 00 00 00 00 00 00 ........
0x000008: 00 00 00 00 00 00 00 00 ........
0x000010: 00 00 00 00 00 00 00 00 ........
0x000018: 00 00 00 00 00 00 00 00 ........
0x000020: 00 00 00 00 00 00 00 00 ........
0x000028: 00 00 00 00 00 00 00 00 ........
0x000030: 00 00 00 00 00 20 00 00 ..... ..
0x000038: 00 00 00 00 40 00 00 00 ....@...
```

Cette fois, on obtient bien le même résultat que sous GDB : le binaire `badbios2.bin` utilise bien l'algorithme `chacha`.

2.4.7 Synthèse de l'analyse de la machine virtuelle

Maintenant que les fonctions principales du programme `badbios2.bin` ont été analysées, il est possible de mettre à jour le graphe d'appels en renommant les fonctions selon leur rôle. Le graphe mis à jour est représenté à la figure 2.3.

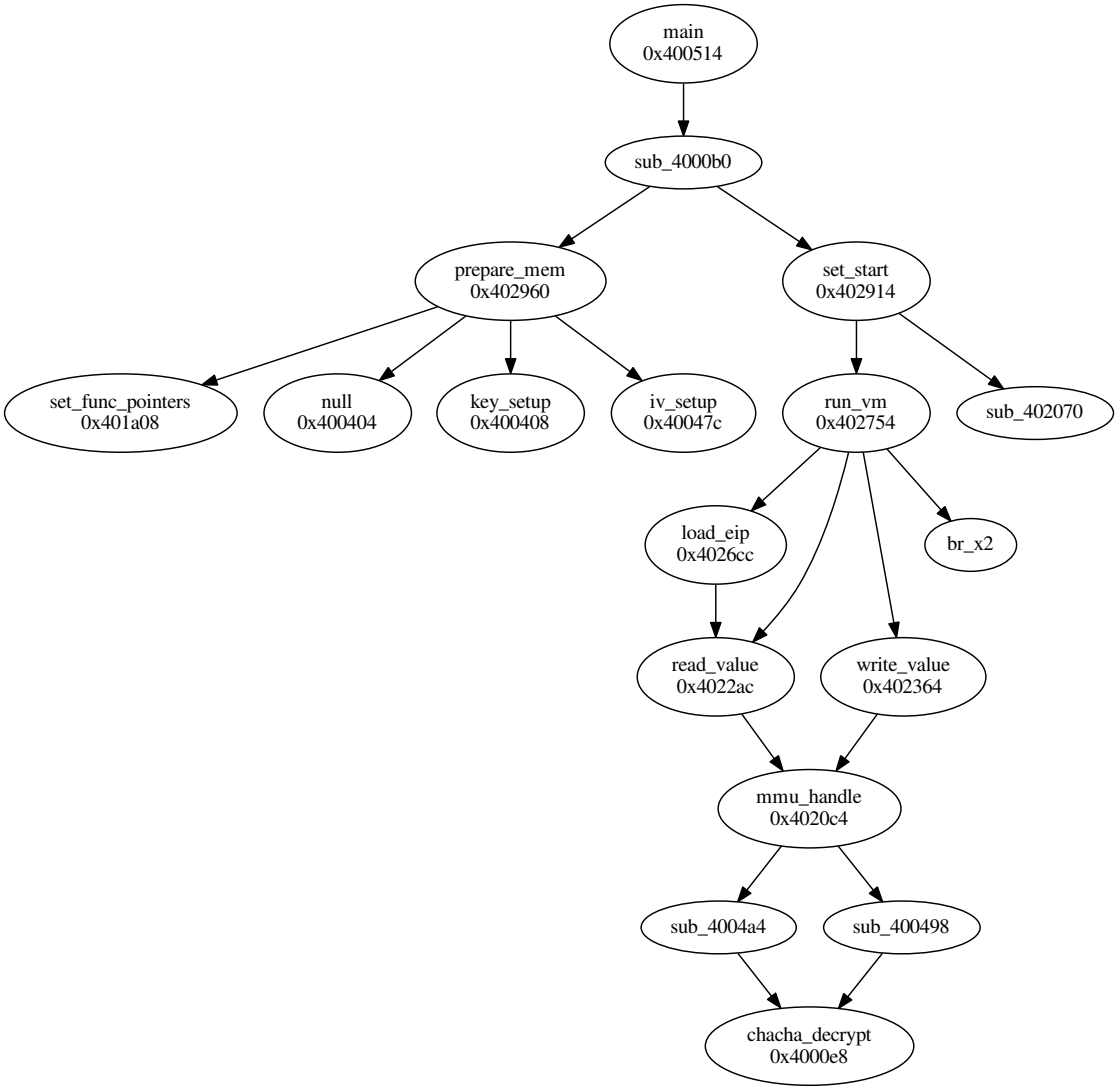

FIGURE 2.3 – Graphe d'appels de `badbios2.bin`

L'analyse des fonctions a permis de déterminer le mode de fonctionnement de la machine virtuelle :

- la fonction `prepare_mem` (`sub_402960`) réalise les opérations suivantes :
  - allocation d'une zone mémoire de 4096 octets (`addr0`) ,
  - allocation d'une zone mémoire de 65536 octets (`addr1`) qui stocke l'espace mémoire de la machine virtuelle mais sous forme chiffrée,
  - allocation d'une zone mémoire de 4096 octets (`addr2`) qui contient, sous forme déchiffrée, des blocs mémoire de 64 octets chargés par la machine virtuelle,
  - initialisation dans `addr0` d'un tableau de 32 structures utilisées pour le chargement des blocs mémoires ;
  - définition des pointeurs de fonctions correspondant à chaque instruction de la machine virtuelle,
  - mise en place du contexte cryptographique.
- la fonction `run_vm` effectue les opérations suivantes :
  - chargement de la valeur du registre `eip` de la machine virtuelle à l'adresse `0x3c` ;
  - lecture en mémoire de l'opcode correspondant ;
  - en fonction de l'opcode, lecture en mémoire de 2 ou 4 octets qui contiennent les données de l'instruction (opcode et opérande) ;
  - détermination de la fonction à exécuter en fonction de l'opcode ;
  - mise à jour du registre `eip` et exécution de la fonction.

Pour poursuivre l'analyse, il faut maintenant s'intéresser aux fonctions appelées par la machine virtuelle. Les adresses de ces fonctions sont définies par la fonction `set_func_pointers` (`sub_401a08`) et sont (avec l'opcode correspondant) :

- 0 : `sub_400d9c` ;
- 1 : `sub_400dac` ;
- 2 : `sub_401580` ;
- 3 : `sub_401634` ;
- 4 : `sub_4016e4` ;
- 5 : `sub_401030` ;
- 6 : `sub_4010ec` ;
- 7 : `sub_4011b4` ;
- 8 : `sub_401794` ;
- 9 : `sub_400d58` ;
- 10 : `sub_400c90` ;
- 11 : `sub_400c20` ;
- 12 : `sub_400bd0` ;
- 13 : `sub_400b78` ;
- 14 : `sub_400b04` ;
- 15 : `sub_400a8c` ;
- 16 : `sub_400a08` ;
- 17 : `sub_400978` ;
- 18 : `sub_400918` ;
- 19 : `sub_4008c4` ;
- 20 : `sub_400864` ;
- 21 : `sub_4007ec` ;
- 22 : `sub_400d24` ;
- 23 : `sub_400ce0` ;
- 24 : `sub_401970` ;
- 25 : `sub_4018d0` ;
- 26 : `sub_40187c` ;
- 27 : `sub_4005f4` ;
- 28 : `sub_4005fc` ;
- 29 : `sub_401490` ;
- 30 : `sub_40077c` ;

En réalité, en observant le fonctionnement de la machine virtuelle, on se rend compte que certaines de ces fonctions ne sont jamais appelées.

Le script GDB ci-dessous permet d'afficher l'adresse de chaque fonction exécutée par la machine virtuelle :

```
$ cat trace-ins.gdb
file badbios2.bin
target remote 127.1:1234

break *0x40285C // blr x2
commands
silent
printf "-> calling sub_%8.8x (index = %d), x0 = 0x%8.8x, arg_48 = 0x%8.8x\n", $x2, *($x29 + 0x5c), $x0, $x1
cont
end

cont
$ gdb-multiarch -q badbios2.bin < trace-ins.gdb
Reading symbols from badbios2.bin...(no debugging symbols found)...done.
(gdb) Reading symbols from badbios2.bin...(no debugging symbols found)...done.
(gdb) Remote debugging using 127.1:1234
0x0000000000400514 in ?? ()
(gdb) (gdb) Breakpoint 1 at 0x40285c
(gdb) >>>>(gdb) (gdb) Continuing.
-> calling sub_00400d9c (index = 0), x0 = 0x00801000, arg_48 = 0x00000100
-> calling sub_00400dac (index = 1), x0 = 0x00801000, arg_48 = 0x00002101
-> calling sub_00400d9c (index = 0), x0 = 0x00801000, arg_48 = 0x00000200
-> calling sub_00400dac (index = 1), x0 = 0x00801000, arg_48 = 0x00001201
-> calling sub_00400d9c (index = 0), x0 = 0x00801000, arg_48 = 0x00000300
-> calling sub_00400dac (index = 1), x0 = 0x00801000, arg_48 = 0x0032e301
-> calling sub_00400d9c (index = 0), x0 = 0x00801000, arg_48 = 0x00000400
-> calling sub_00400dac (index = 1), x0 = 0x00801000, arg_48 = 0x00024401
-> calling sub_00401490 (index = 29), x0 = 0x00801000, arg_48 = 0x0000001d
-> calling sub_00400d9c (index = 0), x0 = 0x00801000, arg_48 = 0x00000100
-> calling sub_00400dac (index = 1), x0 = 0x00801000, arg_48 = 0x00001101
-> calling sub_00400c90 (index = 10), x0 = 0x00801000, arg_48 = 0x0000220a
-> calling sub_00400d9c (index = 0), x0 = 0x00801000, arg_48 = 0x00000300
-> calling sub_00400dac (index = 1), x0 = 0x00801000, arg_48 = 0x003fc301
-> calling sub_00400d9c (index = 0), x0 = 0x00801000, arg_48 = 0x00000400
-> calling sub_00400dac (index = 1), x0 = 0x00801000, arg_48 = 0x00010401
-> calling sub_00401490 (index = 29), x0 = 0x00801000, arg_48 = 0x0000001d
[...]
```

On peut alors exploiter la trace produite pour identifier la liste des fonctions appelées :

```
$ grep calling trace.txt|cut -d ' ' -f 3 | sort | uniq -c
   5281 sub_0040077c
   5378 sub_004008c4
    676 sub_00400918
  10562 sub_00400b04
  15851 sub_00400b78
  21140 sub_00400bd0
  15859 sub_00400c20
   6604 sub_00400c90
   5298 sub_00400ce0
    684 sub_00400d24
  17243 sub_00400d9c
  17243 sub_00400dac
    676 sub_004011b4
      3 sub_00401490
  21193 sub_00401580
    692 sub_004016e4
  10675 sub_00401794
$ grep calling trace.txt|cut -d ' ' -f 3 | sort | uniq -c | wc -l
17
```

L'analyse de ces 17 fonctions est alors nécessaire pour déterminer le jeu d'instructions de la machine virtuelle.

2.4.8 Détermination du jeu d'instructions

Les fonctions analysées dans cette partie correspondent à un exemple de chaque type de fonction pouvant être exécuté par la machine virtuelle. La méthodologie présentée dans la suite de ce document est applicable à l'ensemble des autres fonctions.

Analyse de la fonction `sub_400d9c`

Le code de la fonction `sub_400d9c` est présenté ci-dessous :

```
400d9c:    d34c6c22    ubfx    x2, x1, #12, #16 // x2 = x1[12..27]
400da0:    53103c42    lsl     w2, w2, #16      // w2 = w2 << 16
400da4:    d3482c21    ubfx    x1, x1, #8, #4   // x1 = x1[8..12]
400da8:    1400062e    b    0x402660            // sub_402660(x0, x1, w2)
```

La fonction va simplement extraire des séquences de bit depuis le second argument de la fonction et appelle ensuite la fonction `sub_402660`.

Analyse de la fonction `sub_402660`

Le code de la fonction `sub_402660` est présenté ci-dessous :

```
402660:    a9bd7bfd    stp    x29, x30, [sp,#-48]!
402664:    910003fd    mov    x29, sp
402668:    f9000bf3    str    x19, [sp,#16]
40266c:    7100403f    cmp    w1, #0x10              // compare w1 à 16
402670:    aa0003f3    mov    x19, x0                // x19 = arg0
402674:    2a0203e4    mov    w4, w2                 // w4 = arg2
402678:    540001ac    b.gt    0x4026ac              // saute si w1 >16
40267c:    35000081    cbnz    w1, 0x40268c          // saute si w1 != 0
402680:    f9400bf3    ldr    x19, [sp,#16]
402684:    a8c37bfd    ldp    x29, x30, [sp],#48
402688:    d65f03c0    ret

40268c:    51000421    sub    w1, w1, #0x1           // w1 -= 1
402690:    937e7c21    sbfiz    x1, x1, #2, #32      // x1 = x1 << 2
402694:    910083a2    add    x2, x29, #0x20         // x2 = x29 + 32
402698:    d2800083    mov    x3, #0x4               // x3 = 4
40269c:    b90023a4    str    w4, [x29,#32]          // *(x29 + 32) = w4 = arg2
4026a0:    97ffff31    bl    0x402364                // sub_402364(arg0, x1, x29 + 32, 4)
4026a4:    7100101f    cmp    w0, #0x4               // compare w0 à 4
4026a8:    54fffec0    b.eq    0x402680              // retourne w0 si égal à 4

4026ac:    39400260    ldrb    w0, [x19]             // w0 = *(arg0)
4026b0:    121f7800    and    w0, w0, #0xfffffffe    // w0 &= 0xfffffffe
4026b4:    39000260    strb    w0, [x19]             // *(arg0) = w0
4026b8:    52800080    mov    w0, #0x4               // w0 = 4
4026bc:    b9000660    str    w0, [x19,#4]           // *(arg0 + 4) = 4
4026c0:    f9400bf3    ldr    x19, [sp,#16]
4026c4:    a8c37bfd    ldp    x29, x30, [sp],#48
4026c8:    d65f03c0    ret
```

La fonction vérifie que le paramètre `arg1` est bien compris entre 0 et 16. Si c'est le cas, une adresse est calculée à partir de `4 * (arg - 1)`. La valeur passée dans le paramètre `arg2` est alors écrite à l'aide de la fonction `sub_402364` à l'adresse calculée précédemment.

On peut supposer que cette fonction sert à écrire une valeur dans un des registres de la machine virtuelle.

Analyse de la fonction `sub_400dac`

Le code de la fonction `sub_400dac` est présenté ci-dessous :

```
400dac:    a9bd7bfd     stp    x29, x30, [sp,#-48]!
400db0:    910003fd     mov    x29, sp
400db4:    2a0103e2     mov    w2, w1               // w2 = arg1
400db8:    a90153f3     stp    x19, x20, [sp,#16]
400dbc:    d3482c53     ubfx    x19, x2, #8, #4     // x19 = arg1[8..11]
400dc0:    2a1303e1     mov    w1, w19              // w1 = arg1[8..11]
400dc4:    910043ff     add    sp, sp, #0x10
400dc8:    8a3f039c     bic    x28, x28, xzr        // x28 = x28 and ~xzr = x28
400dcc:    910043ff     add    sp, sp, #0x10
400dd0:    f90003f5     str    x21, [sp]            // *(sp) = x21
400dd4:    d10043ff     sub    sp, sp, #0x10
400dd8:    d10043ff     sub    sp, sp, #0x10
400ddc:    d34c6c54     ubfx    x20, x2, #12, #16   // x20 = arg1[12..27]
400de0:    8a000015     and    x21, x0, x0          // x21 = arg0
400de4:    94000604     bl    0x4025f4              // w0 = sub_4025f4(arg1[8..1])
400de8:    2a140002     orr    w2, w0, w20          // w2 = w0 | arg1[12..27]
400dec:    2a1303e1     mov    w1, w19              // w1 = arg1[8..11]
400df0:    8a1502a0     and    x0, x21, x21         // x0 = x21 = arg0
400df4:    a94153f3     ldp    x19, x20, [sp,#16]
400df8:    f94013f5     ldr    x21, [sp,#32]
400dfc:    a8c37bfd     ldp    x29, x30, [sp],#48
400e00:    ca1f0108     eor    x8, x8, xzr           // x8 = 0
400e04:    14000617     b    0x402660                // sub_402660(arg0, arg1[8..11], w2)
```

Cette fonction extrait des champs de bits depuis `arg1` et appelle la fonction `sub_4025f4` avec le paramètre `arg1[8..11]`. L'analyse de cette fonction est donc nécessaire pour poursuivre.

Analyse de la fonction `sub_4025f4`

Le code de la fonction `sub_4025f4` est présenté ci-dessous :

```
4025f4:    a9bd7bfd     stp    x29, x30, [sp,#-48]!
4025f8:    910003fd     mov    x29, sp
4025fc:    f9000bf3     str    x19, [sp,#16]
402600:    7100403f     cmp    w1, #0x10            // compare arg1 à 16
402604:    aa0003f3     mov    x19, x0              // x19 = arg0
402608:    5400016d     b.le    0x402634            // saute si arg <= 16

40260c:    39400260     ldrb    w0, [x19]
402610:    12800002     mov    w2, #0xffffffff      // #-1
402614:    121f7800     and    w0, w0, #0xfffffffe
402618:    39000260     strb    w0, [x19]
40261c:    52800080     mov    w0, #0x4             // #4
402620:    b9000660     str    w0, [x19,#4]

402624:    2a0203e0     mov    w0, w2               // retourne w2
402628:    f9400bf3     ldr    x19, [sp,#16]
40262c:    a8c37bfd     ldp    x29, x30, [sp],#48
402630:    d65f03c0     ret
```

```
402634:    52800002    mov    w2, #0x0              // w2 = 0
402638:    34ffff61    cbz    w1, 0x402624          // retourne w2 si w1 == 0
40263c:    51000421    sub    w1, w1, #0x1          // w1--
402640:    937e7c21    sbfiz    x1, x1, #2, #32     // w1 = w1 << 2
402644:    910083a2    add    x2, x29, #0x20        // x2 = x29 + 32
402648:    d2800083    mov    x3, #0x4              // x3 = 4
40264c:    97ffff18    bl    0x4022ac               // w0 = sub_4022ac(arg0, w1, x2, 4)
402650:    7100101f    cmp    w0, #0x4              // compare w0 à 4
402654:    54fffdc1    b.ne    0x40260c             // sort en erreur
402658:    b94023a2    ldr    w2, [x29,#32]         // charge le résultat dans w2
40265c:    17fffff2    b    0x402624                // retourne w2
```

Cette fonction va lire le registre dont le numéro est spécifié dans l'argument `arg1` et va retourner la valeur lue.

On peut donc conclure sur l'utilité de la fonction `sub_400dac` : cette fonction va simplement lire la valeur d'un registre, effectuer un « ou logique » avec une valeur immédiate décodée depuis l'opérande (`arg1`) puis sauvegarder le résultat dans le même registre.

Analyse de la fonction `sub_401794`

Le code de la fonction `sub_401794` est présenté ci-dessous :

```
400548:    8a3f03bd    bic    x29, x29, xzr         // x29 = x29
40054c:    140004a9    b    0x4017f0
[...]
40154c:    ca140000    eor    x0, x0, x20
401550:    ca000294    eor    x20, x20, x0
401554:    ca000280    eor    x0, x20, x0
401558:    aa000014    orr    x20, x0, x0
40155c:    9400045c    bl    0x4026cc               // w0 = sub_4026cc(arg0) = eip
401560:    140000b7    b    0x40183c

401794:    a9be7bfd    stp    x29, x30, [sp,#-32]!
401798:    a90153f3    stp    x19, x20, [sp,#16]
40179c:    910003fd    mov    x29, sp
4017a0:    2a0103f3    mov    w19, w1               // w19 = arg1
4017a4:    d3493261    ubfx    x1, x19, #9, #4      // x1 = arg1[9..12]
4017a8:    8a000014    and    x20, x0, x0           // x20 = arg0
4017ac:    94000392    bl    0x4025f4               // w0 = sub_4025f4(arg0, arg1[9..12])
4017b0:    2a0003e1    mov    w1, w0                // w1 = w0
4017b4:    d34d3e60    ubfx    x0, x19, #13, #3     // x0 = arg1[13..15]
4017b8:    97fffb5c    bl    0x400528               // w0 = sub_400528(arg1[13..15], w1)
4017bc:    34ff6c60    cbz    w0, 0x400548          // retourne si w0 == 0
4017c0:    3747ec73    tbnz    w19, #8, 0x40154c    // teste le bit 8 de arg0, saute si != 0
4017c4:    aa3403e0    mvn    x0, x20               // x0 = ~arg0
4017c8:    aa2003e0    mvn    x0, x0                // x0 = ~x0 = arg0
4017cc:    53107e61    lsr    w1, w19, #16          // w1 = arg1 >> 16
4017d0:    a94153f3    ldp    x19, x20, [sp,#16]
4017d4:    a8c27bfd    ldp    x29, x30, [sp],#32
4017d8:    140003cb    b    0x402704                // sub_402704(arg0, arg1 >> 16)

4017f0:    a94153f3    ldp    x19, x20, [sp,#16]
4017f4:    a8c27bfd    ldp    x29, x30, [sp],#32
4017f8:    d61f03c0    br    x30
[...]
40183c:    ca000042    eor    x2, x2, x0
401840:    ca020000    eor    x0, x0, x2
401844:    ca020002    eor    x2, x0, x2            // x2 = x0 = eip
401848:    aa0203e0    mov    x0, x2
40184c:    528001e1    mov    w1, #0xf              // w1 = 15
401850:    ca0d01ad    eor    x13, x13, x13
401854:    ca0d01ad    eor    x13, x13, x13
```

```
  401858:    ca0d01ad     eor    x13, x13, x13
  40185c:    8a0d01ad     and    x13, x13, x13
  401860:    8a140280     and    x0, x20, x20           // x0 = arg0
  401864:    9400037f     bl    0x402660               // sub_402660(arg0, 15, eip)
  401868:    8a140280     and    x0, x20, x20
  40186c:    53107e61     lsr    w1, w19, #16
  401870:    a94153f3     ldp    x19, x20, [sp,#16]
  401874:    a8c27bfd     ldp    x29, x30, [sp],#32
  401878:    140003a3     b    0x402704                 // sub_402704(arg0, arg1 >> 16)
```

Le code de cette fonction est équivalent au code en C ci-dessous :

```
void sub_401794(char *addr, uint32_t arg) {
    uint32_t w0, w1;
    uint32_t eip;

    w1 = sub_4025f4(addr, (arg >> 9) & 0xf);
    w0 = (arg >> 13) & 7;

    w0 = sub_400528(w0, w1);
    if (w0 == 0) {
        return;
    }
    if ( ((arg >> 8) & 1) != 0) {
        eip = load_eip(addr);
        sub_402660(addr, 0xf, eip);
    }
    sub_402704(addr, (arg >> 16));
}
```

Les fonctions `sub_405f4` et `sub_402660` sont déjà connues (respectivement lecture et écriture d'un registre), il reste à étudier les fonctions `sub_400528` et `sub_402704`.

Analyse de la fonction `sub_400528`

Le code de la fonction `sub_400528` est présenté ci-dessous :

```
  400528:    2a0003e2     mov    w2, w0
  40052c:    52800020     mov    w0, #0x1                    // #1
  400530:    34000382     cbz    w2, 0x4005a0
  400534:    6b00005f     cmp    w2, w0
  400538:    54005d20     b.eq    0x4010dc
  40053c:    7100085f     cmp    w2, #0x2
  400540:    54000400     b.eq    0x4005c0
  400544:    14000019     b    0x4005a8
  [...]
  400550:    7100105f     cmp    w2, #0x4
  400554:    540003a0     b.eq    0x4005c8
  400558:    14000314     b    0x4011a8
  [...]
  400584:    7100185f     cmp    w2, #0x6
  400588:    54000260     b.eq    0x4005d4
  40058c:    1400044e     b    0x4016c4
  [...]
  4005a0:    d61f03c0     br    x30
  [...]
  4005a8:    71000c5f     cmp    w2, #0x3
  4005ac:    54000040     b.eq    0x4005b4
  4005b0:    17ffffe8     b    0x400550

  4005b4:    52800020     mov    w0, #0x1                    // #1
  4005b8:    34fffcc1     cbz    w1, 0x400550
```

```
  4005bc:    d61f03c0     br    x30

  4005c0:    35ffff41     cbnz    w1, 0x4005a8
  4005c4:    d61f03c0     br    x30

  4005c8:    52800020     mov    w0, #0x1                    // #1
  4005cc:    36f85ee1     tbz    w1, #31, 0x4011a8
  4005d0:    d61f03c0     br    x30
  4005d4:    6b1f003f     cmp    w1, wzr
  4005d8:    52800020     mov    w0, #0x1                    // #1
  4005dc:    5400874c     b.gt    0x4016c4
  4005e0:    17fffff0     b    0x4005a0

  4005e4:    6b1f003f     cmp    w1, wzr
  4005e8:    52800020     mov    w0, #0x1                    // #1
  4005ec:    54fffccd     b.le    0x400584
  4005f0:    d61f03c0     br    x30
  [...]
  4010dc:    5a8087e0     csneg    w0, wzr, w0, hi
  4010e0:    5a8097e0     csneg    w0, wzr, w0, ls
  4010e4:    d61f03c0     br    x30
  [...]
  4011a8:    7100145f     cmp    w2, #0x5
  4011ac:    54ffa1c0     b.eq    0x4005e4
  4011b0:    17fffcf5     b    0x400584
  [...]
  4016c4:    71001c5f     cmp    w2, #0x7
  4016c8:    1a9f17e2     cset    w2, eq
  4016cc:    2a2103e0     mvn    w0, w1
  4016d0:    0a407c40     and    w0, w2, w0, lsr #31
  4016d4:    17fffbb3     b    0x4005a0
```

Cette fonction est assez complexe et multiplie les sauts conditionnels. Le code C ci-dessous propose une implémentation équivalente :

```c
uint32_t sub_400528(uint32_t w0, int32_t w1) {
    uint32_t w2;
    w2 = w0;
    if (w2 == 0) {
        return 1;
    }
    if (w2 == 1) {
         return 1;
    }
    if (w2 == 2) {
        if (w1 == 0) {
            return 1;
    }
    }
    if (w2 == 3) {
        if (w1 != 0) {
        return 1;
    }
    }
    if (w2 == 4) {
        if ( ((w1 >> 31) & 1 ) != 0) {
            return 1;
    }
    }
    if (w2 == 5) {
        if (w1 > 0) {
            return 1;
    }
    }
    if (w2 == 6) {
```

```
        if (w1 <= 0) {
            return 1;
    }
    }
    if (w2 == 7) {
        w2 = 1;
    } else {
        w2 = 0;
    }
    w0 = ~w1;
    w0 = w2 & (w0 >> 31);
    return w0;
}
```

Cette fonction effectue des tests, déterminés par la valeur du paramètre `w0`, sur la valeur `w1` et retourne 1 si le test réussit, 0 sinon.

Analyse de la fonction `sub_402704`

Le code de la fonction `sub_402704` est présenté ci-dessous :

```
  402704:    a9be7bfd    stp    x29, x30, [sp,#-32]!
  402708:    910003fd    mov    x29, sp
  40270c:    529fffe2    mov    w2, #0xffff               // w2 = 65535
  402710:    b90013a1    str    w1, [x29,#16]             // *(x29 + 16) = w1
  402714:    6b02003f    cmp    w1, w2                    // compare w1 à 65535
  402718:    aa0003e1    mov    x1, x0                    // x1 = arg0
  40271c:    54000109    b.ls    0x40273c                 // saute si w1 < 65535
  402720:    39400000    ldrb    w0, [x0]
  402724:    121f7800    and    w0, w0, #0xfffffffe
  402728:    39000020    strb    w0, [x1]
  40272c:    52800020    mov    w0, #0x1                  // #1
  402730:    b9000420    str    w0, [x1,#4]
  402734:    a8c27bfd    ldp    x29, x30, [sp],#32
  402738:    d65f03c0    ret

  40273c:    910043a2    add    x2, x29, #0x10            // x2 = x29 + 16
  402740:    d2800781    mov    x1, #0x3c                 // x1 = 0x3c
  402744:    d2800083    mov    x3, #0x4                  // x3 = 4
  402748:    97ffff07    bl    0x402364                   // sub_402364(arg0, 0x3c, x29 + 16, 4)
  40274c:    a8c27bfd    ldp    x29, x30, [sp],#32
  402750:    d65f03c0    ret
```

Cette fonction va mettre à jour la valeur du registre `eip` avec la valeur du paramètre `arg1` si ce dernier est inférieur à 65535.

L'analyse de cette fonction permet de déterminer le rôle de la fonction parente, `sub_401794` : cette dernière implémente un saut conditionnel en fonction des données codées dans l'opérande.

Analyse de la fonction `sub_401490`

Le code de la fonction `sub_401490` est présenté ci-dessous :

```
  40100c:    ca130000    eor    x0, x0, x19
  401010:    ca000273    eor    x19, x19, x0
  401014:    ca000260    eor    x0, x19, x0
  401018:    8a000013    and    x19, x0, x0
  40101c:    aa1f0273    orr    x19, x19, xzr
  401020:    f9400bf3    ldr    x19, [sp,#16]
```

```
401024:    a8c27bfd     ldp    x29, x30, [sp],#32
401028:    17ffff78     b    0x400e08
[...]
401a28:    8a130260     and    x0, x19, x19
401a2c:    f9400bf3     ldr    x19, [sp,#16]
401a30:    a8c27bfd     ldp    x29, x30, [sp],#32
401a34:    17fffaf6     b    0x40060c
[...]
401490:    a9be7bfd     stp    x29, x30, [sp,#-32]!
401494:    910003fd     mov    x29, sp
401498:    910043ff     add    sp, sp, #0x10
40149c:    f90003f3     str    x19, [sp]
4014a0:    d10043ff     sub    sp, sp, #0x10
4014a4:    ca000273     eor    x19, x19, x0
4014a8:    52800021     mov    w1, #0x1                        // #1
4014ac:    ca130000     eor    x0, x0, x19
4014b0:    aa1f0318     orr    x24, x24, xzr
4014b4:    ca130013     eor    x19, x0, x19
4014b8:    aa130260     orr    x0, x19, x19
4014bc:    9400044e     bl    0x4025f4
4014c0:    34002b40     cbz    w0, 0x401a28
4014c4:    7100041f     cmp    w0, #0x1
4014c8:    54ffda20     b.eq    0x40100c
4014cc:    aa120252     orr    x18, x18, x18
4014d0:    7100081f     cmp    w0, #0x2
4014d4:    54000480     b.eq    0x401564
4014d8:    71000c1f     cmp    w0, #0x3
4014dc:    ca130000     eor    x0, x0, x19
4014e0:    ca000273     eor    x19, x19, x0
4014e4:    ca1f0318     eor    x24, x24, xzr
4014e8:    ca000260     eor    x0, x19, x0
4014ec:    aa0003f3     mov    x19, x0
4014f0:    54000260     b.eq    0x40153c
4014f4:    f9400bf3     ldr    x19, [sp,#16]
4014f8:    a8c27bfd     ldp    x29, x30, [sp],#32
4014fc:    528000a1     mov    w1, #0x5                        // #5
401500:    140003cf     b    0x40243c
[...]
40153c:    f9400bf3     ldr    x19, [sp,#16]
401540:    a8c27bfd     ldp    x29, x30, [sp],#32
401544:    17fffc71     b    0x400708
[...]
401564:    ca130000     eor    x0, x0, x19
401568:    ca000273     eor    x19, x19, x0
40156c:    ca000260     eor    x0, x19, x0
401570:    8a000013     and    x19, x0, x0
401574:    f9400bf3     ldr    x19, [sp,#16]
401578:    a8c27bfd     ldp    x29, x30, [sp],#32
40157c:    17ffff3d     b    0x401270
```

Cette fonction est équivalente au code C ci-dessous :

```c
void vm_sub_401490(char *addr, uint32_t arg) {
    uint32_t w0;

    w0 = sub_4025f4(addr, 1);

    if (w0 == 0) {
       sub_40060c(addr);
       return;
    }

    if (w0 == 1) {
        sub_400e08(addr);
        return;
```

```
    }

    if (w0 == 2) {
        sub_401270(addr);
        return;
    }

    if (w0 == 3) {
        sub_400708(addr);
        return;
    }
    sub_40243c(addr);
}
```

Cette fonction appelle une autre sous-fonction selon la valeur du registre 1. L'analyse de ces sous-fonctions permet de déterminer le comportement suivant :

- fonction `sub_40060c` : appel système `open` en utilisant les valeurs des registres `r2`, `r3` et `r4` ;
- fonction `sub_400e08` : lis des données depuis un numéro de descripteur de fichier stocké dans le registre `r2`, alloue une zone mémoire avec mmap pour lire le nombre d'octets spécifiés par le registre `r4` puis écrit les données lues à l'adresse spécifiée par le registre `r3` ;
- fonction `sub_401270` : lis des données à une adresse spécifiée par le registre `r3` (le registre `r4` précisant le nombre d'octets à lire) puis écrit ces données sur le descripteur de fichiers stocké dans le registre `r2` ;
- fonction `sub_400708` : appel système `close` sur le descripteur de fichiers stocké dans le registre `r2` ;
- fonction `sub_40243c` : termine l'exécution du programme.

La fonction `sub_401490` permet donc de gérer les appels systèmes de la machine virtuelle.

Analyse de la fonction `sub_40077c`

Le code de la fonction `sub_40077c` est présenté ci-dessous :

```
  40077c:    a9be7bfd     stp    x29, x30, [sp,#-32]!
  400780:    910003fd     mov    x29, sp
  400784:    a90153f3     stp    x19, x20, [sp,#16]
  400788:    2a0103f3     mov    w19, w1
  40078c:    ca000294     eor    x20, x20, x0
  400790:    d34c3e61     ubfx    x1, x19, #12, #4
  400794:    ca140000     eor    x0, x0, x20
  400798:    9b1f6659     madd    x25, x18, xzr, x25
  40079c:    ca140014     eor    x20, x0, x20
  4007a0:    ca140000     eor    x0, x0, x20
  4007a4:    ca000294     eor    x20, x20, x0
  4007a8:    ca000280     eor    x0, x20, x0
  4007ac:    aa0003f4     mov    x20, x0
  4007b0:    94000791     bl    0x4025f4
  4007b4:    4a400400     eor    w0, w0, w0, lsr #1
  4007b8:    4a400801     eor    w1, w0, w0, lsr #2
  4007bc:    1200e022     and    w2, w1, #0x11111111
  4007c0:    3200e3e0     mov    w0, #0x11111111                // #286331153
  4007c4:    1b007c42     mul    w2, w2, w0
  4007c8:    d3482e61     ubfx    x1, x19, #8, #4
  4007cc:    aa140280     orr    x0, x20, x20
  4007d0:    9a9ee042     csel    x2, x2, x30, al
  4007d4:    a94153f3     ldp    x19, x20, [sp,#16]
  4007d8:    d503201f     nop
  4007dc:    a8c27bfd     ldp    x29, x30, [sp],#32
  4007e0:    d35c7042     ubfx    x2, x2, #28, #1
  4007e4:    91000084     add    x4, x4, #0x0
  4007e8:    1400079e     b    0x402660
```

Le code C ci-dessous est équivalent à la fonction `sub_40077c` :

```
void sub_40077c(char *addr, uint32_t arg) {
    uint32_t rd, rn, w0, w1, w2;

    rn = (arg >> 12) & 0xf;
    w0 = sub_4025f4(addr, rn);
    w0 = w0 ^ (w0 >> 1);
    w1 = w0 ^ (w0 >> 2);
    w2 = w1 & 0x11111111;
    w2 = w2 * 0x11111111;

    rd = (arg >> 8) & 0xf;
    w2 = (w2 >> 28) & 1;

    sub_402660(addr, rd, w2);
}
```

Cette fonction calcule en fait la parité de la valeur stockée dans le registre spécifié par l'opérande, c'est-à-dire si la valeur contient un nombre impair de bits à 1 ou non. Des détails sur cette technique peuvent être obtenus à l'adresse http://bits.stephan-brumme.com/parity.html.

Résultat

En appliquant la même méthode que pour les fonctions précédemment analysées, il est possible de déduire l'ensemble du jeu d'instructions. Ces instructions peuvent être classées en quatre catégories :

— les opérations entre registres ;
— les opérations de chargement de valeurs dans les registres ;
— l'instruction du saut conditionnel ;
— deux instructions spéciales.

Les instructions sur les registres sont codées de la façon suivante :

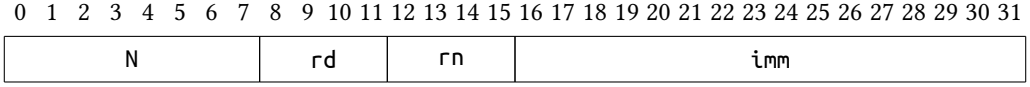

FIGURE 2.4 – Format d'une instruction sur les registres

Les instructions codées sur ce format sont présentées au tableau 2.1. Parmi ces instructions, seules celles prenant une valeur immédiate de 16 bits sont codées sur 32 bits, les autres instructions étant codées sur 16 bits.

Les instructions de chargement de valeurs dans les registres sont codées de la façon ci-dessous :

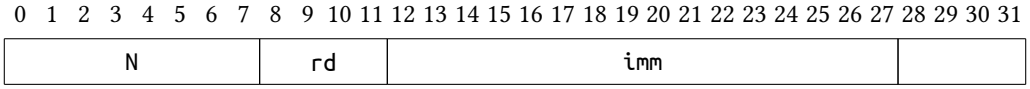

FIGURE 2.5 – Format des chargements de valeurs dans les registres

Les instructions codées sur ce format sont présentées au tableau 2.2.

L'instruction qui correspond à un saut conditionnel est codée de la façon ci-dessous :

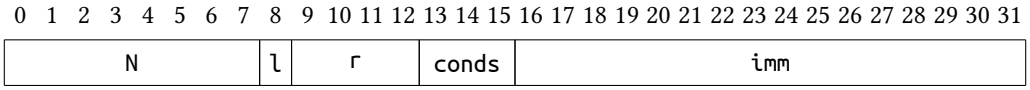

FIGURE 2.6 – Codage du saut conditionnel

La sémantique de cette instruction est présentée sur le tableau 2.3.

Opcode	Fonction	Description
2	`sub_401580`	`rd = load_word [rn + imm]`
4	`sub_4016e4`	`rd = load_byte [rn + imm]`
7	`sub_4011b4`	`store_byte [rn + imm], rd`
10	`sub_400c90`	`rd = rd ^ rn`
11	`sub_400c20`	`rd = rd | rn`
12	`sub_400bd0`	`rd = rd & rn`
13	`sub_400b78`	`rd = rd << rn`
14	`sub_400b04`	`rd = rd >> rn`
18	`sub_400918`	`rd = rd + rn`
19	`sub_4008c4`	`rd = rd - rn`
22	`sub_400d24`	`rd = rd + 1`
23	`sub_400ce0`	`rd = rd - 1`
30	`sub_40077c`	`rd = parity rn`

Table 2.1 – Opérations sur les registres de la machine virtuelle

Opcode	Fonction	Description
0	`sub_400d9c`	`rd = (imm) << 16`
1	`sub_400dac`	`rd = rd | imm`

Table 2.2 – Chargement de valeurs dans les registres de la machine virtuelle

Opcode	Fonction	Description
8	`sub_401794`	`jmp imm if r test conds` (sauvegarde d'eip si l = 1)

Table 2.3 – Saut conditionnel de la machine virtuelle

Enfin, les deux dernières instructions n'utilisent pas d'opérandes et sont présentées au tableau 2.4.

Opcode	Fonction	Description
28	`sub_4005fc`	`halt`
29	`sub_401490`	`syscall`

Table 2.4 – Opérations spéciales de la machine virtuelle

2.5 Désassemblage du programme de la machine virtuelle

L'espace mémoire initial de la machine virtuelle peut être obtenu en déchiffrant, à l'aide de l'algorithme `chacha`, la zone de données copiée à l'adresse `addr1`. L'utilisation de GDB permet d'abord d'obtenir les données chiffrées, en mettant des points d'arrêt au sein de la fonction `prepare_mem`.

```
$ gdb-multiarch -q badbios2.bin
Reading symbols from badbios2.bin...(no debugging symbols found)...done.
(gdb) target remote 127.1:1234
Remote debugging using 127.1:1234
0x0000000000400514 in ?? ()
(gdb) break *0x4029f0         // retour du second mmap
Breakpoint 1 at 0x4029f0
(gdb) break *0x402b00         // fin de la copie des données
Breakpoint 2 at 0x402b00
(gdb) cont
Continuing.
```

```
Breakpoint 1, 0x0000000004029f0 in ?? ()
(gdb) p $x0
$1 = 0x4000802000
(gdb) cont
Continuing.

Breakpoint 2, 0x0000000000402b00 in ?? ()
(gdb) dump binary memory cipher.bin 0x4000802000 (0x4000802000+0x10000)
```

Il reste maintenant à déchiffrer ces données en utilisant l'algorithme chacha. Pour cela, le programme en C ci-dessous est utilisé :

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#include "ecrypt-sync.h"

void init_ctx(ECRYPT_ctx *ctx) {
    int i;

    ctx->input[0] = 0x61707865;
    ctx->input[1] = 0x3120646e;
    ctx->input[2] = 0x79622d36;
    ctx->input[3] = 0x6b206574;

    for (i = 4; i < 12; i++) {
        ctx->input[i] = 0x05b1ad0b;
    }
    for (i = 12; i < 16; i++) {
        ctx->input[i] = 0;
    }

}

int main(int argc, char **argv) {
    char *input, *output;
    unsigned char *input_data;
    struct stat in_st;
    int input_fd, output_fd;
    int i;
    unsigned char tmp[64];
    ECRYPT_ctx ctx;

    if (argc != 3) {
        fprintf(stderr, "usage: %s input output\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    input = argv[1]; output = argv[2];

    init_ctx(&ctx);

    if (stat(input, &in_st) == -1) {
        perror("stat");
        exit(EXIT_FAILURE);
    }

    input_fd = open(input, O_RDONLY);
    if (input_fd == -1) {
        perror("open");
```

```
        exit(EXIT_FAILURE);
    }

    output_fd = creat(output, S_IRWXU);
    if (output_fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    input_data = mmap(NULL, in_st.st_size, PROT_READ, MAP_SHARED, input_fd, 0);
    if (input_data == (void *) -1) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }

    ECRYPT_init();
    for (i = 0; i < (in_st.st_size / 64); i++) {
        ECRYPT_decrypt_bytes(&ctx, input_data + 64 * i, tmp, 64);
        write(output_fd, tmp, 64);
    }

    close(output_fd);
    munmap(input_data, in_st.st_size);
    close(input_fd);

    exit(EXIT_SUCCESS);
}
```

Il ne reste plus qu'à l'exécuter sur le fichier `cipher.bin` obtenu précédemment avec GDB :

```
$ gcc -o decrypt-data decrypt-data.c chacha.c
$ ./decrypt-data cipher.bin cleartext.bin
```

Le fichier obtenu contient bien des données en clair :

```
$ hexdump -C cleartext.bin
hexdump -C cleartext.bin
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000030  00 00 00 00 00 20 00 00  00 00 00 00 40 00 00 00  |..... ......@...|
00000040  00 01 00 00 01 21 00 00  00 02 00 00 01 12 00 00  |.....!..........|
00000050  00 03 00 00 01 e3 32 00  00 04 00 00 01 44 02 00  |......2......D..|
00000060  1d 00 00 01 00 00 01 11  00 00 0a 22 00 03 00 00  |..........."....|
00000070  01 c3 3f 00 00 04 00 00  01 04 01 00 1d 00 02 05  |..?.............|
[...]
00000320  1d 00 08 00 b2 02 00 00  00 00 00 00 00 00 3a 3a  |..............::|
00000330  20 50 6c 65 61 73 65 20  65 6e 74 65 72 20 74 68  | Please enter th|
00000340  65 20 64 65 63 72 79 70  74 69 6f 6e 20 6b 65 79  |e decryption key|
00000350  3a 20 00 00 3a 3a 20 54  72 79 69 6e 67 20 74 6f  |: ..:: Trying to|
00000360  20 64 65 63 72 79 70 74  20 70 61 79 6c 6f 61 64  | decrypt payload|
00000370  2e 2e 2e 0a 20 20 20 57  72 6f 6e 67 20 6b 65 79  |....   Wrong key|
00000380  20 66 6f 72 6d 61 74 2e  0a 00 20 20 20 49 6e 76  | format...   Inv|
00000390  61 6c 69 64 20 70 61 64  64 69 6e 67 2e 0a 00 00  |alid padding....|
000003a0  20 20 20 43 61 6e 6e 6f  74 20 6f 70 65 6e 20 66  |   Cannot open f|
000003b0  69 6c 65 20 70 61 79 6c  6f 61 64 2e 62 69 6e 2e  |ile payload.bin.|
000003c0  0a 00 3a 3a 20 44 65 63  72 79 70 74 65 64 20 70  |..:: Decrypted p|
000003d0  61 79 6c 6f 61 64 20 77  72 69 74 74 65 6e 20 74  |ayload written t|
000003e0  6f 20 70 61 79 6c 6f 61  64 2e 62 69 6e 2e 0a 00  |o payload.bin...|
000003f0  70 61 79 6c 6f 61 64 2e  62 69 6e 00 58 58 58 58  |payload.bin.XXXX|
00000400  58 58 58 58 58 58 58 58  58 58 58 58 00 00 00 00  |XXXXXXXXXXXX....|
00000410  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00008000  00 bc 68 15 b5 6b 1b 41  a2 19 c4 57 e0 01 f6 af  |..h..k.A...W....|
00008010  4b 35 98 b9 38 94 3a 6f  8c 86 6a d7 2a 23 4f 6f  |K5..8.:o..j.*#Oo|
00008020  ee a5 93 20 4c 55 f0 aa  e5 f3 59 38 da 18 39 bf  |... LU....Y8..9.|
```

```
[...]
00009fe0  4e 80 b6 bf 5f 7f 6a d5  2e db ae f4 f4 cc 35 dd  |N..._.j.......5.|
00009ff0  84 10 d1 76 6a 31 e5 d3  6a b6 54 c3 ca 8f 53 02  |...vj1..j.T...S.|
0000a000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00010000
```

On peut remarquer plusieurs sections dans ce fichier :

— une section de données, du début jusqu'au décalage `0x32d` ;
— des chaînes de caractères, de `0x32e` jusqu'au décalage `0x40b` ;
— enfin une importante section de données, de `0x8000` jusqu'à `0x9fff`, soit 8192 octets.

A partir du jeu d'instructions déterminé précédemment, il est possible de coder un désassembleur. Le script `disassvm.rb`, disponible à l'annexe A.2, retourne le résultat suivant par rapport aux données du fichier `cleartext.bin` :

```
$ ruby disassvm.rb cleartext.bin
[64]  R1 = 0
[68]  R1 |= 2
[72]  R2 = 0
[76]  R2 |= 1
[80]  R3 = 0
[84]  R3 |= 814
[88]  R4 = 0
[92]  R4 |= 36
[96]  syscall
[98]  R1 = 0
[...]
[726]  JMP 690 if R0 ALWAYS (0)
[730]  R1 = 0
[734]  R1 |= 2
[738]  R2 = 0
[742]  R2 |= 2
[746]  R3 = 0
[750]  R3 |= 906
[754]  R4 = 0
[758]  R4 |= 20
[762]  syscall
[764]  JMP 690 if R0 ALWAYS (0)
[768]  R1 = 0
[772]  R1 |= 2
[776]  R2 = 0
[780]  R2 |= 2
[784]  R3 = 0
[788]  R3 |= 928
[792]  R4 = 0
[796]  R4 |= 33
[800]  syscall
[802]  JMP 690 if R0 ALWAYS (0)
[806]  R0 = 0
```

Le résultat du désassemblage est également disponible à l'annexe A.2.

Le programme de la machine virtuelle commence par les instructions ci-dessous :

```
[68]  R1 = 2
[76]  R2 = 1
[84]  R3 = 814
[92]  R4 = 36
[96]  syscall                  # affiche ":: Please enter the decryption key: " sur stdout
[102]  R1 = 1
[106]  R2 ^= R2
```

```
[112]  R3 = 1020
[120]  R4 = 16
[124]  syscall                  # lis 16 octets sur stdin et les stocke à l'adresse 1020 (0x3fc)
[126]  R5 = *W[0x0 + R0]        # R5 vaut 16
[134]  R3 = 16
[138]  R5 -= R3                 # R5 = 0
[140]  JMP 692 if R5 != 0 (0)   # saut pas pris
[148]  R15 = 16                # R5 = 16
[156]  R14 = 1020              # R14 = 1020 (0x3fc)
[164]  R13 = 806               # R13 = 806 (0x326)
[168]  R13--                    # R13 = 805 (0x325)
[174]  R2 = 48                 # R2 = 48 ( 0 en ascii)
[182]  R3 = 57                 # R3 = 57 (9 en ascii)
[190]  R4 = 65                 # R4 = 65 (A en ascii)
[198]  R5 = 70                 # R5 = 70 (F en ascii)

[202]  R12 = *B[0x0 + R14]      # R12 = premier octet de clé = K[0]
[206]  R1 = *W[0x2c + R0]       # R1 = K[0]
[210]  R1 -= R2                 # R1 = K[0] - 48
[212]  JMP 692 if R1 < 0 (0)    # saute si K[0] < 48
[216]  R1 = *W[0x2c + R0]       # R1 = K[0]
[220]  R1 -= R3                 # R1 = K[0] - 57
[222]  JMP 262 if R1 <= 0 (0)   # saute si K[0] <= 57 (K[0] est un chiffre)
[226]  R1 = *W[0x2c + R0]       # R1 = K[0]
[230]  R1 -= R4                 # R12 = K[0] - 65
[232]  JMP 692 if R1 < 0 (0)    # saute si K[0] < 65
[236]  R1 = *W[0x2c + R0]       # R1 = K[0]
[240]  R1 -= R5                 # R1 = K[0] - 70
[242]  JMP 692 if R1 > 0 (0)    # saute si K[0] <= 70
[246]  R12 -= R4                # K[0] -= 65
[252]  R1 = 10                 # R1 = 10
[256]  R12 += R1                # K[0] += 10 (conversion hexa -> dec)
[258]  JMP 264 if R0 ALWAYS (0)

loc_262:
[262]  R12 -= R2                # K[0] -= 48 (conversion hexa -> dec)
```

Ces instructions vont simplement lire les 16 caractères de la clé puis convertir chaque caractère en l'équivalent décimal.

Le programme continue avec les instructions suivantes :

```
loc_264:
[268]  R7 = 16                  # R7 = 16
[272]  R7 -= R15                # R7 -= R15 (index de clé)
[278]  R1 = 1                   # R1 = 1
[282]  R1 &= R7                 # R1 = (16 - R15) & 1
[284]  JMP 300 if R1 != 0 (0)   # Saute à 300 si index de clé en cours est impair
[292]  R7 = 4                   #
[296]  R12 <<= R7               # R12 = R12 * 16 # décale de 16 un octet sur 2
[298]  R13++                    # R13++ = 0x326

loc_300:
[300]  R1 = *B[0x0 + R13]
[304]  R1 |= R12
[306]  *B[0x0 + R13] = R1
[310]  R14++
[312]  R15--
[314]  JMP 202 if R15 != 0 (0)  # répète pour tous les octets de clé
```

Ces instructions traitent les caractères de la clé deux par deux pour reconstituer chaque octet.

Le programme continue ensuite avec les instructions ci-dessous :

```
[322]  R1 = 2
[330]  R2 = 1
[338]  R3 = 852             # 0x354
[346]  R4 = 32
[350]  syscall              # affiche "Trying to decrypt payload"
[356]  R1 = 806             # R1 = 0x326 (pointe vers la clé convertie)
[360]  R10 = *W[0x0 + R1]   # R10 = K2[0..3]
[364]  R11 = *W[0x4 + R1]   # R11 = K2[4..7]
[368]  R1 ^= R1             # R1 = 0
[374]  R2 = 32768           # R2 = 0x8000
[382]  R3 = 8               # R3 = 8
[386]  R4 ^= R4             # R4 = 0
[388]  R12 = 2952790016     # R12 = 0xb0000000
[400]  R13 = 1              # R13 = 1
```

La clé convertie est chargée dans les registres `R10` et `R11` puis l'exécution continue :

```
loc_404:
[404]  R8 = *W[0x24 + R0]     # R8 = R10 = KO
[408]  R9 = *W[0x28 + R0]     # R8 = R11 = K1
[412]  R8 &= R12              # R8 = K0 & 0xb0000000
[414]  R9 &= R13              # R9 = K1 & 1
[416]  R8 ^= R9               # R8 = (K0 & 0xb0000000) ^ (K1 & 1)
[418]  R9 = PARITY R8         # R9 = 0 si nb de bits à 1 dans R8 pair
[424]  R8 = 1                 # R8 = 1
[432]  R7 = 31                # R7 = 31
[436]  R6 = *W[0x24 + R0]     # R6 = KO
[440]  R6 &= R8               # R6 = K0 & 1
[442]  R6 <<= R7              # R6 = (K0 & 1) << 31
[444]  R11 >>= R8             # R11 = K1 >> 1
[446]  R11 |= R6              # R11 = ( K1 >> 1 ) | ( (K0 & 1) << 31 )
[448]  R10 >>= R8             # R10 = K0 >> 1
[450]  R9 <<= R7              # R9 = R9 << 31
[452]  R10 |= R9              # R10 = (K0 >> 1) | ( (K1 & 1) << 31 )
[454]  R3--                   # R3--
[456]  R7 = *W[0x28 + R0]     # R7 = R11 = ( K1 >> 1 ) | ( (K0 & 1) << 31 )
[460]  R7 &= R8               # R7 = ( ( K1 >> 1 ) | ( (K0 & 1) << 31 ) ) & 1
[462]  R7 <<= R3              # R7 <<= R3
[464]  R4 |= R7               # R4 = R7
[466]  JMP 502 if R3 != 0 (0) #
[474]  R7 = 32768             # R7 = 32768
[478]  R7 += R1               # R7 = 32768 + R1
[480]  R8 = *B[0x0 + R7]      # R8 = *B[32768 + R1]
[484]  R8 ^= R4               # R8 = (*B[32768 + R1]) ^ R4
[486]  *B[0x0 + R7] = R8      # *B[32768 + R1] ^= R4
[494]  R3 = 8                 # R3 = 8
[498]  R1++                   # R1++
[500]  R4 ^= R4               # R4 = 0

loc_502:
[502]  R8 = 0
[506]  R8 |= 8192              # R8 = 8192
[510]  R8 -= R1                # R8 = 8192 - R1
[512]  JMP 404 if R8 > 0 (0)   # saute à 404 si (8192 - R1) > 0
```

Le programme effectue deux boucles imbriquées : la boucle extérieur effectue 8192 itérations, la seconde boucle effectue 8 itérations. A la sortie de la seconde boucle, un « ou-exclusif » est appliqué sur un octet en mémoire.

Le programme termine par les instructions ci-dessous :

```
[520]  R13 = 32768           # R13 = 32768
[528]  R12 = 8192            # R12 = 8192
[536]  R11 = 128             # R11 = 128
[540]  R10 ^= R10             # R10 = 0
```

```
[546]  R9 = 8                # R9 = 8

loc_550:
[550]  R10++
[552]  R12--
[554]  JMP "invalid padding" if R12 <= 0 (0)
[558]  R10 = *W[0x30 + R0]
[562]  R10 += R12
[564]  R1 = *B[0x0 + R10]
[568]  JMP 550 if R1 == 0 (0)
[572]  R1 -= R11
[574]  JMP "invalid padding" if R1 != 0 (0)
[578]  R10 -= R9
[580]  JMP "invalid padding" if R10 <= 0 (0)
[584]  R1 &= R0        # R1 = 0
[590]  R2 = 1008      # pointe vers payload.bin
[598]  R3 = 577
[606]  R4 = 438
[610]  syscall         # open
[612]  JMP exit if R1 < 0 (0)
[616]  R2 = *W[0x0 + R0]  # retour du syscall, fd
[624]  R1 = 2
[632]  R3 = 32768
[636]  R4 = *W[0x2c + R0] # longueur
[640]  syscall            # write
[646]  R1 = 3
[650]  syscall            # close
```

Le programme teste certaines conditions sur les données en mémoire et écrit le résultat dans le fichier `payload.bin` si les conditions sont remplies.

2.6 Inversion du LFSR et obtention de la clé

Le résultat de la rétroconception du programme de la machine virtuelle est présenté ci-dessous :

```c
uint32_t parity(uint32_t x) {
        x = x ^ (x >> 1);
        x = (x ^ (x >> 2)) & 0x11111111;
        x = x * 0x11111111;
        return (x >> 28) & 1;
}

int decrypt(uint32_t k0, uint32_t k1) {
        uint32_t r1, r4, r9, r10, r11, r12, r13;
        uint8_t b0;
        int i, j;

        r10 = k0; r11 = k1;
        for (i = 0; i < 8192; i++) {
                r4 = 0;
                for (j = 0; j < 8; j++) {
                        r9 = parity( (r10 & 0xb0000000) ^ (r11 & 1) );
                        r11 = (r11 >> 1) | ( (r10 & 1) << 31 );
                        r10 = (r10 >> 1) | (r9 << 31);

                        r4 |= (r11 & 1 ) << (7 - j) ;
                }

                b0 = mem[32768 + i];
                mem[32768 + i] ^= r4;
        }
        r13 = 32768;
```

```
        r12 = 8192;
        r11 = 128;
        r10 = 0;
        r9 = 8;
loc_550:
        r10++;
        r12--;
        if ( (int32_t) r12 <= 0) {
                printf("[0] Invalid padding\n");
                exit(EXIT_FAILURE);
        }
        r10 = r13 + r12;
        r1 = mem[r10] & 0xff;
        if (r1 == 0)
                goto loc_550;
        r1 = r1 - 128;
        if (r1 != 0) {
                printf("[1] Invalid padding\n");
                exit(EXIT_FAILURE);
        }
        r10 -= 8;
        if ( (int32_t) r10 <= 0) {
                printf("[2] Invalid padding\n");
                exit(EXIT_FAILURE);
        }
        printf("valid key !\n");
}
```

La fonction `decrypt` prend en argument deux entiers de 32 bits qui correspondent respectivement au début et à la fin de la clé passée en argument au programme `badbios2.bin`. Ces deux entiers sont ensuite utilisés pour initialiser l'état d'un registre à décalage à rétroaction linéaire sur 64 bits (LFSR) composé des registres `r10` et `r11`. Chaque octet en sortie du LFSR est utilisé pour calculer le résultat d'un « ou-exclusif » sur les octets de données, de la position `32768` à `32768 + 8192`.

Des tests sont ensuite effectués sur le contenu déchiffré :

— le résultat obtenu doit se terminer par un certain nombre d'octets à 0, suivi d'un octet à 0x80 ;
— le nombre d'octets à 0 doit être strictement supérieur à 8.

Ces conditions permettent de déterminer l'état attendu du LFSR à la fin du fichier : les 8 derniers octets des données déchiffrées doivent égaux à 0.

La fin des données chiffrées est présentée ci-dessous :

```
$ hexdump -C cleartext.bin|tail
00009f90  7a 15 8c de d6 05 d5 25  0d ac 9b e9 ae 3e b0 3e  |z......%.....>.>|
00009fa0  30 d2 ba 0f 2e 16 1c 5f  db e0 ca a1 f5 4d a3 e8  |0......_.....M..|
00009fb0  61 6d 29 fe 88 9a 73 2a  e2 08 d4 2e d5 a7 4b 58  |am)...s*......KX|
00009fc0  4a a4 1b 5f 77 c2 34 f1  bb b1 d7 7a fa 3b 4e 8f  |J.._w.4....z.;N.|
00009fd0  34 54 19 7c ca 82 32 a1  e4 f8 f0 5c ee d8 c4 48  |4T.|..2....\...H|
00009fe0  4e 80 b6 bf 5f 7f 6a d5  2e db ae f4 f4 cc 35 dd  |N..._.j.......5.|
00009ff0  84 10 d1 76 6a 31 e5 d3  6a b6 54 c3 ca 8f 53 02  |...vj1..j.T...S.|
0000a000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00010000
```

On en déduit la valeur de `r4` pour les 8 derniers octets :

— `i = 8184` : `r4 = 0x6a`
— `i = 8185` : `r4 = 0xb6`
— `i = 8186` : `r4 = 0x54`
— `i = 8187` : `r4 = 0xc3`

— `i = 8188 : r4 = 0xca`
— `i = 8189 : r4 = 0x8f`
— `i = 8190 : r4 = 0x53`
— `i = 8191 : r4 = 0x02`

En observant le fonctionnement du LFSR, on s'aperçoit que `r4` est construit bit par bit, en partant du bit de poids fort jusqu'au bit de poids faible. A chaque itération, le bit utilisé pour construire `r4` correspond au bit de poids faible de `r11`. Il faut donc inverser les octets attendus pour `r4` pour obtenir ceux correspondant aux registres `r10` et `r11`. Cette opération peut être effectuée avec le code Ruby ci-dessous :

```
2.1.1 :001 > a = [0x6a, 0xb6, 0x54, 0xc3, 0xca, 0x8f, 0x53, 0x2]
2.1.1 :002 > a.map {|b| "%x" % ("%8.8b" % b).reverse.to_i(2) }
 => ["56", "6d", "2a", "c3", "53", "f1", "ca", "40"]
```

On en déduit les valeurs de `r10` et `r11` à l'itération 8184 :

— `r10 = 0x40caf153` ;
— `r11 = 0xc32a6d56`.

Il reste maintenant à inverser les 8184 itérations pour obtenir la clé voulue.

Une étape du LFSR peut être inversée avec la fonction C ci-dessous :

```
void reverse_lfsr(uint32_t r10_1, uint32_t r11_1, uint32_t *r10_0, uint32_t *r11_0) {
    uint32_t r9;
    *r10_0 = (r10_1 << 1) | (r11_1 >> 31);
    r9 = (r10_1 >> 31);
    *r11_0 = (r11_1 << 1);

    if (parity( (*r10_0 & 0xb0000000) ^ (1)) == r9) {
        *r11_0 |= 1;
    }
}
```

Cette fonction retourne dans les variables `r10_0` et `r11_0` l'état précédent du LFSR.

Pour finir, il suffit d'appeler cette fonction `8184 * 8` fois pour découvrir la clé cherchée :

```
void crackme(uint32_t r10, uint32_t r11) {
    int i;
    uint32_t tmp0, tmp1;

    for (i = (8184 * 8) ; i >= 0; i--) {
    reverse_lfsr(r10, r11, &tmp0, &tmp1);
    r10 = tmp0;
    r11 = tmp1;
    printf("[%u] r10 = %x, r11 = %x\n", i, r10, r11);
    }
}
```

L'appel de cette fonction avec les paramètres `r10 = 0x40caf153` et `r11 = 0xc32a6d56` retourne :

```
$ ./crackme
[65472] r10 = 8195e2a7, r11 = 8654daad
[65471] r10 = 32bc54f, r11 = ca9b55b
[...]
[2] r10 = c16c6b42, r11 = c46b7785
[1] r10 = 82d8d685, r11 = 88d6ef0a
[0] r10 = 5b1ad0b, r11 = 11adde15
```

La clé correspond alors aux valeurs de `r10` et `r11`, c'est-à-dire `0BADB10515DEAD11`.

On peut alors tenter un déchiffrement avec QEMU :

```
$ qemu-aarch64 badbios2.bin
:: Please enter the decryption key: 0BADB10515DEAD11
:: Trying to decrypt payload...
:: Decrypted payload written to payload.bin.
$ file payload.bin
file payload.bin
payload.bin: Zip archive data, at least v2.0 to extract
```

L'analyse du fichier obtenu fait l'objet du chapitre suivant.

Chapitre 3

Analyse du microcontrôleur

3.1 Découverte

Le fichier obtenu à l'étape précédente se révèle être une archive au format zip qui contient deux autres fichiers :

```
$ md5sum payload.bin
eaf6caaaf9089ad689c50cc72d03efbd  payload.bin
$ file payload.bin
payload.bin: Zip archive data, at least v2.0 to extract
$ unzip payload.bin
unzip payload.bin
Archive:  payload.bin
  inflating: mcu/upload.py
  inflating: mcu/fw.hex
$ ls -al mcu
total 16
drwxrwxr-x 2 babar babar 4096 mai   23 14:23 .
drwxr-xr-x 3 babar babar 4096 mai   23 14:23 ..
-rw-r--r-- 1 babar babar 1323 avril 17 13:00 fw.hex
-rwxr-xr-x 1 babar babar 1247 avril 16 17:45 upload.py
```

Le fichier `upload.py` extrait de l'archive contient des informations intéressantes pour le reste du challenge. Celui-ci étant relativement court, l'intégralité de son contenu est présenté ci-après :

```
#!/usr/bin/env python

import socket, select

#
# Microcontroller architecture appears to be undocumented.
# No disassembler is available.
#
# The datasheet only gives us the following information:
#
#   == 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)            /
#

FIRMWARE = "fw.hex"
```

```
print("--------------------------------------------")
print("----- Microcontroller firmware uploader -----")
print("--------------------------------------------")
print()

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('178.33.105.197', 10101))

print(":: Serial port connected.")
print(":: Uploading firmware... ", end='')

[ s.send(line) for line in open(FIRMWARE, 'rb') ]

print("done.")
print()

resp = b''
while True:
    ready, _, _ = select.select([s], [], [], 10)
    if ready:
        try:
            data = s.recv(32)
        except:
            break
        if not data:
            break
        resp += data
    else:
        break

print(resp.decode("utf-8"))
s.close()
```

La lecture du script nous donne les informations suivantes :

— le fichier `fw.hex` est envoyé ligne par ligne à l'adresse IP `178.33.105.197`, port 10101, afin d'être exécuté par un microcontrôleur 16 bits (l'espace mémoire s'étalant de 0 à `0xffff`) ;
— l'espace mémoire est divisé en six segments, eux-mêmes répartis au sein d'un mode « utilisateur » et d'un mode « privilégié ».

Au sein de la partie privilégiée, on retrouve un segment nommé « Secret memory area ». Il est fort probable que l'objectif de cette étape est d'accéder au contenu de ce segment.

Le programme correspondant au fichier `fw.hex` contient les données suivantes :

```
$ cat fw.hex
:100000002100111B2001108CC0D2201010002101F2
:10001000117C2200120FC03C20101000210111B2EF
:1000200022001229C07620111000C0B4C0B65A00B8
:10003000210011242001 10B2C0BE51AAC10A210022
[...]
:1001B0000A00942B506FAE0CBB1F39B4D8CA05FD92
:1001C0008A0F5AE8B5D40D6CE86AA6ACC492F8F16F
:0C01D00072A77CE6D5A5680921D4410087
:00000001FF%
```

Enfin, l'exécution du script `upload.py` retourne le résultat suivant :

```
$ python3 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.
```

3.2 Analyse du micro-logiciel

3.2.1 Compréhension du format du programme

Comme le nom du fichier le laisse supposer, les données du fichier `fw.hex` semblent codées sous forme hexadécimale.

La modification d'un octet du fichier (par exemple transformer le premier `10` en `00`) et l'envoi au microcontrôleur aboutit au résultat ci-dessous :

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

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

CLOSING: bad checksum.
```

Un contrôle d'intégrité est donc effectué au niveau des données envoyées. Avant d'être capable d'interagir avec le microcontrôleur en envoyant un programme modifié, il est donc nécessaire de comprendre le fonctionnement de ce mécanisme de contrôle.

En cherchant les termes « program hex format », on peut identifier la page Wikipedia sur le format « Intel HEX » [1].

Une ligne au format Intel HEX est constituée des champs suivants :

- le caractère `:`
- le premier octet correspond à la longueur des données envoyées ;
- la deuxième valeur sur 16 bits correspond à l'adresse en mémoire où seront chargées les données ;
- le quatrième octet décrit le type d'informations envoyées : 0 pour des données, 1 pour signifier la fin du programme ;
- les données à proprement parler (dont la longueur correspond à la première valeur) ;
- enfin la somme de contrôle correspondant à la ligne.

La somme de contrôle d'une ligne peut être recalculée en soustrayant un par un tous les octets qui composent la ligne, à l'exception du caractère `:`.

Le code Ruby ci-dessous recalcule la somme de contrôle pour une ligne de données :

```
2.1.0 :001 > s = "100000002100111B2001108CC0D2201010002101"
 => "100000002100111B2001108CC0D2201010002101"
2.1.0 :002 > s.scan(/../).map {|x| x.to_i(16)}.inject(0) {|cksum, x| (cksum - x) & 0xff }.to_s(16)
 => "f2"
```

1. http://en.wikipedia.org/wiki/Intel_HEX

```
2.1.0 :003 >
```

On retrouve bien la valeur à la fin de la première ligne. Maintenant que la méthode de calcul de la somme de contrôle est connue, il est alors possible de modifier le contenu du programme et d'observer le comportement du microcontrôleur. Par exemple, en remplaçant le premier octet de données de la première ligne qui vaut `0x21` par 0 et en mettant à jour la somme de contrôle, on obtient la ligne suivante :

```
$ cat fw.hex
:100000000000111B2001108CC0D220101000210113
[...]
:00000001FF
```

L'exécution retourne alors :

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

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

System reset.
-- Exception occurred at 0000: Invalid instruction.
   r0:0000     r1:0000    r2:0000    r3:0000
   r4:0000     r5:0000    r6:0000    r7:0000
   r8:0000     r9:0000   r10:0000   r11:0000
  r12:0000    r13:EFFE   r14:0000   r15:0000
   pc:0000 fault_addr:0000 [S:0 Z:0] Mode:user
CLOSING: Invalid instruction.
```

Une exception de type « Invalid instruction » est déclenchée à l'adresse 0, ce qui correspond à la modification effectuée. On observe que le microcontrôleur dispose de 16 registres `r0` à `r15`, d'un registre `pc` indiquant l'adresse de l'instruction en cours d'exécution, de deux drapeaux de conditions (`S` pour « signe » et `Z` pour « zero ») et enfin du mode courant du microcontrôleur (utilisateur ou privilégié). Enfin, l'adresse 0 semble être le point d'entrée du programme, les registres valant tous (sauf pour `r13`) la valeur 0.

3.2.2 Identification du jeu d'instructions

La démarche générale pour identifier le jeu d'instruction est d'injecter des erreurs au niveau du programme et d'observer le résultat, notamment juste avant et après l'exécution d'une instruction pour constater des différences au niveau de l'état des registres.

Le deuxième octet du programme original vaut `0x00`. Cet octet ne correspond donc pas à une instruction car il a été observé précédemment qu'une instruction `0x00` déclenche une exception de type « Invalid instruction ». Il s'agit donc sans doute d'une opérande.

Une modification du deuxième octet de `0x00` à `0x001` retourne le résultat suivant :

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

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

Traceback (most recent call last):
  File "upload.py", line 53, in <module>
```

```
    print(resp.decode("utf-8"))
UnicodeDecodeError: 'utf-8' codec can't decode byte 0x94 in position 52: invalid start byte
```

Le programme termine mais les données ne semblent pas être au format unicode.

Ensuite, une modification du troisième octet de `0x11` à `00` déclenche une exception de type « Invalid instruction » à l'adresse `0x0002` :

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

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

System reset.
-- Exception occurred at 0002: Invalid instruction.
   r0:0000     r1:0000    r2:0000    r3:0000
   r4:0000     r5:0000    r6:0000    r7:0000
   r8:0000     r9:0000   r10:0000   r11:0000
  r12:0000    r13:EFFE   r14:0000   r15:0000
   pc:0002 fault_addr:0000 [S:0 Z:1] Mode:user
CLOSING: Invalid instruction.
```

On peut alors en déduire que les instructions sont alignées sur deux octets et que chaque instruction utilise une opérande d'un octet.

Pour simplifier la démarche, l'outil `loader.rb` a été développé qui permet de décoder le programme, de modifier les données binaires, de réencoder au format attendu (avec mise à jour des sommes de contrôle de chaque line) et d'envoyer le résultat au microcontrôleur pour exécution.

```
$ ./loader.rb --help
Usage: loader.rb [options]
    -i, --input FILE                 Specify input file
    -d, --decode [OUTPUT]            Decode a program to binary data
    -e, --encode [OUTPUT]            Encode a program from binary data
    -s, --send                       Send and execute a program
    -p, --patch ADDR1=V1, ADDR2=V2   Patch specified addresses
    -h, --help                       Show this message
$ ./loader.rb --input fw.hex --decode --patch 2=0 --encode --send
[+] patching
[0x02] 0x11 -> 0x00
System reset.
-- Exception occurred at 0002: Invalid instruction.
   r0:0000     r1:0000    r2:0000    r3:0000
   r4:0000     r5:0000    r6:0000    r7:0000
   r8:0000     r9:0000   r10:0000   r11:0000
  r12:0000    r13:EFFE   r14:0000   r15:0000
   pc:0002 fault_addr:0000 [S:0 Z:1] Mode:user
CLOSING: Invalid instruction.
```

Il est possible d'utiliser l'outil pour enregistrer l'état des données après chaque étape, par exemple pour obtenir le programme sous forme binaire :

```
$ ./loader.rb --input fw.hex --decode fw.bin
$ hexdump -C fw.bin
00000000  21 00 11 1b 20 01 10 8c  c0 d2 20 10 10 00 21 01  |!... ..... ...!.|
00000010  11 7c 22 00 12 0f c0 3c  20 10 10 00 21 01 11 b2  |.|"....< ...!...|
00000020  22 00 12 29 c0 76 20 11  10 00 c0 b4 c0 b6 5a 00  |"..).v .......Z.|
00000030  21 00 11 24 20 01 10 b2  c0 be 51 aa c1 0a 21 00  |!..$ .....Q...!.|
[...]
00000170  18 30 68 82 f8 03 54 44  a7 de d0 0f 59 65 61 68  |.0h...TD....Yeah|
```

```
00000180  52 69 73 63 49 73 47 6f  6f 64 21 00 46 69 72 6d  |RiscIsGood!.Firm|
00000190  77 61 72 65 20 76 31 2e  33 33 2e 37 20 73 74 61  |ware v1.33.7 sta|
000001a0  72 74 69 6e 67 2e 0a 00  48 61 6c 74 69 6e 67 2e  |rting...Halting.|
000001b0  0a 00 94 2b 50 6f ae 0c  bb 1f 39 b4 d8 ca 05 fd  |...+Po....9.....|
000001c0  8a 0f 5a e8 b5 d4 0d 6c  e8 6a a6 ac c4 92 f8 f1  |..Z....l.j......|
000001d0  72 a7 7c e6 d5 a5 68 09  21 d4 41 00              |r.|...h.!.A.|
```

Les chaînes de caractères suivantes sont présentes au sein du programme :

— « YeahRiscIsGood ! »
— « Firmware v1.33.7 starting. »
— « Halting. »

Les deux dernières chaînes sont affichées lors de l'exécution du programme, par contre le rôle de la chaîne « YeahRiscIsGood ! » est encore inconnu. On remarque également la présence de données binaires après la chaîne « Halting. », au décalage `0x1b2`.

L'outil permet également, via le paramètre `test`, de comparer l'état des registres avant et après exécution. Pour cela, la fonction ci-dessous a été ajoutée au programme `loader.rb`.

```
def test_ins(addr)
  puts "[+] testing ins at address 0x%x : 0x%2.2x 0x%2.2x" %
    [addr, @binary[addr], @binary[addr + 1]]
  orig_binary = @binary.dup
  @binary[addr] = 0
  e1 = encode.send.parse_result

  @binary = orig_binary
  @binary[addr + 2] = 0
  encode
  e2 = encode.send.parse_result

  puts "Differences:"
  diff_exceptions(e1, e2)
end
```

Le résultat sur la première instruction est :

```
$ ./loader.rb --input fw.hex --decode --test 0
[+] testing ins at address 0x0 : 0x21 0x00
Differences:
pc      : 0x0 => 0x2
Z       : 0x0 => 0x1
exception_addr  : 0x0 => 0x2
```

Les deux seules différentes sont la modification du registre `pc`, ce qui était attendu, et le passage du drapeau `Z` de 0 à 1. Pour aller plus loin, il est alors possible de modifier la valeur de la première opérande :

```
$ ./loader.rb --input fw.hex --decode --patch 1=1 --test 0
[+] patching
[0x01] 0x00 -> 0x01
[+] testing ins at address 0x0 : 0x21 0x01
Differences:
r1      : 0x0 => 0x100
pc      : 0x0 => 0x2
exception_addr  : 0x0 => 0x2
```

Cette fois, la valeur du registre `r1` est modifiée. L'opérande modifiée se retrouve dans l'octet de poids fort du registre `r1`. On peut continuer de jouer avec la valeur de l'opérande pour confirmer cette hypothèse :

```
$ ./loader.rb --input fw.hex --decode --patch 1=0xab --test 0
[+] patching
```

```
[0x01] 0x00 -> 0xab
[+] testing ins at address 0x0 : 0x21 0xab
Differences:
r1      : 0x0 => 0xab00
pc      : 0x0 => 0x2
S       : 0x0 => 0x1
exception_addr  : 0x0 => 0x2
```

Le numéro du registre modifié correspond au nibble de poids faible de l'opcode `0x21`. De même, pour confirmer cette nouvelle hypothèse, il est possible de modifier la valeur de l'opcode :

```
$ ./loader.rb --input fw.hex --decode --patch 0=0x22,1=0xab --test 0
[+] patching
[0x00] 0x21 -> 0x22
[0x01] 0x00 -> 0xab
[+] testing ins at address 0x0 : 0x22 0xab
Differences:
r2      : 0x0 => 0xab00
pc      : 0x0 => 0x2
S       : 0x0 => 0x1
exception_addr  : 0x0 => 0x2
```

Le registre `r2` est cette fois modifié.

Ces expérimentations permettent de comprendre la sémantique de la première instruction du programme : la valeur de l'opérande est chargée dans l'octet de poids fort du registre correspond au nibble de poids faible de l'opcode.

On s'attaque ensuite à la seconde instruction du programme :

```
$ ./loader.rb --input fw.hex --decode --test 2
[+] testing ins at address 0x2 : 0x11 0x1b
Differences:
r1      : 0x0 => 0x1b
pc      : 0x2 => 0x4
Z       : 0x1 => 0x0
exception_addr  : 0x2 => 0x4
```

Cette fois, l'octet de poids faible du registre `r1` contient la valeur de l'opérande. Par analogie avec la première instruction, on peut supposer que le registre est spécifié par le nibble de poids faible de l'opcode. Le test suivant permet de confirmer cette hypothèse :

```
$ ./loader.rb --input fw.hex --decode --patch 2=0x12 --test 2
[+] patching
[0x02] 0x11 -> 0x12
[+] testing ins at address 0x2 : 0x12 0x1b
Differences:
r2      : 0x0 => 0x1b
pc      : 0x2 => 0x4
Z       : 0x1 => 0x0
exception_addr  : 0x2 => 0x4
```

Le format d'une instruction est décrit à la figure 3.1, où `N` est le numéro de l'instruction, `OP0` une opérande sur 4 bits qui spécifie le registre de destination et `OP1` une opérande sur 8 bits utilisée comme valeur immédiate.

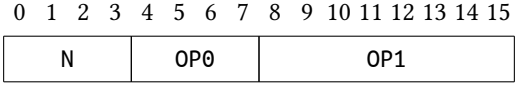

FIGURE 3.1 – Format d'une instruction du microcontrôleur

On peut alors s'intéresser à la troisième instruction à l'adresse 4.

```
$ ./loader.rb --input fw.hex --decode --test 4
```

```
[+] testing ins at address 0x4 : 0x20 0x01
Differences:
r0      : 0x0 => 0x100
pc      : 0x4 => 0x6
exception_addr  : 0x4 => 0x6
```

L'instruction numéro 2, déjà étudiée à l'adresse 0, est alors exécutée et les modifications au niveau des registres correspondent aux conclusions précédemment énoncées. De même, l'instruction à l'adresse 6 est la même que celle étudiée à l'adresse 2 :

```
$ ./loader.rb --input fw.hex --decode --test 6
[+] testing ins at address 0x6 : 0x10 0x8c
Differences:
r0      : 0x100 => 0x18c
pc      : 0x6 => 0x8
exception_addr  : 0x6 => 0x8
```

Par contre, l'instruction à l'adresse 8 est inconnue :

```
$ ./loader.rb --input fw.hex --decode --test 8
[+] testing ins at address 0x8 : 0xc0 0xd2
Differences:
r15     : 0x0 => 0xa
pc      : 0x8 => 0xa
S       : 0x0 => 0x1
Z       : 0x0 => 0x1
exception_addr  : 0x8 => 0xa
```

On remarque que le registre `r15` est modifié avec la valeur `0xa`, qui correspond à l'adresse de la prochaine instruction à exécuter. Pour en savoir plus sur l'instruction `0xc`, il est possible de jouer avec la valeur de l'opérande `OP1`.

```
$ ./loader.rb --input fw.hex --decode --patch 9=0xd3 --test 8
[+] patching
[0x09] 0xd2 -> 0xd3
[+] testing ins at address 0x8 : 0xc0 0xd3
Differences:
r15     : 0x0 => 0xa
pc      : 0x8 => 0xdd
exception_addr  : 0x8 => 0xdd
exception message: Invalid instruction. => Unaligned instruction.
```

Le résultat obtenu est intéressant : le message de l'exception ne parle plus d'instruction invalide mais d'un problème d'alignement. Effectivement, le registre `pc` a pour valeur `0xdd`, soit 221, qui est impair et donc ne correspond pas au début d'une instruction. De plus, la modification du registre `r15` semble correspondre à la sauvegarde d'une adresse de retour. Enfin, la valeur `0xdd` correspond à la somme de l'adresse sauvegardée dans `r15` et de la valeur de l'opérande `OP1` : `0xdd = 0x8 + 2 + 0xd3 = 0xa + 0xd3`.

On retrouve donc la même sémantique que l'instruction `BL` sur une architecture ARM, c'est-à-dire :

— une sauvegarde de l'adresse de retour dans le registre `r15`, qui sert alors de « link register » ;
— un mode d'adresse relatif à la valeur du registre `pc`.

L'instruction `0xc` peut donc être assimilé à un appel de fonction.

La prochaine instruction inconnue se situe à l'adresse `46`.

```
$ ./loader.rb --input fw.hex --decode --test 46
[+] testing ins at address 0x2e : 0x5a 0x00
Differences:
r10     : 0x29 => 0x2093
```

```
pc      : 0x2e => 0x30
exception_addr  : 0x2e => 0x30
```

La valeur du registre `r10` a été modifiée avec la valeur `0x2093`. Pour comprendre d'où vient cette valeur, la connaissance de l'état des registres avant l'exécution de l'instruction est utile :

```
$ ./loader.rb --input fw.hex --decode --patch 46=0 --encode -send
[+] patching
[0x2e] 0x5a -> 0x00
System reset.
Firmware v1.33.7 starting.
-- Exception occurred at 002E: Invalid instruction.
   r0:2093     r1:0000    r2:0100    r3:0093
   r4:2000     r5:0000    r6:000A    r7:0000
   r8:1000     r9:01B2   r10:0029   r11:0000
  r12:0000    r13:EFFE   r14:0000   r15:002E
   pc:002E fault_addr:0000 [S:0 Z:0] Mode:user
CLOSING: Invalid instruction.
```

La valeur `0x2093` est présente dans le registre `r0`. Pour confirmer que l'instruction `0x5` corresponde bien à une affectation entre registres, des modifications sont effectuées sur la valeur de l'opérande `OP1` :

```
$ ./loader.rb --input fw.hex --decode --patch 47=0x01 --test 46
[+] patching
[0x2f] 0x00 -> 0x01
[+] testing ins at address 0x2e : 0x5a 0x01
Differences:
r10     : 0x29 => 0x0
pc      : 0x2e => 0x30
Z       : 0x0 => 0x1
exception_addr  : 0x2e => 0x30
```

Cette fois le registre `r10` prend la valeur 0, qui correspond à la valeur du registre `r1` spécifié par l'opérande `OP1`. Par contre, un test avec `0x2` comme opérande contredit l'hypothèse d'une affectation entre registres :

```
$ ./loader.rb --input fw.hex --decode --patch 47=0x02 --test 46
[+] patching
[0x2f] 0x00 -> 0x02
[+] testing ins at address 0x2e : 0x5a 0x02
Differences:
r10     : 0x29 => 0x0
pc      : 0x2e => 0x30
Z       : 0x0 => 0x1
exception_addr  : 0x2e => 0x30
```

La valeur du registre `r10` ne correspond pas à celle du registre `r2`, à savoir `0x100`. En continuant à incrémenter la valeur de `OP1`, on obtient une résultat intéressant avec la valeur `0x0d` :

```
$ ./loader.rb --input fw.hex --decode --patch 47=0x0d --test 46
[+] patching
[0x2f] 0x00 -> 0x0d
[+] testing ins at address 0x2e : 0x5a 0x0d
Differences:
r10     : 0x29 => 0x2092
pc      : 0x2e => 0x30
exception_addr  : 0x2e => 0x30
```

La nouvelle valeur du registre `r10` ne correspond pas à la valeur du registre `r13`, à savoir `0xeffe`. Par contre, on retrouve une valeur proche de la valeur du registre `r0`, à un décalage de 1 près. On peut alors supposer que le registre `r0` intervient dans l'exécution de l'instruction. De plus, le nibble de poids fort de l'opérande `OP1`, c'est-à-dire 0, correspond bien à `r0`. Pour déterminer la sémantique de l'instruction, il est alors nécessaire de tester les différentes

opérations habituelles entre les deux registres :

- `r13 + r0 = 0xeffe + 0x2093 = 0x1091`
- `r13 - r0 = 0xeffe - 0x2093 = 0xcf6b`
- `r13 ^ r0 = 0xeffe ^ 0x2093 = 0xcf6d`
- `r13 | r0 = 0xeffe | 0x2093 = 0xefff`
- `r13 & r0 = 0xeffe & 0x2093 = 0x2092`

La valeur de `r10` correspond bien au résultat d'un « et » logique entre les registres `r13` et `r0`.

En suivant une démarche similaire pour chaque nouvelle instruction inconnue, on déduit finalement le jeu d'instruction correspondant aux opérations sur les registres. Le résultat est présenté dans le tableau 3.1.

Mnémonique	Description	`N` (4 bits)	`OP0` (4 bits)	`OP1` (8 bits)	
`movh`	Stocke `imm` comme octet de poids fort du registre `rd`	1	`rd`	`imm`	
`movl`	Stocke `imm` comme octet de poids faible du registre `rd`	2	`rd`	`imm`	
`xor`	Stocke dans `rd` le résultat d'un « ou-exclusif » entre `rn` et `rm`	3	`rd`	`rn`	`rm`
`or`	Stocke dans `rd` le résultat d'un « ou » logique entre `rn` et `rm`	4	`rd`	`rn`	`rm`
`and`	Stocke dans `rd` le résultat d'un « et » logique entre `rn` et `rm`	5	`rd`	`rn`	`rm`
`add`	Stocke dans `rd` le résultat de l'addition entre `rn` et `rm`	6	`rd`	`rn`	`rm`
`sub`	Stocke dans `rd` le résultat de la soustraction entre `rn` et `rm`	7	`rd`	`rn`	`rm`
`imul`	Stocke dans `rd` le résultat de la multiplication entre `rn` et `rm`	8	`rd`	`rn`	`rm`
`idiv`	Stocke dans `rd` le résultat de la division entre `rn` et `rm`	9	`rd`	`rn`	`rm`

Table 3.1 – Opérations sur les registres

Les opérations de branchement sont présentées sur le tableau 3.2.

Mnémonique	Description	`N` (4 bits)	`OP0` (2 bits)	`OP1` (10 bits)	
`jmpcc`	Saut conditionnel à l'adresse spécifiée par `imm`	10	`cond`	`imm`	
`jmp`	Saut inconditionnel à l'adresse spécifiée par `imm`	11		`imm`	
`call`	Appel de la fonction à l'adresse spécifiée par `imm`	12	mode	`imm`	
`ret`	Retour d'appel de la fonction à l'adresse stockée dans le registre `rn`	13	mode	0	`rn` (8 bits)

Table 3.2 – Opérations de branchement

Enfin, les deux opérations d'accès à la mémoire sont présentées sur le tableau 3.3.

Mnémonique	Description	`N` (4 bits)	`OP0` (4 bits)	`OP1` (8 bits)	
`ldrb`	Charge dans `rd` l'octet en mémoire à l'adresse `rn + rm`	14	`rd`	`rn`	`rm`
`strb`	Stocke en mémoire à l'adresse `rn + rm` l'octet de poids faible de `rd`	15	`rd`	`rn`	`rm`

Table 3.3 – Opérations de lecture / écriture en mémoire

La connaissance du jeu d'instructions permet de commencer à développer un désassembleur et un assembleur.

Cependant, il reste à déterminer la gestion des conditions pour l'instruction 10 (saut conditionnel) et le mode d'adressage utilisé pour les instructions `jmp` et `call`.

Pour cela, la méthode suivante est adoptée :

— pour chaque combinaison possible des drapeaux `S` et `Z`, un programme est écrit permettant d'obtenir la combinaison souhaitée ;
— pour chacune des valeurs possibles pour `OP0` (c'est-à-dire de 0 à 3), chaque programme est exécuté et la valeur du registre `pc` permet de savoir si le saut a été pris ou non.

Le programme Ruby ci-dessous permet d'automatiser cette méthode.

```
require 'loader'

loader = Loader.new

(0..3).each do |i|
  s =<<EOS
movh r0, 0x00  ; pc = 0x0
movl r0, 0x00  ; pc = 0x2
movh r1, 0x00  ; pc = 0x4
movl r1, 0x01  ; pc = 0x6
add r2, r0, r1 ; pc = 0x8
jmp #{i}, 0xab ; pc = 0xa
EOS
  r = loader.assemble_string(s).encode.send.parse_result
  puts "%d S:%d Z:%d => 0x%4.4x" % [ i, r[:S], r[:Z], r[:pc] ]

  s =<<EOS
movh r0, 0x00  ; pc = 0x0
movl r0, 0x00  ; pc = 0x2
movh r1, 0x00  ; pc = 0x4
movl r1, 0x00  ; pc = 0x6
add r2, r0, r1 ; pc = 0x8
jmp #{i}, 0xab ; pc = 0xa
EOS
  r = loader.assemble_string(s).encode.send.parse_result
  puts "%d S:%d Z:%d => 0x%4.4x" % [ i, r[:S], r[:Z], r[:pc] ]

  s =<<EOS
movh r0, 0xf0  ; pc = 0x0
movl r0, 0x00  ; pc = 0x2
movh r1, 0xf0  ; pc = 0x4
movl r1, 0x00  ; pc = 0x6
add r2, r0, r1 ; pc = 0x8
jmp #{i}, 0xab ; pc = 0xa
EOS
  r = loader.assemble_string(s).encode.send.parse_result
  puts "%d S:%d Z:%d => 0x%4.4x" % [ i, r[:S], r[:Z], r[:pc] ]
end
```

Le résultat obtenu est consigné dans le tableau 3.4.

OP1	S:0 Z:0	S:0 Z:1	S:1 Z:0
0	0x000c	0x00b7	0x000c
1	0x00b7	0x000c	0x00b7
2	0x000c	0x000c	0x00b7
3	0x00b7	0x00b7	0x000c

TABLE 3.4 – Valeur du registre `pc` en fonction de `OP0` et des drapeaux de condition

Une valeur pour le registre `pc` différente de `0x000c` signifie que le branchement a été pris. A partir de ces résultats,

on peut en tirer la conclusion suivante sur la valeur de `OP0` :

— 0 : le saut est pris si `Z = 1` ;
— 1 : le saut est pris si `Z = 0` ;
— 2 : le saut est pris si `S = 1` ;
— 3 : le saut est pris si `S = 0`.

La destination du saut, s'il est pris, correspond à la valeur signée de l'opérande `OP1` qui est codée sur 10 bits.

A ce stade, on pense disposer de toutes les informations nécessaires pour finaliser le désassembleur. Cependant, il reste une incertitude sur la sémantique des opérandes par l'instruction `call`. En effet, dans certain cas, la valeur de l'opérande `OP1` semble curieuse, comme le montre les commandes ci-dessous.

```
$ ./loader.rb --input fw.hex --decode --disas fw.asm
$ grep call fw.asm
call 0x0, 0xd2  ; loc_8 : 0xc0 0xd2
call 0x0, 0x3c  ; loc_16 : 0xc0 0x3c
call 0x0, 0x76  ; loc_24 : 0xc0 0x76
call 0x0, 0xb4  ; loc_2a : 0xc0 0xb4
call 0x0, 0xb6  ; loc_2c : 0xc0 0xb6
call 0x0, 0xbe  ; loc_38 : 0xc0 0xbe
call 0x1, 0xa   ; loc_3c : 0xc1 0x0a
call 0x0, 0x94  ; loc_46 : 0xc0 0x94
call 0x0, 0x8a  ; loc_50 : 0xc0 0x8a
call 0x8, 0x1   ; loc_d8 : 0xc8 0x01
call 0x8, 0x2   ; loc_dc : 0xc8 0x02
call 0x8, 0x3   ; loc_e0 : 0xc8 0x03
```

Les trois derniers appels à `call` ont pour opérande `OP1` les valeurs 1, 2 et 3. De plus, l'opérande `OP0` vaut 8 pour ces trois appels. La sémantique de cette opérande n'étant pas connue, le désassemblage des trois derniers appels à `call` fait apparaître les valeurs brutes des opérandes tandis que pour les autres appel, la valeur de l'opérande `OP1` est remplacée par l'adresse de destination. Le résultat est alors le suivant :

```
$ ./loader.rb --input fw.hex --decode --disas fw.asm
$ grep call fw.asm
call sub_dc     ; loc_8 : 0xc0 0xd2
call sub_54     ; loc_16 : 0xc0 0x3c
call sub_9c     ; loc_24 : 0xc0 0x76
call sub_e0     ; loc_2a : 0xc0 0xb4
call sub_e4     ; loc_2c : 0xc0 0xb6
call sub_f8     ; loc_38 : 0xc0 0xbe
call sub_148    ; loc_3c : 0xc1 0x0a
call sub_dc     ; loc_46 : 0xc0 0x94
call sub_dc     ; loc_50 : 0xc0 0x8a
call 0x8, 0x1   ; loc_d8 : 0xc8 0x01
call 0x8, 0x2   ; loc_dc : 0xc8 0x02
call 0x8, 0x3   ; loc_e0 : 0xc8 0x03
```

Il reste maintenant à comprendre le résultat du désassemblage, ce qui fait l'objet du chapitre suivant.

3.3 Désassemblage de `fw.hex`

Le fichier `fw.asm` obtenu par désassemblage est disponible en annexe A.3.

Le désassembleur développé suit les potentiels branchements pris par les instructions et ne désassemble que le code qui semble atteignable. Ce mode de fonctionnement permet d'identifier des zones de données au sein du programme (par exemple les chaînes de caractères présentées au paragraphe 3.2.2) ou du code mort.

3.3.1 Analyse du point d'entrée

Comme évoqué précédemment, on suppose que le programme commence à l'adresse 0. Le désassemblage des instructions à cette adresse donne le résultat ci-dessous :

```
movh r1, 0x0    ; loc_0 : 0x21 0x00
movl r1, 0x1b   ; loc_2 : 0x11 0x1b
movh r0, 0x1    ; loc_4 : 0x20 0x01
movl r0, 0x8c   ; loc_6 : 0x10 0x8c
call sub_dc     ; loc_8 : 0xc0 0xd2
movh r0, 0x10   ; loc_a : 0x20 0x10
movl r0, 0x0    ; loc_c : 0x10 0x00
movh r1, 0x1    ; loc_e : 0x21 0x01
movl r1, 0x7c   ; loc_10 : 0x11 0x7c
movh r2, 0x0    ; loc_12 : 0x22 0x00
movl r2, 0xf    ; loc_14 : 0x12 0x0f
call sub_54     ; loc_16 : 0xc0 0x3c
movh r0, 0x10   ; loc_18 : 0x20 0x10
movl r0, 0x0    ; loc_1a : 0x10 0x00
movh r1, 0x1    ; loc_1c : 0x21 0x01
movl r1, 0xb2   ; loc_1e : 0x11 0xb2
movh r2, 0x0    ; loc_20 : 0x22 0x00
movl r2, 0x29   ; loc_22 : 0x12 0x29
call sub_9c     ; loc_24 : 0xc0 0x76
movh r0, 0x11   ; loc_26 : 0x20 0x11
movl r0, 0x0    ; loc_28 : 0x10 0x00
call sub_e0     ; loc_2a : 0xc0 0xb4
call sub_e4     ; loc_2c : 0xc0 0xb6
and r10, r0, r0 ; loc_2e : 0x5a 0x00
movh r1, 0x0    ; loc_30 : 0x21 0x00
movl r1, 0x24   ; loc_32 : 0x11 0x24
movh r0, 0x1    ; loc_34 : 0x20 0x01
movl r0, 0xb2   ; loc_36 : 0x10 0xb2
call sub_f8     ; loc_38 : 0xc0 0xbe
and r1, r10, r10        ; loc_3a : 0x51 0xaa
call sub_148    ; loc_3c : 0xc1 0x0a
movh r1, 0x0    ; loc_3e : 0x21 0x00
movl r1, 0x29   ; loc_40 : 0x11 0x29
movh r0, 0x1    ; loc_42 : 0x20 0x01
movl r0, 0xb2   ; loc_44 : 0x10 0xb2
call sub_dc     ; loc_46 : 0xc0 0x94
movh r1, 0x0    ; loc_48 : 0x21 0x00
movl r1, 0x9    ; loc_4a : 0x11 0x09
movh r0, 0x1    ; loc_4c : 0x20 0x01
movl r0, 0xa8   ; loc_4e : 0x10 0xa8
call sub_dc     ; loc_50 : 0xc0 0x8a
jmp loc_d8      ; loc_52 : 0xb0 0x84
[...]
loc_d8:
call 0x8, 0x1    ; loc_d8 : 0xc8 0x01
jmp loc_d8    ; loc_da : 0xb3 0xfc
```

Le code C présenté ci-après est équivalent au code assembleur obtenu.

```
void main(void) {
    uint16_t r10;

    sub_dc(0x18c, 0x1b);
    sub_54(0x1000, 0x17c, 0xf);
    sub_9c(0x1000, 0x1b2, 0x29);
    sub_e0(0x1100);
    r10 = sub_e4(0x1100);
    r0 = sub_f8(0x1b2, 0x24);
    sub_148(r0, r10);
```

```
    sub_dc(0x1b2, 0x29);
    sub_dc(0x1a8, 9);
    /* call 0x8, 0x1 */
}
```

Pour comprendre le rôle de l'instruction `call 0x8, 0x1`, une faute est injectée à l'adresse `0xd8` :

```
$ ./loader.rb --input fw.hex --decode --patch 0xd8=0 --encode --send
[+] patching
[0xd8] 0xc8 -> 0x00
System reset.
Firmware v1.33.7 starting.
Execution completed in  8339 CPU cycles.
Halting.
-- Exception occurred at 00D8: Invalid instruction.
   r0:01A8     r1:0009    r2:0009    r3:0000
   r4:0000     r5:000A    r6:000B    r7:0001
   r8:0039     r9:01B2   r10:2093   r11:0000
  r12:0000    r13:EFFE   r14:0000   r15:0052
   pc:00D8 fault_addr:0000 [S:1 Z:1] Mode:user
CLOSING: Invalid instruction.
```

Le programme semble avoir terminé son exécution car la chaîne « Halting. » est affichée mais une erreur est néanmoins déclenchée à l'adresse de l'instruction `call 0x8, 0x1`. Cette instruction pourrait correspondre à une sortie du programme.

Pour confirmer cette hypothèse, l'instruction suivante est modifée avec la valeur 0 pour déclencher une exception :

```
$ ./loader.rb --input fw.hex --decode --patch 0xda=0 --encode --send
[+] patching
[0xda] 0xb3 -> 0x00
System reset.
Firmware v1.33.7 starting.
Execution completed in  8339 CPU cycles.
Halting.
```

Le programme termine normalement, sans erreur. L'instruction `call 0x8, 0x1` a bien provoquée une sortie du programme. On peut donc assimiler cette instruction à l'appel système `exit`.

3.3.2 Analyse de la fonction `sub_dc`

Le code de la fonction `sub_dc` est présenté ci-dessous.

```
sub_dc:
call 0x8, 0x2    ; loc_dc : 0xc8 0x02
ret 0, 0xf    ; loc_de : 0xd0 0x0f
```

La fonction appelle l'appel système 2 puis retourne. Pour comprendre l'utilité de cet appel, il est possible d'injecter une faute à l'adresse de l'instruction puis à l'adresse suivante.

```
$ ./loader.rb --input fw.hex --decode --patch 0xdc=0 --encode --send
[+] patching
[0xdc] 0xc8 -> 0x00
System reset.
-- Exception occurred at 00DC: Invalid instruction.
   r0:018C     r1:001B    r2:0000    r3:0000
   r4:0000     r5:0000    r6:0000    r7:0000
   r8:0000     r9:0000   r10:0000   r11:0000
  r12:0000    r13:EFFE   r14:0000   r15:000A
```

```
   pc:00DC fault_addr:0000 [S:0 Z:0] Mode:user
CLOSING: Invalid instruction.
$ ./loader.rb --input fw.hex --decode --patch 0xde=0 --encode --send
[+] patching
[0xde] 0xd0 -> 0x00
System reset.
Firmware v1.33.7 starting.
-- Exception occurred at 00DE: Invalid instruction.
   r0:018C     r1:001B    r2:0000    r3:0000
   r4:0000     r5:0000    r6:0000    r7:0000
   r8:0000     r9:0000   r10:0000   r11:0000
  r12:0000    r13:EFFE   r14:0000   r15:000A
   pc:00DE fault_addr:0000 [S:1 Z:1] Mode:user
CLOSING: Invalid instruction.
```

On peut remarquer, qu'entre les deux exécutions, la chaîne de caractères « Firmware v1.33.7 starting. » a été affichée. En examinant l'état des registres `r0` et `r1` (qui correspondent aux paramètres passés à la fonction), on peut reconnaître une adresse dans la mémoire du microcontrôleur et une longueur. Le registre `r0` correspond certainement à l'emplacement de la chaîne à afficher et le registre `r1` correspond à la longueur de la chaîne.

Cette hypothèse peut être vérifiée en extrayant la chaîne correspondante dans les données binaires du programme :

```
$ dd if=fw.bin bs=1 skip=$((0x18c)) count=$((0x1b)) 2>/dev/null
Firmware v1.33.7 starting.
```

L'instruction `call 0x8, 0x2` peut être assimilée à l'appel système `write(1, addr, len)`.

3.3.3 Analyse de la fonction `sub_54`

Le code désassemblé de la fonction `sub_54` est présenté ci-après.

```
sub_54:
and r8, r0, r0    ; loc_54 : 0x58 0x00
and r9, r1, r1    ; loc_56 : 0x59 0x11
and r10, r2, r2    ; loc_58 : 0x5a 0x22
xor r0, r0, r0    ; loc_5a : 0x30 0x00
movh r1, 0x1    ; loc_5c : 0x21 0x01
movl r1, 0x0    ; loc_5e : 0x11 0x00
movh r2, 0x0    ; loc_60 : 0x22 0x00
movl r2, 0x1    ; loc_62 : 0x12 0x01

loc_64:
sub r3, r1, r0    ; loc_64 : 0x73 0x10
jmp Z, loc_6e    ; loc_66 : 0xa0 0x06
strb r0, [r8 + r0] ; loc_68 : 0xf0 0x80
add r0, r0, r2    ; loc_6a : 0x60 0x02
jmp loc_64    ; loc_6c : 0xb3 0xf6

loc_6e:
xor r0, r0, r0    ; loc_6e : 0x30 0x00
and r1, r0, r0    ; loc_70 : 0x51 0x00
movh r2, 0x0    ; loc_72 : 0x22 0x00
movl r2, 0x1    ; loc_74 : 0x12 0x01
movh r3, 0x0    ; loc_76 : 0x23 0x00
movl r3, 0xff    ; loc_78 : 0x13 0xff

loc_7a:
ldrb r4, [r8 + r0]    ; loc_7a : 0xe4 0x80
add r1, r1, r4    ; loc_7c : 0x61 0x14
idiv r4, r0, r10    ; loc_7e : 0x94 0x0a
```

```
imul r4, r4, r10    ; loc_80 : 0x84 0x4a
sub r4, r0, r4    ; loc_82 : 0x74 0x04
ldrb r4, [r9 + r4]    ; loc_84 : 0xe4 0x94
add r1, r1, r4    ; loc_86 : 0x61 0x14
and r1, r1, r3    ; loc_88 : 0x51 0x13
ldrb r4, [r8 + r0]    ; loc_8a : 0xe4 0x80
ldrb r5, [r8 + r1]    ; loc_8c : 0xe5 0x81
strb r5, [r8 + r0]    ; loc_8e : 0xf5 0x80
strb r4, [r8 + r1]      ; loc_90 : 0xf4 0x81
add r0, r0, r2    ; loc_92 : 0x60 0x02
sub r4, r3, r0    ; loc_94 : 0x74 0x30
jmp NS, loc_7a    ; loc_96 : 0xaf 0xe2
ret 0, 0xf
```

Le code C ci-dessous est équivalent à la fonction `sub_54`

```c
void sub_54(uint16_t addr0, uint16_t addr1, uint16_t count) {
    uint16_t r1, r4, r5;
    int i;

    for (i = 0; i < 256; i++) {
        mem[addr0 + i] = i;
    }

    r1 = 0;
    for (i = 0 ; i <= 255; i++) {
        r4 = mem[addr0 + i];
        r1 += r4;

        r4 = i % count;
        r4 = mem[addr1 + r4];
        r1 = (r1 + r4) & 0xff;

        r4 = mem[addr0 + i];
        r5 = mem[addr0 + r1];
        mem[addr0 + i] = r5;
        mem[addr0 + r1] = r4;
    }
}
```

Les plus chevronnés reconnaîtront la routine d'initialisation de l'algorithme de chiffrement RC4. L'adresse stockée dans le registre `r1` correspond donc à la clé de chiffrement. D'après l'étude de la fonction principale, cette adresse vaut `0x17c`. Il est possible d'afficher le contenu de la clé en l'extrayant depuis le fichier `fw.bin`.

```
$ dd if=fw.bin bs=1 skip=$((0x17c)) count=$((0xf)) 2>/dev/null
YeahRiscIsGood!
```

3.3.4 Analyse de la fonction `sub_9c`

Le code de la fonction `sub_9c` est présenté ci-dessous.

```
sub_9c:
jmp loc_9e    ; loc_9c : 0xb0 0x00

loc_9e:
and r8, r0, r0    ; loc_9e : 0x58 0x00
and r9, r1, r1    ; loc_a0 : 0x59 0x11
and r10, r2, r2    ; loc_a2 : 0x5a 0x22
xor r0, r0, r0    ; loc_a4 : 0x30 0x00
and r1, r0, r0    ; loc_a6 : 0x51 0x00
and r2, r0, r0    ; loc_a8 : 0x52 0x00
```

```
movh r3, 0x0    ; loc_aa : 0x23 0x00
movl r3, 0xff    ; loc_ac : 0x13 0xff
movh r4, 0x0    ; loc_ae : 0x24 0x00
movl r4, 0x1    ; loc_b0 : 0x14 0x01

loc_b2:
add r0, r2, r4    ; loc_b2 : 0x60 0x24
and r0, r0, r3    ; loc_b4 : 0x50 0x03
ldrb r5, [r8 + r0]    ; loc_b6 : 0xe5 0x80
add r1, r1, r5    ; loc_b8 : 0x61 0x15
and r1, r1, r3    ; loc_ba : 0x51 0x13
ldrb r5, [r8 + r0]    ; loc_bc : 0xe5 0x80
ldrb r6, [r8 + r1]    ; loc_be : 0xe6 0x81
strb r6, [r8 + r0] ; loc_c0 : 0xf6 0x80
strb r5, [r8 + r1] ; loc_c2 : 0xf5 0x81
add r5, r5, r6    ; loc_c4 : 0x65 0x56
and r5, r5, r3    ; loc_c6 : 0x55 0x53
ldrb r5, [r8 + r5] ; loc_c8 : 0xe5 0x85
ldrb r6, [r9 + r2] ; loc_ca : 0xe6 0x92
xor r6, r6, r5    ; loc_cc : 0x36 0x65
strb r6, [r9 + r2] ; loc_ce : 0xf6 0x92
add r2, r2, r4    ; loc_d0 : 0x62 0x24
sub r5, r10, r2    ; loc_d2 : 0x75 0xa2
jmp NZ, loc_b2    ; loc_d4 : 0xa7 0xdc
ret 0, 0xf    ; loc_d6 : 0xd0 0x0f
```

Le code C ci-dessous est équivalent à la fonction `sub_9c`.

```c
void sub_9c(uint16_t addr0, uint16_t addr1, uint16_t count) {
    int i;
    uint16_t r0, r1, r5, r6;

    r1 = 0;
    for (i = 0; i < count; i++) {
        r0 = (i + 1) & 0xff;

        r5 = mem[addr0 + r0];
        r1 = (r1 + r5) & 0xff;

        r5 = mem[addr0 + r0];
        r6 = mem[addr0 + r1];

        mem[addr0 + r0] = r6;
        mem[addr0 + r1] = r5;

        r5 = (r5 + r6) & 0xff;

        r5 = mem[addr0 + r5];
        r6 = mem[addr1 + i];
        mem[addr1 + i] = r6 ^ r5;
    }
}
```

Là encore, il est possible de reconnaître la boucle de chiffrement / déchiffrement de RC4. Un affichage en hexadécimal des données obtenues après déchiffrement est présenté ci-dessous :

```
$ gcc -o decomp decomp.c
$ ./decomp
Firmware v1.33.7 starting.
0x000000: 45 78 65 63 75 74 69 6f Executio
0x000008: 6e 20 63 6f 6d 70 6c 65 n comple
0x000010: 74 65 64 20 69 6e 20 24 ted in $
0x000018: 24 24 24 24 20 43 50 55 $$$$ CPU
0x000020: 20 63 79 63 6c 65 73 2e  cycles.
```

```
0x000028: 0a
```

On retrouve la chaîne de caractères affichée lors de l'exécution du programme, à l'exception que le nombre de cycles CPU est remplacé par des caractères `$`.

Pour confirmer l'hypothèse de l'utilisation de RC4, le code Ruby [2] ci-dessous effectue des manipulations simulaires.

```
2.1.1 :001 > require 'rc4'
 => true
2.1.1 :002 > data = File.open("fw.bin", "rb").read
 => "!\x00\x11\e \x01\x10\x8C[...]\xD5\xA5h\t!\xD4A\x00"
2.1.1 :003 > key = data[0x17c, 0xf]
 => "YeahRiscIsGood!"
2.1.1 :004 > crypted = data[0x1b2, 0x29]
 => "\x94+Po\xAE\f\xBB\x1F9[...]\xA7|\xE6\xD5\xA5h\t!\xD4A"
2.1.1 :005 > rc4 = RC4.new(key)
 => #<RC4:0x000000011be350 @q2=0, @q1=0, @key=[89, 101, [...], 33, 89], @s=[61, 9, [...], 87, 51]>
2.1.1 :006 > rc4.decrypt(crypted)
 => "Execution completed in $$$$$ CPU cycles.\n"
```

On retrouve bien la même chaîne.

3.3.5 Analyse de la fonction `sub_e0`

Le code de la fonction `sub_e0` est présenté ci-dessous.

```
sub_e0:
call 0x8, 0x3    ; loc_e0 : 0xc8 0x03
ret 0, 0xf    ; loc_e2 : 0xd0 0x0f
```

Cette fonction est très simple et ne fait qu'appeler l'appel système numéro 3. La valeur `0x1100` est passée en paramètre à cette fonction, ce qui semble correspondre à une adresse mémoire. A ce stade, le rôle de l'appel système 3 reste inconnu.

3.3.6 Analyse de la fonction `sub_e4`

Le code de la fonction `sub_e4` est présenté ci-dessous.

```
sub_e4:
movh r1, 0x0    ; loc_e4 : 0x21 0x00
movl r1, 0x1    ; loc_e6 : 0x11 0x01
movh r2, 0x1    ; loc_e8 : 0x22 0x01
movl r2, 0x0    ; loc_ea : 0x12 0x00
ldrb r3, [r0 + r1]    ; loc_ec : 0xe3 0x01
sub r1, r1, r1    ; loc_ee : 0x71 0x11
ldrb r4, [r0 + r1]    ; loc_f0 : 0xe4 0x01
imul r4, r4, r2    ; loc_f2 : 0x84 0x42
or r0, r3, r4    ; loc_f4 : 0x40 0x34
ret 0, 0xf    ; loc_f6 : 0xd0 0x0f
```

Le rôle de cette fonction est de lire, à l'adresse spécifiée dans le registre `r0`, deux octets consécutifs et de les stocker dans le registre `r0` qui constitue la valeur de retour de la fonction. Autrement dit, la fonction va lire 16 bits de données à l'adresse indiquée par `r0`.

2. l'installation de la Gem `ruby-rc4` est nécessaire

La valeur `0x1100` est passée en paramètre de cette fonction. Il est donc possible d'injecter une erreur à l'adresse `0xf6` pour connaître la valeur du résultat.

```
$ ./loader.rb --input fw.hex --decode --patch 0xf6=0 --encode --send
[+] patching
[0xf6] 0xd0 -> 0x00
System reset.
Firmware v1.33.7 starting.
-- Exception occurred at 00F6: Invalid instruction.
   r0:2093     r1:0000    r2:0100    r3:0093
   r4:2000     r5:0000    r6:000A    r7:0000
   r8:1000     r9:01B2   r10:0029   r11:0000
  r12:0000    r13:EFFE   r14:0000   r15:002E
   pc:00F6 fault_addr:0000 [S:0 Z:0] Mode:user
CLOSING: Invalid instruction.
```

La valeur de retour de la fonction est `0x2093`, qui correspond à 8339 en décimal. Il s'agit du nombre de cycles CPU affichés lors de l'exécution du programme, comme vu au chapitre 3.1. On peut supposer que cette valeur a été écrite à l'adresse `0x1100` par l'appel système numéro 3 qui a également été appelée avec la même adresse en paramètre.

3.3.7 Analyse de la fonction `sub_f8`

Le code de la fonction `sub_f8` est présenté ci-dessous.

```
sub_f8:
xor r2, r2, r2    ; loc_f8 : 0x32 0x22
movh r3, 0x0    ; loc_fa : 0x23 0x00
movl r3, 0x1    ; loc_fc : 0x13 0x01

loc_fe:
xor r4, r4, r4    ; loc_fe : 0x34 0x44
ldrb r4, [r0 + r2]    ; loc_100 : 0xe4 0x02
and r4, r4, r4    ; loc_102 : 0x54 0x44
jmp Z, loc_10e    ; loc_104 : 0xa0 0x08
sub r4, r4, r1    ; loc_106 : 0x74 0x41
jmp Z, loc_110    ; loc_108 : 0xa0 0x06
add r0, r0, r3    ; loc_10a : 0x60 0x03
jmp loc_fe    ; loc_10c : 0xb3 0xf0

loc_10e:
xor r0, r0, r0    ; loc_10e : 0x30 0x00

loc_110:
ret 0, 0xf    ; loc_110 : 0xd0 0x0f
```

Le code C présenté ci-dessous est équivalent au code de la fonction `sub_f8` :

```
uint16_t sub_f8(uint16_t r0, uint16_t r1) {
    uint16_t r4;

loc_fe:
    r4 = mem[r0];
    if (r4 == 0) {
        /* fin de chaîne */
        return 0;
    }
    if (r4 == r1) {
        return r0;
    }
    r0++;
    goto loc_fe;
}
```

Cette fonction est relativement simple et va chercher depuis l'adresse spécifiée par le registre `r0` la position du premier octet dont la valeur est égale à celle du registre `r1`. Dans notre cas, cette fonction est appelée avec les paramètres `0x1b2` et `0x24`, comme présenté au paragraphe 3.3.1. La valeur `0x1b2` correspond à l'adresse des données déchiffrées par RC4 et `0x24` est la valeur hexadécimale du caractère `$`.

L'appel de la fonction va de retourner la position du premier caractère `$` au sein de la chaine `Execution completed in $$$$$ CPU cycles.`

3.3.8 Analyse de la fonction `sub_148`

Le code assembleur de la fonction `sub_148` est présenté ci-dessous :

```
sub_148:
movh r4, 0x27    ; loc_148 : 0x24 0x27
movl r4, 0x10    ; loc_14a : 0x14 0x10
movh r5, 0x0     ; loc_14c : 0x25 0x00
movl r5, 0xa     ; loc_14e : 0x15 0x0a
xor r6, r6, r6    ; loc_150 : 0x36 0x66
movh r7, 0x0     ; loc_152 : 0x27 0x00
movl r7, 0x1     ; loc_154 : 0x17 0x01
sub r0, r0, r7    ; loc_156 : 0x70 0x07

loc_158:
add r0, r0, r7    ; loc_158 : 0x60 0x07
idiv r2, r1, r4    ; loc_15a : 0x92 0x14
imul r3, r2, r4    ; loc_15c : 0x83 0x24
sub r1, r1, r3    ; loc_15e : 0x71 0x13
idiv r4, r4, r5    ; loc_160 : 0x94 0x45
movh r8, 0x0     ; loc_162 : 0x28 0x00
movl r8, 0x20    ; loc_164 : 0x18 0x20
xor r3, r3, r3    ; loc_166 : 0x33 0x33
strb r8, [r0 + r3] ; loc_168 : 0xf8 0x03
or r6, r6, r2    ; loc_16a : 0x46 0x62
jmp Z, loc_158    ; loc_16c : 0xa3 0xea
movh r8, 0x0     ; loc_16e : 0x28 0x00
movl r8, 0x30    ; loc_170 : 0x18 0x30
add r8, r8, r2    ; loc_172 : 0x68 0x82
strb r8, [r0 + r3] ; loc_174 : 0xf8 0x03
and r4, r4, r4    ; loc_176 : 0x54 0x44
jmp NZ, loc_158    ; loc_178 : 0xa7 0xde
ret 0, 0xf    ; loc_17a : 0xd0 0x0f
```

Le code C ci-dessous est équivalent à la fonction `sub_148`.

```
void sub_148(uint16_t r0, uint16_t r1) {
    uint16_t r2, r3, r4;

    r4 = 10000;
    r0--;
loc_158:
    r0++;
    r2 = r1 / r4;
    r3 = r2 * r4;
    r1 = r1 - r3;
    r4 = r4 / 10;

    mem[r0] = 0x20;
    if (r2 == 0) {
        goto loc_158;
    }
```

```
    mem[r0] = r2 + 0x30;
    if (r4 != 0)
        goto loc_158;

}
```

Cette fonction va diviser le registre `r1` successivement par 10000, 1000, 100 et 10 et stocker le résultat à l'adresse indiquée par le registre `r0`. Dans notre cas, cette fonction est appelée avec :

— comme premier paramètre le retour de la fonction `sub_f8` qui retourne la position du premier caractère `$` au sein de la chaîne déchiffrée ;
— comme second paramètre la valeur du nombre de cycles CPU obtenu via l'appel système numéro 3.

C'est donc cette fonction qui va être responsable de remplacer dans la chaîne `Execution completed in $$$$$ CPU cycles.` les caractères `$` par le nombre de cycles CPU obtenus avec l'appel système 3.

3.3.9 Conclusion

Maintenant que le rôle de chaque fonction a été déterminé, il est possible de réécrire le code de la fonction principale en renommant les fonctions, comme présenté ci-dessous.

```
#define KEY_ADDR 0x17c
#define KEY_LEN 16
#define CIPHER_ADDR 0x1b2
#define CIPHER_LEN 0x29
#define FIRMWARE_STARTING_ADDR 0x18c
#define FIRMWARE_STARTING_LEN
#define HALTING_ADDR 0x1a8
#define HALTING_LEN 0x9

void main(void) {
    uint16_t r10;

    write_stdout(FIRMWARE_STARTING_ADDR, FIRMWARE_STARTING_LEN);
    RC4_init(0x1000, KEY_ADDR, KEY_LEN);
    RC4_decrypt(0x1000, CIPHER_ADDR, CIPHER_LEN);
    syscall_3(0x1100);
    r10 = read_word(0x1100);
    r0 = find_pos(CIPHER_ADDR, '$');
    do_subst(r0, r10);
    write_stdout(CIPHER_ADDR, CIPHER_LEN);
    write_stdout(HALTING_ADDR, HALTING_LEN);
    exit(EXIT_SUCCESS);
}
```

Le programme `fw.hex` a été entièrement porté en C (fichier `decomp.c`), disponible à l'annexe A.3.

Maintenant que le fonctionnement du programme a été compris, il reste à trouver une solution pour accéder au contenu de la « Secret area ». Une première piste est d'utiliser l'appel système 2 pour afficher le contenu de cette zone sur la sortie standard. Le fichier `upload.py` nous indique que la zone commence à l'adresse `0xf000` et termine à `0xfbff`. La taille de cette zone est donc `0xfbff - 0xf000 = 0xbff`.

Le programme ci-dessous tente d'accéder au contenu de la zone secrète :

```
movh r0, 0xf0
movl r0, 0x00
movh r1, 0x0b
movl r1, 0xff
syscall 2
```

```
syscall 1
```

Malheureusement, le résultat n'est pas celui escompté :

```
$ ./loader.rb -i dump-secret-area.asm -a -e -s
System reset.
[ERROR] Printing at unallowed address. CPU halted.
```

Il n'est donc pas possible d'accéder au contenu de la zone en utilisant directement l'appel système 2. Par contre, en effectuant des tests, on se rend compte qu'il est possible d'accéder au contenu de la zone kernel avec le programme ci-dessous :

```
movh r0, 0xfd
movl r0, 0x00
movh r1, 0x02
movl r1, 0xff
syscall 2
syscall 1
```

Pour obtenir les données de la section kernel, il faut rediriger le résultat de l'exécution vers un fichier tout en supprimant la chaîne « System reset. » qui est systématiquement affichée lors du démarrage d'un programme.

```
$ ./loader.rb -i dump-secret-area.asm -a -e -s > kernel.rom
$ dd if=kernel.rom of=kernel.stripped bs=1 skip=14
$ hexdump -C kernel.stripped
00000000  50 00 a0 6c 21 00 11 03  72 10 a8 12 22 00 12 02  |P..l!...r..."...|
00000010  81 02 71 12 20 f0 10 00  60 01 c0 94 d0 00 21 00  |..q. ...`.....!.|
00000020  11 2b 20 fe 10 5a c0 be  30 00 21 fc 11 10 22 00  |.+ ..Z..0.!...".|
00000030  12 01 f2 10 b3 f2 20 fc  10 22 c0 74 55 00 20 fc  |...... .."tU. .|
00000040  10 20 c0 6c 51 55 c0 9e  d8 00 20 fc 10 20 c0 60  |. .lQU.... .. .`|
00000050  26 fc 16 12 21 00 11 01  34 44 e5 61 e2 64 e3 64  |&...!...4D.a.d.d|
00000060  73 32 a7 f6 23 01 13 00  82 23 41 25 c0 56 d8 00  |s2..#....#A%.V..|
00000070  21 00 11 0e 20 fe 10 86  c0 6c 24 00 14 02 21 fd  |!... ....l$...!.|
[...]
00000120  10 26 c3 c2 b3 02 5b 45  52 52 4f 52 5d 20 50 72  |.&....[ERROR] Pr|
00000130  69 6e 74 69 6e 67 20 61  74 20 75 6e 61 6c 6c 6f  |inting at unallo|
00000140  77 65 64 20 61 64 64 72  65 73 73 2e 20 43 50 55  |wed address. CPU|
00000150  20 68 61 6c 74 65 64 2e  0a 00 5b 45 52 52 4f 52  | halted...[ERROR|
00000160  5d 20 55 6e 64 65 66 69  6e 65 64 20 73 79 73 74  |] Undefined syst|
00000170  65 6d 20 63 61 6c 6c 2e  20 43 50 55 20 68 61 6c  |em call. CPU hal|
00000180  74 65 64 2e 0a 00 53 79  73 74 65 6d 20 72 65 73  |ted...System res|
00000190  65 74 2e 0a 00 00 00 00  00 00 00 00 00 00 00 00  |et..............|
000001a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000002f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 0a  |................|
00000300
```

On retrouve dans les données obtenues certaines chaînes de caractères rencontrées précédemment.

3.4 Désassemblage du kernel

Il est possible de désassembler le fichier `kernel.stripped` obtenu précédemment.

```
$ ./loader.rb -i kernel.stripped -x kernel.asm
[+] disassembling
[+] loading binary data from kernel.stripped
```

L'analyse du fichier `kernel.asm` fait l'objet des paragraphes suivants.

3.4.1 Analyse du point d'entrée

De façon analogue à l'analyse précédemment sur le programme exécuté en mode utilisateur, on suppose que le kernel commence son exécution à l'adresse 0.

Le code correspondant est présenté ci-dessous :

```
and r0, r0, r0    ; loc_0 : 0x50 0x00
jmp Z, loc_70    ; loc_2 : 0xa0 0x6c
movh r1, 0x0    ; loc_4 : 0x21 0x00
movl r1, 0x3    ; loc_6 : 0x11 0x03
sub r2, r1, r0    ; loc_8 : 0x72 0x10
jmp S, loc_1e    ; loc_a : 0xa8 0x12
movh r2, 0x0    ; loc_c : 0x22 0x00
movl r2, 0x2    ; loc_e : 0x12 0x02
imul r1, r0, r2    ; loc_10 : 0x81 0x02
sub r1, r1, r2    ; loc_12 : 0x71 0x12
movh r0, 0xf0    ; loc_14 : 0x20 0xf0
movl r0, 0x0    ; loc_16 : 0x10 0x00
add r0, r0, r1    ; loc_18 : 0x60 0x01
call sub_b0    ; loc_1a : 0xc0 0x94
ret 0, 0x0    ; loc_1c : 0xd0 0x00

loc_1e:
movh r1, 0x0    ; loc_1e : 0x21 0x00
movl r1, 0x2b    ; loc_20 : 0x11 0x2b
movh r0, 0xfe    ; loc_22 : 0x20 0xfe
movl r0, 0x5a    ; loc_24 : 0x10 0x5a
call sub_e6    ; loc_26 : 0xc0 0xbe

[...]
loc_70:
movh r1, 0x0    ; loc_70 : 0x21 0x00
movl r1, 0xe    ; loc_72 : 0x11 0x0e
movh r0, 0xfe    ; loc_74 : 0x20 0xfe
movl r0, 0x86    ; loc_76 : 0x10 0x86
call sub_e6    ; loc_78 : 0xc0 0x6c
movh r4, 0x0    ; loc_7a : 0x24 0x00
movl r4, 0x2    ; loc_7c : 0x14 0x02
movh r1, 0xfd    ; loc_7e : 0x21 0xfd
movl r1, 0x28    ; loc_80 : 0x11 0x28
movh r0, 0xf0    ; loc_82 : 0x20 0xf0
movl r0, 0x0    ; loc_84 : 0x10 0x00
call sub_c4    ; loc_86 : 0xc0 0x3c
add r0, r0, r4    ; loc_88 : 0x60 0x04
movh r1, 0xfd    ; loc_8a : 0x21 0xfd
movl r1, 0x36    ; loc_8c : 0x11 0x36
call sub_c4    ; loc_8e : 0xc0 0x34
add r0, r0, r4    ; loc_90 : 0x60 0x04
movh r1, 0xfd    ; loc_92 : 0x21 0xfd
movl r1, 0x4a    ; loc_94 : 0x11 0x4a
call sub_c4    ; loc_96 : 0xc0 0x2c
movh r0, 0xfc    ; loc_98 : 0x20 0xfc
movl r0, 0x20    ; loc_9a : 0x10 0x20
xor r1, r1, r1    ; loc_9c : 0x31 0x11
movh r2, 0x0    ; loc_9e : 0x22 0x00
movl r2, 0x36    ; loc_a0 : 0x12 0x36
call sub_d6    ; loc_a2 : 0xc0 0x32
movh r0, 0xfc    ; loc_a4 : 0x20 0xfc
movl r0, 0x3a    ; loc_a6 : 0x10 0x3a
movh r1, 0xef    ; loc_a8 : 0x21 0xef
movl r1, 0xfe    ; loc_aa : 0x11 0xfe
call sub_c4    ; loc_ac : 0xc0 0x16
ret 8, 0x0    ; loc_ae : 0xd8 0x00
```

Ce code peut être traduit en C de la façon suivante :

```
void start(uint16_t r0) {
  if (r0 == 0) {
      sub_e6(0xfe86, 0xe);
      sub_c4(0xf000, 0xfd28);
      sub_c4(0xf002, 0xfd36);
      sub_c4(0xf004, 0xfd4a);
      sub_d6(0xfc20, 0, 0x36);
      sub_c4(0xfc3a, 0xeffe);
      /* ret 8, 0x0 */
  } else {
      if (r0 > 3) {
            sub_e6(0xfe5a, 0x2b);
      } else {
            sub_b0(0xf000 + 2 * (r0 - 1));
            /* ret 0, 0x0 */
      }
  }
}
```

Par analogie avec la gestion des appels systèmes étudiée précédemment, on peut supposer que l'instruction `ret 8, 0x0` effectue un retour en espace utilisateur. Par contre, l'instruction `ret 0, 0x0` permet de rester dans le mode courant, c'est-à-dire kernel, tout en effectuant un branchement sur l'adresse contenue dans le registre `r0`.

L'analyse se poursuit en étudiant les fonctions `sub_e6`, `sub_c4`, `sub_d6` et `sub_b0`.

3.4.2 Analyse de la fonction `sub_e6`

Le code de la fonction `sub_e6` est présenté ci-dessous :

```
loc_28:
xor r0, r0, r0    ; loc_28 : 0x30 0x00
movh r1, 0xfc     ; loc_2a : 0x21 0xfc
movl r1, 0x10     ; loc_2c : 0x11 0x10
movh r2, 0x0      ; loc_2e : 0x22 0x00
movl r2, 0x1      ; loc_30 : 0x12 0x01
strb r2, [r1 + r0] ; loc_32 : 0xf2 0x10
jmp loc_28     ; loc_34 : 0xb3 0xf2

[...]

sub_e6:
and r14, r0, r0    ; loc_e6 : 0x5e 0x00
movh r13, 0xfc     ; loc_e8 : 0x2d 0xfc
movl r13, 0x0      ; loc_ea : 0x1d 0x00
movh r12, 0xf0     ; loc_ec : 0x2c 0xf0
movl r12, 0x0      ; loc_ee : 0x1c 0x00
xor r8, r8, r8     ; loc_f0 : 0x38 0x88
and r9, r8, r8     ; loc_f2 : 0x59 0x88
movh r10, 0x0      ; loc_f4 : 0x2a 0x00
movl r10, 0x1      ; loc_f6 : 0x1a 0x01
xor r11, r11, r11     ; loc_f8 : 0x3b 0xbb

loc_fa:
and r1, r1, r1     ; loc_fa : 0x51 0x11
jmp Z, loc_118     ; loc_fc : 0xa0 0x1a
add r9, r14, r8     ; loc_fe : 0x69 0xe8
sub r9, r9, r12     ; loc_100 : 0x79 0x9c
jmp S, loc_10c     ; loc_102 : 0xa8 0x08
add r9, r14, r8     ; loc_104 : 0x69 0xe8
sub r9, r9, r13     ; loc_106 : 0x79 0x9d
```

```
jmp NS, loc_10c    ; loc_108 : 0xac 0x02
jmp loc_11a    ; loc_10a : 0xb0 0x0e

loc_10c:
xor r9, r9, r9    ; loc_10c : 0x39 0x99
ldrb r9, [r14 + r8]    ; loc_10e : 0xe9 0xe8
strb r9, [r13 + r11]    ; loc_110 : 0xf9 0xdb
add r8, r8, r10    ; loc_112 : 0x68 0x8a
sub r1, r1, r10    ; loc_114 : 0x71 0x1a
jmp loc_fa    ; loc_116 : 0xb3 0xe2

loc_118:
ret 0, 0xf    ; loc_118 : 0xd0 0x0f

loc_11a:
movh r1, 0x0    ; loc_11a : 0x21 0x00
movl r1, 0x33    ; loc_11c : 0x11 0x33
movh r0, 0xfe    ; loc_11e : 0x20 0xfe
movl r0, 0x26    ; loc_120 : 0x10 0x26
call sub_e6    ; loc_122 : 0xc3 0xc2
jmp loc_28    ; loc_124 : 0xb3 0x02
```

Ce code peut être traduit par le code C ci-dessous :

```c
void sub_e6(uint16_t r0, uint16_t r1) {
    uint16_t r8;

    r8 = 0;

loc_fa:
    if (r1 == 0) {
        return;
    }

    if (r0 + r8 < 0xf000) {
        /* loc_10c */
        mem[0xfc00] = mem[r0 + r8];
        r8++; r1--;
        goto loc_fa;
    }
    if (r0 + r8 >= 0xfc00) {
        /* loc_10c */
        mem[0xfc00] = mem[r0 + r8];
        r8++; r1--;
        goto loc_fa;
    }

    /* loc_11a */
    sub_e6(0xfe26, 0x33);

    /* loc_28 */
    mem[0xfc10] = 1;
}
```

La fonction lit un octet à l'adresse spécifiée par le registre `r0` puis l'écrit à l'emplacement en mémoire `0xfc00`. Cette opération est répétée autant de fois que la valeur stockée dans le registre `r1`, à condition que l'adresse lue soit inférieure à `0xf000` ou supérieure à `0xfc00`, c'est-à-dire en dehors de la zone secrète. Sinon, la fonction est appelée récursivement avec les paramètres `0xfe26` et `0x33`.

La valeur `0xfe26` peut représenter une adresse mémoire dans l'espace kernel. Il est possible de retrouver les données correspondantes grâce à la commande suivante :

```
$ dd if=kernel.stripped bs=1 skip=$((0xfe26-0xfd00)) count=$((0x33)) 2>/dev/null
[ERROR] Printing at unallowed address. CPU halted.
```

Cette fonction sert donc à afficher les données à l'adresse contenue dans le registre `r0` et de longueur correspondante à la valeur dans le registre `r1`.

3.4.3 Analyse de la fonction `sub_c4`

Le code de la fonction `sub_c4` est présenté ci-dessous :

```
sub_c4:
movh r2, 0x0    ; loc_c4 : 0x22 0x00
movl r2, 0x1    ; loc_c6 : 0x12 0x01
movh r3, 0x1    ; loc_c8 : 0x23 0x01
movl r3, 0x0    ; loc_ca : 0x13 0x00
strb r1, [r0 + r2]    ; loc_cc : 0xf1 0x02
sub r2, r2, r2    ; loc_ce : 0x72 0x22
idiv r1, r1, r3    ; loc_d0 : 0x91 0x13
strb r1, [r0 + r2]    ; loc_d2 : 0xf1 0x02
ret 0, 0xf    ; loc_d4 : 0xd0 0x0f
```

Cette fonction peut être traduite par le code C ci-dessous :

```c
void sub_c4(uint16_t r0, uint16_t r1) {
    mem[r0 + 1] = r1 & 0xff;
    mem[r0] =  r1 << 8;
}
```

La fonction `sub_c4` va simplement stocker la valeur contenue dans le registre `r1` à l'adresse spécifiée dans le registre `r0`.

3.4.4 Analyse de la fonction `sub_d6`

Le code de la fonction `sub_d6` est présenté ci-dessous :

```
sub_d6:
movh r3, 0x0    ; loc_d6 : 0x23 0x00
movl r3, 0x1    ; loc_d8 : 0x13 0x01
and r2, r2, r2    ; loc_da : 0x52 0x22
jmp Z, loc_e4    ; loc_dc : 0xa0 0x06
sub r2, r2, r3    ; loc_de : 0x72 0x23
strb r1, [r0 + r2] ; loc_e0 : 0xf1 0x02
jmp loc_d6    ; loc_e2 : 0xb3 0xf2

loc_e4:
ret 0, 0xf    ; loc_e4 : 0xd0 0x0f
```

Cette fonction est très simple et va simplement écrire à l'adresse spécifiée par le registre `r0` la valeur stockée dans `r1`, et ce autant de fois que la valeur du registre `r0`.

3.4.5 Analyse de la fonction `sub_b0`

Le code de la fonction `sub_b0` est présenté ci-dessous :

```
sub_b0:
movh r1, 0x0    ; loc_b0 : 0x21 0x00
movl r1, 0x1    ; loc_b2 : 0x11 0x01
```

```
movh r2, 0x1    ; loc_b4 : 0x22 0x01
movl r2, 0x0    ; loc_b6 : 0x12 0x00
ldrb r3, [r0 + r1]    ; loc_b8 : 0xe3 0x01
sub r1, r1, r1    ; loc_ba : 0x71 0x11
ldrb r4, [r0 + r1]    ; loc_bc : 0xe4 0x01
imul r4, r4, r2    ; loc_be : 0x84 0x42
or r0, r3, r4    ; loc_c0 : 0x40 0x34
ret 0, 0xf    ; loc_c2 : 0xd0 0x0f
```

Cette fonction a pour rôle de lire deux octets à l'adresse spécifiée par le registre `r0` et à stocker le résultat dans ce même registre.

Maintenant que les fonctions principales ont été analysées, il est possible de mettre à jour le code de la fonction principale, ainsi que la nature des chaînes de caractères aux adresses `0xfe86` et `0xfe5a`.

```
$ dd if=kernel.stripped bs=1 skip=$((0xfe86-0xfd00)) count=$((0xe)) 2>/dev/null
System reset.
$ dd if=kernel.stripped bs=1 skip=$((0xfe5a-0xfd00)) count=$((0x2b)) 2>/dev/null
[ERROR] Undefined system call. CPU halted.
```

Le code devient donc :

```
#define SYSTEM_RESET_ADDR 0xfe86
#define SYSTEM_RESET_LEN 0xe
#define UNDEF_SYSCALL_ADDR 0xfe5a
#define UNDEF_SYSCALL_LEN 0x2b

void start(uint16_t r0) {
  if (r0 == 0) {
      write_string(SYSTEM_RESET_ADDR, SYSTEM_RESET_LEN);
      store_word(0xf000, 0xfd28);
      store_word(0xf002, 0xfd36);
      store_word(0xf004, 0xfd4a);
      memset(0xfc20, 0, 0x36);
      store_word(0xfc3a, 0xeffe);
      /* ret 8, 0x0 */
  } else {
      if (r0 > 3) {
            write_string(UNDEF_SYSCALL_ADDR, UNDEF_SYSCALL_LEN);
      } else {
            r0 = load_word(0xf000 + 2 * (r0 - 1));
            /* ret 0, 0x0 */
      }
  }
}
```

Le code analysé est donc responsable de la gestion des appels systèmes. Si le numéro de l'appel système (stocké dans le registre `r0`), alors le programme procède à l'initialisation du microcontrôleur. Si le numéro de l'appel système est supérieur à 3, un message d'erreur est affiché. Enfin, si celui-ci est compris entre 1 et 3, le registre `r0` est mis à jour avec une valeur lue dans une table commençant à l'adresse `0xf000` puis un branchement est effectué sur la valeur lue.

La routine d'initialisation va donc associer à chaque numéro d'appel système une fonction de traitement. Le code de ces fonctions n'étant pas atteignable directement, il est alors nécessaire d'effectuer un nouveau désassemblage en spécifiant des points d'entrées supplémentaires correspondant aux fonctions de traitement des appels systèmes.

```
$ ./loader.rb -i kernel.stripped -x kernel.asm --entrypoints 0,0x28,0x36,0x4a
[+] disassembling
[+] loading binary data from kernel.stripped
```

Le fichier obtenu est disponible à l'annexe A.3.

Maintenant que ces fonctions sont correctement désassemblées, il s'agit mainteneant de poursuivre leur analyse.

3.4.6 Analyse de l'appel système 1 (loc_28)

Le code de traitement de l'appel système 1 (`exit`) est présenté ci-dessous :

```
loc_28:
xor r0, r0, r0    ; loc_28 : 0x30 0x00
movh r1, 0xfc     ; loc_2a : 0x21 0xfc
movl r1, 0x10     ; loc_2c : 0x11 0x10
movh r2, 0x0     ; loc_2e : 0x22 0x00
movl r2, 0x1     ; loc_30 : 0x12 0x01
strb r2, [r1 + r0]     ; loc_32 : 0xf2 0x10
jmp loc_28     ; loc_34 : 0xb3 0xf2
```

Le programme écrit simplement une valeur à l'adresse `0xfc10` pour provoquer un arrêt.

3.4.7 Analyse de l'appel système 2 (loc_36)

Le code de traitement de l'appel système 2 (`write_stdout`) est présenté ci-dessous :

```
movh r0, 0xfc     ; loc_36 : 0x20 0xfc
movl r0, 0x22     ; loc_38 : 0x10 0x22
call sub_b0     ; loc_3a : 0xc0 0x74
and r5, r0, r0     ; loc_3c : 0x55 0x00
movh r0, 0xfc     ; loc_3e : 0x20 0xfc
movl r0, 0x20     ; loc_40 : 0x10 0x20
call sub_b0     ; loc_42 : 0xc0 0x6c
and r1, r5, r5     ; loc_44 : 0x51 0x55
call sub_e6     ; loc_46 : 0xc0 0x9e
ret 8, 0x0     ; loc_48 : 0xd8 0x00
```

Le code C ci-dessous est équivalent :

```c
void sub_36(void) {
    uint16_t r0, r1;

    r1 = load_word(0xfc22);
    r0 = load_word(0xfc20);
    write_string(r0, r1);
    /* ret 8, 0x0 */
}
```

On peut en déduire que les adresses `0xfc20` et `0xfc22` correspondent aux emplacements de sauvegarde des registres `r0` et `r1` en espace utilisateur avant l'appel système.

3.4.8 Analyse de l'appel système 3 (loc_4a)

Le code de traitement de l'appel système 3 est présenté ci-dessous :

```
movh r0, 0xfc     ; loc_4a : 0x20 0xfc
movl r0, 0x20     ; loc_4c : 0x10 0x20
call sub_b0     ; loc_4e : 0xc0 0x60
movh r6, 0xfc     ; loc_50 : 0x26 0xfc
movl r6, 0x12     ; loc_52 : 0x16 0x12
movh r1, 0x0     ; loc_54 : 0x21 0x00
movl r1, 0x1     ; loc_56 : 0x11 0x01
```

```
xor r4, r4, r4    ; loc_58 : 0x34 0x44

loc_5a:
ldrb r5, [r6 + r1]    ; loc_5a : 0xe5 0x61
ldrb r2, [r6 + r4]    ; loc_5c : 0xe2 0x64
ldrb r3, [r6 + r4]    ; loc_5e : 0xe3 0x64
sub r3, r3, r2    ; loc_60 : 0x73 0x32
jmp NZ, loc_5a    ; loc_62 : 0xa7 0xf6
movh r3, 0x1    ; loc_64 : 0x23 0x01
movl r3, 0x0    ; loc_66 : 0x13 0x00
imul r2, r2, r3    ; loc_68 : 0x82 0x23
or r1, r2, r5    ; loc_6a : 0x41 0x25
call sub_c4    ; loc_6c : 0xc0 0x56
ret 8, 0x0    ; loc_6e : 0xd8 0x00
```

Le code C ci-dessous correspond au traitement de l'appel système 3 :

```
void sub_4a(void) {
    uint16_t r0, r1, r2, r3, r5;
    r0 = load_word(0xfc20);

loc_5a:
    r5 = mem[0xfc13];
    r2 = mem[0xfc12];
    r3 = mem[0xfc12];

    if (r3 != r2)
        goto loc_5a;
    r2 = r2 * 0x1000;
    r1 = r2 | r5;
    store_word(r0, r1);
    /* ret 8, 0x0 */
}
```

Cet appel système effectue des lectures successives à l'adresse `0xfc12` et sort de la boucle dès que deux lectures consécutives donnent le même résultat. Une fois que cette condition est remplie, le mot de 16 bits lu à l'adresse `0xfc12` est écrit à l'adresse dans le registre `r0` qui correspond au paramètre de l'appel système.

L'appel système 3 est particulièrement intéressant car il constitue une primitive d'écriture à une adresse arbitraire, et ce depuis l'espace utilisateur. Malheureusement, la valeur écrite, qui correspond au nombre de cycles CPU exécutés, n'est pas directement sous le contrôle de l'utilisateur.

3.5 Prise de contrôle du kernel et mise au point de l'exploit

Une première piste est d'utiliser l'appel système 3 pour réécrire la table des appels systèmes et rediriger le flot d'exécution du kernel.

Un test est effectué avec le programme ci-dessous :

```
movh r0, 0xf0
movl r0, 0x04
syscall 3
syscall 3
```

L'exécution du programme donne le résultat suivant :

```
$ ./loader.rb -i exploit1.asm -a -e -s
System reset.
-- Exception occurred at 07C0: Invalid instruction.
   r0:07C0     r1:0000     r2:0100     r3:00C0
```

```
   r4:0700     r5:0000    r6:0000    r7:0000
   r8:0000     r9:0000   r10:0000   r11:0000
  r12:0000    r13:EFFE   r14:0000   r15:FD1C
   pc:07C0 fault_addr:0000 [S:0 Z:0] Mode:kernel
CLOSING: Invalid instruction.
```

Le programme a branché sur l'adresse `0x7c0` mais qui constitue une adresse invalide. Cette valeur correspond au nombre de cycles CPU retournés par l'appel système 3. On remarque cependant que l'exécution s'effectue bien en mode kernel.

Pour éviter de déclencher cette exception, on peut imaginer rajouter des instructions dans notre programme pour que l'instruction à l'adresse `0x7c0` devienne valide. `0x7c0` correspond à 1984 : on décide alors d'ajouter 1000 instructions `xor r0, r0, r0` (chaque instruction étant codée sur 2 octets). Pour sortir proprement, l'appel système 1 est déclenché.

Le programme devient alors :

```
movh r0, 0xf0
movl r0, 0x04
syscall 3
syscall 3
xor r0, r0, r0
; instruction xor r0, r0, r0 répétée 1000 fois
syscall 1
```

L'exécution donne alors le résultat suivant :

```
$ ./loader.rb -i exploit1.asm -a -e -s
System reset.
```

Cette fois, aucune exception n'est déclenchée. Il reste alors à intercaller une boucle pour afficher le contenu de la zone secrète.

L'exploit final est alors :

```
movh r0, 0xf0
movl r0, 0x04
syscall 3
syscall 3
xor r0, r0, r0
; instruction xor r0, r0, r0 répétée 1000 fois

movh r0, 0xf0
movl r0, 0x06
movh r1, 0x00
movl r1, 0x00
movh r3, 0xfc
movl r3, 0x00
movh r4, 0x0b
movl r4, 0xff
movh r5, 0x00
movl r5, 0x01
movh r6, 0x00
movl r6, 0x00

loop_start:
ldrb r2, [r0 + r1]
strb r2, [r3 + r6]
add r1, r1, r5
sub r4, r4, r5
jmp NZ, loop_start
```

Le résultat de l'exploit est présenté à la figure 3.2.

Figure 3.2 – Résultat de l'exploit

L'adresse email recherchée est donc : `66a65dc050ec0c84cf1dd5b3bbb75c8c@challenge.sstic.org`.

Le code source complet de l'exploit est disponible à l'annexe A.3.

Chapitre 4

Conclusion

4.1 Synthèse

L'ensemble des développements réalisés dans le cadre de ce challenge seront disponibles dès la fin de l'édition 2014 du SSTIC à l'adresse https://github.com/nieluj/sstic2014.

4.2 Remerciements

Comme chaque année, le challenge du SSTIC s'est montré à la hauteur des attentes des participants. J'ai personnellement apprécié le fait de devoir analyser un binaire sur une architecture ARM64, ce que je n'avais jamais fait auparavant. Je tiens donc à remercier particulièrement le concepteur du challenge ainsi que l'ensemble du comité d'organisation du SSTIC.

Annexe A

Annexes

A.1 Annexe : étude de la trace USB

Enregistrer `usbmon.txt` :

Enregistrer `protocol.txt` :

Enregistrer `SYNC.TXT` :

Enregistrer `parse-usbmon.rb` :

A.2 Annexe : analyse de `badbios.bin`

Enregistrer `badbios.objdump` :

Résultat de la rétro-conception de la fonction `sub_10304` :

```
int sub_10304(char *src, char *dst, uint32_t slen, uint32_t dlen) {
        char *p;
        uint8_t b;
        uint64_t off1, off2, off3;
        uint64_t a1[8] = { 4, 1, 2, 1, 4, 4, 4, 4 };
        uint64_t a2[8] = { 0, 0, 0, 0xFFFFFFFFFFFFFFFF, 0, 1, 2, 3 };
        int i, src_idx = 0, pdst_idx, dst_idx = 0;

        do {
            pdst_idx = dst_idx;

            b = src[src_idx++];

            off1 = b >> 4;
            off2 = b & 0xf;

            if ((off1 == 0xf) && (src_idx < slen)) {
                do {
                    b = src[src_idx++];
                    off1 += b;
                } while (src_idx == slen || b == 0xff);
            }

            if (dst_idx + off1 > dlen - 12)
```

```
            break;

        if (src_idx + off1 > slen - 8)
            break;

        do {
            memcpy(dst + dst_idx, src + src_idx, 8);
            dst_idx += 8; src_idx += 8;
        } while (dst_idx < (pdst_idx + off1));

        src_idx += off1 - (dst_idx - pdst_idx);

        off3 = *((uint16_t *) (src + src_idx));
        src_idx += 2;

        if (off2 == 0xf) {
            do {
                if (src_idx >= slen - 6)
                    break;
                b = src[src_idx++];
                off2 += b;
            } while (b == 0xff);
        }

        p = dst + pdst_idx + off1 - off3;
        if (off3 <= 7) {
            for (i = 0; i < 4; i ++)
                p[i + off3] = p[i];

            memcpy(p + off3 + 4, p + a1[off3], 4);
            p += a1[off3] - a2[off3];
        } else {
            memcpy(p + off3, p, 8);
            p += 8;
        }

        for (i = 0; i + 4 < off2 ; i += 8) {
            memcpy(dst + pdst_idx + off1 + 8 + i, p + i, 8);
        }

        dst_idx = pdst_idx + off1 + off2 + 4;
    } while (1);

    if (src_idx + off1 == slen) {
        if (off1 != 0) {
            for (i = 0; i < off1 + 1; i++) {
                dst[dst_idx + i] = src[src_idx + i];
            }
            dst_idx += off1;
        }
    }

    return dst_idx;
}
```

Enregistrer `unpack.c` :

Enregistrer `disassvm-rb` :

Enregistrer `badbios2.asm` :

A.3 Annexe : étude du microcontrôleur

Enregistrer `fw.hex` :

Enregistrer `loader.rb` :

Enregistrer `test-conds.rb` :

Enregistrer `fw.asm` :

Enregistrer `decomp.c` :

Enregistrer `kernel.asm` :

Enregistrer `kernel.c` :

Enregistrer `exploit1.asm` :