Solution Challenge SSTIC 2019

Jean Bernard Beuque
20 Mai 2019

Sommaire

1. Préambule 2
2. SPA attack 3
3. ALU component 7
4. Dwarf VM 9
   1. Programme safe_01 9
   2. Dwarf Virtual Machine 13
5. ARM TrustZone 19
   1. Programme safe_02 19
   2. module sstic.ko 21
   3. Boot loaders 23
   4. Security monitor 25
   5. Secure OS Virtual machine 31
      1. Mémoire protégée 31
      2. Machine virtuelle 34
   6. Byte code program 38
6. Android image 44
7. Annexes 46
   1. Find_key.py 46
   2. Secure_device 47
   3. dwarf_disass 53
   4. InvHashC 57
   5. Disass.py 66
   6. Decrypt.c 68

1. Préambule

L'objectif du challenge est de trouver une adresse email @challenge.sstic.org.

Le challenge est constitué d'une image du logiciel d'un « téléphone » pour un processeur ARMv8. Ce logiciel peut s'executer via l'émulateur qemu. On dispose également d'un ficher contenant l'enregistrement de la consommation éléctrique lors du démarrage du téléphone.

2. SPA attack

La première fois, qu'on démarre l'image du « téléphone » via qemu, il crée un fichier keystore.

Puis il nous demande d'entrer un exposant privé RSA.

```
$ ./strt
###########################################
#     virtual environment detected      #
#         QEMU 3.1+ is needed         #
###########################################
NOTICE:  Booting SSTIC ARM Trusted Firmware
KEYSTORE: keystore doesn't exist
ERROR:   KEYSTORE: Can't read keystore, reset keystore, try to boot again
```

```
$ ./strt

###########################################
#     virtual environment detected      #
#         QEMU 3.1+ is needed         #
###########################################
NOTICE:  Booting SSTIC ARM Trusted Firmware
KEYSTORE: AES Key is still encrypted, need decryption
KEYSTORE: Need RSA key to decrypt
KEYSTORE: RSA private exponent is not set, please set it in the keystore or enter hex value :
```

Comme on dispose de l'enregistrement de la consommation éléctrique, on pense à une attaque SPA (Simple Power Analysis) sur l'implémentation du RSA.

On écrit un scipt python pour afficher la courbe de la consommation éléctrique.

```
import matplotlib.pyplot as plt
import numpy as np

from scipy.fftpack import fft, ifft

data = np.load('arr_0.npy')

#y = fft(data)

st = 1480000
lg =   20000

st = 697000
lg =  20000



plt.plot(data[st:st + lg])
#plt.plot(y)
plt.show()
```

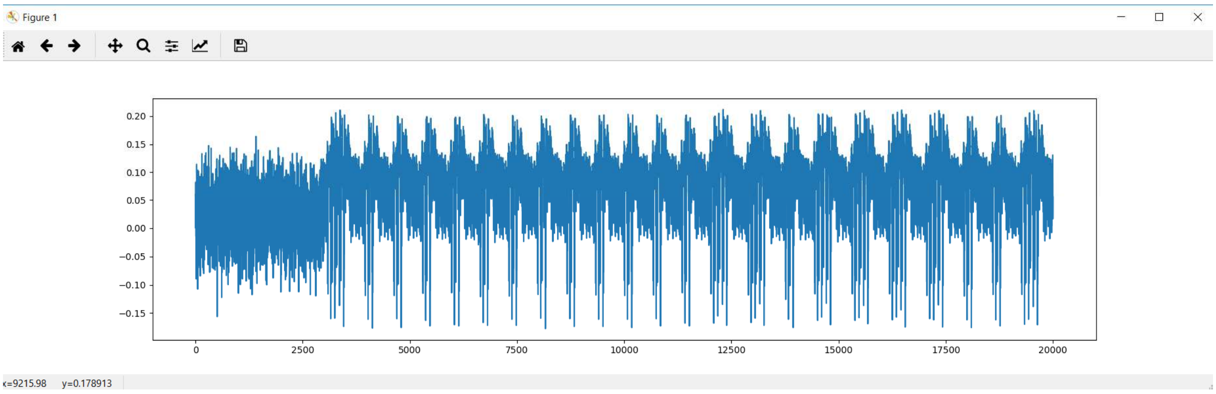


Figure 1 power consumption

La courbe de consummation éléctrique montre une répétion de motif correspondant aux bits de l'exposant privée RSA.

L'exponentiation modulaire du RSA est implementé par l'algorithme « square and multiply » dans la fonction FUN_00001348() dans rom.bin.

On peut voir ci-dessous la fonction décompilée par l'outil « Ghidra ».

```
void FUN_00001348(undefined8 uParm1,undefined8 uParm2,undefined8 uParm3,undefined8 uParm4)

{
  int iVar1;
  undefined auStack768 [256];
  uint local_200 [64];
  undefined auStack256 [256];

  FUN_000018bc(uParm4,1);
  FUN_00001bc8(auStack768,uParm1);
  FUN_00001bc8(local_200,uParm2);
  while( true ) {
    if ((local_200[0] & 1) != 0) {
      FUN_00002da0(&DAT_00005c55);
      FUN_00001a54(uParm4,auStack768,auStack256);
      FUN_00001d18(auStack256,uParm3,uParm4);
    }
    FUN_00001d34(local_200,auStack256,1);
    FUN_00001bc8(local_200,auStack256);
    iVar1 = FUN_00001ba0(local_200);
    if (iVar1 != 0) break;
    FUN_00002da0(&DAT_00005c57);
    FUN_00001a54(auStack768,auStack768,auStack256);
    FUN_00001d18(auStack256,uParm3,auStack768);
  }
  FUN_00002da0(0x571d);
  return;
```

Cet algorithme est vulnerable aux attaques SPA.

Ene effet l'algorithme effectue les opérations suivantes (pour calculer $y := x^e \pmod N$) :

Pour chaque bit de l'exposant,

si le bit vaut 0, on calcule $y := y^2 \pmod N$

si le bit vaut 1, on calcule $y := y^2 * x \pmod N$

La séquence d'opération est differente pour les bits à 0 et les bits à 1.

Comme la courbe de consommation éléctrique est fonction des instructions executées par le processeur, on peut lire directement les bits de l'exposant privée sur la courbe.

On écrit un script python pour extraire les bits de l'exposant privé à partir de la coube de consommation éléctrique.

Le script localise sur la courbe les positions des pattern correspondant à l'operation y^2 (qui est effectuée pour chaque bit). Pour cela, pour chaque position sur la courbe, on calcule la difference avec un pattern de reference. Si la difference est inferieur à un seuil (fixé empiriquement), on a localisé le pattern.

Ensuite l'intervalle entre les positions des patterns est utilisé pour distinguer les bits à 0 et les bits à 1. En effet, pour un bit à 1, l'intervalle est plus grand que pour un bit à 0 car une multiplication est effectuée en plus. Si l'intervalle est inférieur à un seuil (fixé empiriquement), le bit vaut 1 sinon il vaut 0.

Le script *find_key.py* est disponible en annexe. Il trouve la clef suivante de 1024 bits :

```
23D87CDF97BB95ABE6273C384190C765F552AB86F6DE30A8DB74435C95E6E3138F54AF689812D8F9359CF0F4D453A0C11EC68CE470216C
09E74C8947ADAF23E902415D61DDF2C0FFE459CBB40F7DE42BDB7CD14093100A570E8C29819765E2D8D276F86471B52AC29AA2CE2BB72
CD45006279E82BEC253AE9675FE45824F6001
```

```
############################################
#      virtual environment detected      #
#          QEMU 3.1+ is needed          #
############################################
NOTICE:  Booting SSTIC ARM Trusted Firmware
KEYSTORE: AES Key is still encrypted, need decryption
KEYSTORE: Need RSA key to decrypt
KEYSTORE: RSA private exponent is not set, please set it in the keystore or enter hex value :
23D87CDF97BB95ABE6273C384190C765F552AB86F6DE30A8DB74435C95E6E3138F54AF689812D8F9359CF0F4D453A0C11EC68CE470216C
09E74C8947ADAF23E902415D61DDF2C0FFE459CBB40F7DE42BDB7CD14093100A570E8C29819765E2D8D276F86471B52AC29AA2CE2BB72
CD45006279E82BEC253AE9675FE45824F6001
KEYSTORE: Key read:
HEXDUMP :
---------------------------------------------
23 d8 7c df 97 bb 95 ab e6 27 3c 38 41 90 c7 65
f5 52 ab 86 f6 de 30 a8 db 74 43 5c 95 e6 e3 13
8f 54 af 68 98 12 d8 f9 35 9c f0 f4 d4 53 a0 c1
1e c6 8c e4 70 21 6c 09 e7 4c 89 47 ad af 23 e9
02 41 5d 61 dd f2 c0 ff e4 59 cb b4 0f 7d e4 2b
db 7c d1 40 93 10 0a 57 0e 8c 29 81 97 65 e2 d8
d2 76 f8 64 71 b5 2a c2 9a a2 ce 2b b7 2c d4 50
```

```
06 27 9e 82 be c2 53 ae 96 75 fe 45 82 4f 60 01
-----------------------------------------------
KEYSTORE: Decrypting ...
+-----------+-+--+-+-+-+---+---+-----+-+--+---+---+-+-+-+-+-+-+-+--+--+-+-+----+-+-+---+-+--+---+--+-+-+--+--+-+-+---+--+---+-----+-+--+-+-+-+-+--+--+-----+--
+-+-+-+---+-+-+-+---+----+-+---------+--+----+--+--+-+---+-+--+---+-+-+--+-+---+-+--+-+-+--+--+----+-+-+---+-+--+----+--+--+--+-+---+--+-----+-+--+--+--+---+--
+--+-+--+-+----+-+-+----+---+-+-----+-+-+-+-+--+-+-+--+-+-+---+---+--+-+----+-+---+-+--+-+--+----+-+-+-+--+---+-+--+-+-+--+---+-+------+-+---+--+-----+-+----+--
+-+-+-----+-+-+--+--+---+--+---------+----+-+---+---+------+--+----+--+-+---+-+-+---+-+-+-+-+--+-+--+-+--+-+-+-+--+--+-----+---+-+-+-+--+-+-+-+-+--+-+-+-+------
+--+-+--+-+-+--+---+-+-+---+-+--+----+---+-+-+-+-+-+-+-+-+-+------+-+--+----+--+---+-+-+-+-+--+-+-+-+--+-+-+-----+-+--+--+-+-+--+--+------+---+------+---+--+-
+-+-+-+----+---+-+-+-+--+--+-+--+-+--+--+-+-+-+----+--+---+----+---+-+----+-+-+---+--+-+-+---+-+-+-+---+------+-+--+-+--+-----+------+-+-+----+---+-+-+---+-+----
+--+-+----+-+--+-+-+-+----+------+-+------+--+-+-+---+--+----+--+--+-+---+--+-+-+--+-+-+-+-----+-+-+-+---+-+-+---+-+--+--+-+---+---+-+-+-+-+----+-+--+-+--+---+-
-----+-+---+----+--+-+--+-+-+-+--+---+--+--+--+-+-+-+----+-+-+---+----+-+------+-+----+-+-+--+-+---+-+-+-+--+--+---+---+-+-+--+--+-+-----+----+--+-+-+--+-+--
+-+--+-+----+--+--+-----+-+----+-+-+-+--+-+--+-+-+-+-+-+-+-+-----+-+-+--+-+---+-+--+--+--+---+--+--+--+--+-+-+-+-+--+---+-+--+-+-+----+-+-----+---+-+------+---
--+-+-+-----+-+-+-+---+-+-+---+----+-+---+-+-+-+-+-+--+--+-+-+--+-+-+-+--+---+-+-+-+-+-+--+-+---+-+-+-+-+-----+-+--+-+-+-+-----+
```
Bravo, envoyez le flag SSTIC{a947d6980ccf7b87cb8d7c246} à l'adresse challenge2019@sstic.org pour valider votre avancée
```
NOTICE:  Loading image id=1
NOTICE:  BL1: Booting BL2
KEYSTORE: BL2 got key : key_type:0x02, key_len:0x20
HEXDUMP :
-----------------------------------------------
53 53 54 49 43 7b 61 39 34 37 64 36 39 38 30 63
63 66 37 62 38 37 63 62 38 64 37 63 32 34 36 7d
-----------------------------------------------
NOTICE:  Loading image id=3
KEYSTORE: bad keystore magic
KEYSTORE: BL2 got key : key_type:0x02, key_len:0x20
HEXDUMP :
-----------------------------------------------
44 44 44 44 44 44 44 44 44 44 44 44 44 44 44 44
44 44 44 44 44 44 44 44 44 44 44 44 44 44 44 44
-----------------------------------------------
NOTICE:  Loading image id=4
KEYSTORE: BL2 got key : key_type:0x02, key_len:0x20
HEXDUMP :
-----------------------------------------------
53 53 54 49 43 7b 61 39 34 37 64 36 39 38 30 63
63 66 37 62 38 37 63 62 38 64 37 63 32 34 36 7d
-----------------------------------------------
NOTICE:  Loading image id=5
NOTICE:  BL1: Booting BL31
ERROR:   Secure-OS not not available : need decryption key
UEFI firmware (version  built at 00:01:39 on Feb 25 2019)
EFI stub: Booting Linux Kernel...
EFI stub: Generating empty DTB
EFI stub: Exiting boot services and installing virtual address map...
[    0.000000] Booting Linux on physical CPU 0x0000000000 [0x411fd070]
```

3. ALU component

Cette épreuve est constituée d'un schéma de porte logiques « le secure element » et d'un programme python qui utilise le secure element.

Le programme python attend en entrée une séquence de 8 combinaisons de 4 boutons (i.e. 32 bits).

A partir de cette combinaison, il appel le secure element pour obtenir une clef de 64 bits.

On connait le SHA256 de la clef pour vérifier sa valeur.

En examinant le schéma logique du « secure element » il est façile de reconnaitre un ALU (Arithmetic Logic Unit). En fonction des selecteurs, il retourne en sortie un XOR, un OR, un AND ou bien une addition entre les valeurs A et B.

On peut écrire la fonction C suivante qui est équivalente au « secure element ».

```
unsigned char secure_device_int(unsigned char A, unsigned char B, unsigned char op, unsigned char buttons)
{
	int bt4, bt1, bt2, bt3;
	int op0, op1;

	unsigned char IA, IB;

	unsigned char out;

	bt1 = buttons & 1;
	bt2 = buttons & 2;
	bt3 = buttons & 4;
	bt4 = buttons & 8;

	op0 = op & 1;
	op1 = (op & 2)>>1;

	if (bt3 != 0)
		IA = ((A <<1) | (A >>7 )) & 0xFF ;

	else
		IA = A;



	if (bt4 != 0)
		IB = ((B <<1) | (B >>7 )) & 0xFF ;

	else
		IB = B;

	if (bt1 != 0)
		op0 = 1- op0;

	if (bt2 != 0)
		op1 = 1- op1;

	if (op0 != 0 && op1 == 0)
		out = IA | IB;
	else if (op0 == 0 && op1 == 0)
		out = IA & IB;
	else if (op0 == 0 && op1 != 0)
		out = IA ^ IB;
	else {
		out = IA + IB;
	}
```

```
        return (out);
}
```

Maintenant on peut brute forcer les 32 bits de la combinaison pour trouver la clef attendue. Le programme secure_device est disponible en annexe.

En quelques minutes, on trouve :

```
Key Found
8F A4 DF A9 D4 ED BB F0

# python get_safe1_key_V2.py
[i] Dechiffrement du conteneur

[+] Hash ok
[i] Dérivation de la clef AES safe_01
[i] aes key : 5fb3a83d1fd97137076019ad6e96c6a366fb6b32618d162e00cdee9bad427a8a
[i] Vous pouvez sauvegarder cette clef en utilisant /root/tools/add_key.py key

[+] Key with key_id 00000002 ok
[+] Key added into keystore
[+] Envoyez le flag SSTIC{5fb3a83d1fd97137076019ad6e96c6a366fb6b32618d162e00cdee9bad427a8a} à l'adresse
challenge2019@sstic.org pour valider votre avancée
[+] Container /root/safe_01/.encrypted decrypted to /root/safe_01/decrypted_file
```

4. Dwarf VM

1. Programme safe_01

Le decrypted_file dans safe_01 est un programme de type « crackme ». On doit trouver le flag qui est accepté par le programme.

```
# ./decrypted_file
Usage : ./decrypted_file <flag>
#
# ./decrypted_file sstic
Not good
```

On decompile le programme avec l'outil *Ghidra*, on trouve que la structure du programme est très simple :

- La fonction main(int argc, char *arvg[]) [*FUN_00402e68*] appel la fonction *FUN_00402e34* avec argv[1] en paramètre.
- La fonction *FUN_00402e34* jette une exception avec la chaine de caractère passée en paramètre.

```
undefined8 FUN_00402e68(int iParm1,undefined8 *puParm2)

{
 if (iParm1 != 2) {
  printf("Usage : %s <flag>\n",*puParm2);
          /* WARNING: Subroutine does not return */
  exit(1);
 }
          /* try { // try from 00402eb4 to 00402eb7 has its CatchHandler @ 00402f20 */
 FUN_00402e34(puParm2[1]);
 return 0;
}
```

```
void FUN_00402e34(undefined8 uParm1)

{
 undefined8 *puVar1;

 puVar1 = (undefined8 *)__cxa_allocate_exception(8);
 *puVar1 = uParm1;
          /* WARNING: Subroutine does not return */
 __cxa_throw(puVar1,typeinfo,0);
}
```

Le handler d'exception est situé à l'adresse 0x402f20.

(NB : L'adresse des handlers d'exception se trouve dans la section *.gcc_except_table* du fichier ELF dans les structures LSDA call site record).

Il affiche la chaine de caractère dont l'adresse est dans le registre x28. (`0x4030b8 : « Not good »` ou `0x403098 : « That's good flag »).` Mais d'où vient la valeur du registre x28 ?

```
402f20:    aa1c03e1    mov    x1, x28
402f24:    d0000080    adrp   x0, 414000 <_ZTIPc@@CXXABI_1.3+0x248>
402f28:    9102e000    add    x0, x0, #0xb8
402f2c:    f9000001    str    x1, [x0]
402f30:    d0000080    adrp   x0, 414000 <_ZTIPc@@CXXABI_1.3+0x248>
402f34:    9102e000    add    x0, x0, #0xb8
402f38:    f9400000    ldr    x0, [x0]
402f3c:    97ffff3d    bl     402c30 <puts@plt>
402f40:    97ffff5c    bl     402cb0 <__cxa_end_catch@plt>
402f44:    17ffffdd    b      402eb8 <_ZNSt8ios_base4InitD1Ev@plt+0x198>
402f48:    aa0003f3    mov    x19, x0
402f4c:    97ffff59    bl     402cb0 <__cxa_end_catch@plt>
402f50:    aa1303e0    mov    x0, x19
402f54:    97ffff67    bl     402cf0 <_Unwind_Resume@plt>
402f58:    f9400bf3    ldr    x19, [sp,#16]
402f5c:    a8c47bfd    ldp    x29, x30, [sp],#64
402f60:    d65f03c0    ret
```

Quand on observe la taille des sections du fichier ELF, on trouve une anomalie. La taille de la section .gnu.hash est anormalement grande. Cette section contient une table de hash précalculée sur les symboles du programme. La section .gnu.hash est probablement utilisée pour dissimuler des données…

```
S:~/SSTIC/2019/virtual_phone/safe1$ size -A decrypted_file
decrypted_file  :
section              size      addr
.interp                27   4194760
.note.ABI-tag          32   4194788
.note.gnu.build-id     36   4194820
.gnu.hash            9248   4194856
.dynsym               480   4204104
.dynstr               379   4204584
.gnu.version           40   4204964
.gnu.version_r        112   4205008
.rela.dyn              48   4205120
.rela.plt             384   4205168
.init                  20   4205552
.plt                  288   4205584
.text                 820   4205872
.fini                  16   4206692
.rodata               141   4206712
.eh_frame_hdr          92   4206856
.eh_frame             352   4206952
.gcc_except_table      36   4207308
.init_array            16   4275616
.fini_array             8   4275632
.data.rel.ro           32   4275640
.dynamic              512   4275672
.got                   16   4276184
.got.plt              152   4276200
.data                  48   4276352
.bss                   24   4276400
.comment               17         0
Total               13376
```

Pour trouver d'où vient la valeur du registre x28, il est nécessaire de comprendre le fonctionnement interne des exceptions C++.

Quand une exception est jetté en C++, la fonction *__cxa_throw* est appelée. Cette fonction doit remonter les frames de la pile d'appel jusqu'à trouver un handler qui accepte le type d'exception envoyé. Pour remonter les frames, *__cxa_throw* appel la fonction *_Unwind_RaiseException*. La fonction *_Unwind* utilise les données CFI (Call Frame Information) qui sont situées dans la section *.eh_frame* du fichier ELF pour remonter la pile d'appel.

Le format des données CFI est spécifié dans le standard « *DWARF Debugging Information Format* ».

Il est possible d'afficher les données CFI à l'aide de la commande *readelf --debug-dump=frames ./decrypted_file*.

```
$ readelf --debug-dump=frames ./decrypted_file

Contents of the .eh_frame section:

00000000 0000000000000010 00000000 CIE
  Version:               1
  Augmentation:          "zR"
  Code alignment factor: 1
  Data alignment factor: -8
  Return address column: 30
  Augmentation data:     1b

  DW_CFA_def_cfa: r31 (sp) ofs 0

00000014 0000000000000018 00000000 CIE
  Version:               1
  Augmentation:          "zPLR"
  Code alignment factor: 1
  Data alignment factor: -8
  Return address column: 30
  Augmentation data:     9b 19 0f 01 00 1b 1b

  DW_CFA_def_cfa: r31 (sp) ofs 0

....

00000090 000000000000001c 00000094 FDE cie=00000000 pc=0000000000402e34..0000000000402e68
  DW_CFA_advance_loc: 1 to 0000000000402e35
  DW_CFA_def_cfa_offset: 32
  DW_CFA_offset: r29 (x29) at cfa-32
  DW_CFA_offset: r30 (x30) at cfa-24
  DW_CFA_val_expression: r28 (x28) (DW_OP_skip: -12222)
  DW_CFA_nop
  DW_CFA_nop
```

Les instructions DW_CFA permettent typiquement à un debugger de restaurer le pointeur de pile (CFA : Canonical Frame Address) ainsi que les registres en remontant la pile d'appel.

Par exemple, pour la fonction FUN_00402e34 (*plage d'adresse : pc=0000000000402e34..0000000000402e68*), les instructions : *"DW_CFA_offset: r29 (x29) at cfa-32"* et*"DW_CFA_offset: r30 (x30) at cfa-24"* indiquent que les registres x29 et x30 sont sauvegardés dans la pile aux offsets -32 et -24. Ça correspont effectivement au code de la fonction, les registres x29 et x30 sont bien sauvegardés dans la pile au début de la fonction.

```
402e34:    a9be7bfd    stp    x29, x30, [sp,#-32]!
402e38:    910003fd    mov    x29, sp
402e3c:    f9000fe0    str    x0, [sp,#24]
402e40:    d2800100    mov    x0, #0x8            // #8
```

Mais la partie intéressante est la ligne :

DW_CFA_val_expression: r28 (x28) (DW_OP_skip: -12222)

La valeur du registre x28 provient du résultat de l'évaluation de « DWARF expressions ». Les DWARF expressions sont une séquence d'opération sur une machine à pile.

Elles sont normalement utilisées pour les « cas compliqués » (frames non standard...) où les instructions DW_CFA_ ne suffisent pas pour restaurer les registres.

L'opcode DW_OP_skip est l'équivalent d'un JUMP, il permet de déplacer le pointeur d'instruction des opcodes DWARF. Ici on recule de -12222 octets. On saute en dehors des données de la section *.eh_frame* dans les données cachées dans la section *.gnu.hash_table* !!

- ⇨ Le code de vérification du flag est donc écrit avec des opcodes DWARF !

2. Dwarf Virtual Machine

Le code de la machine à pile DWARF est implementé dans la lib *libgcc_s.so* dans la fonction *execute_stack_op (const unsigned char *op_ptr, const unsigned char *op_end, struct _Unwind_Context *context, _Unwind_Word initial)* dans le fichier *unwind-dw2.c*.

Avec l'aide de gdb, on va tracer l'execution de la VM Dwarf

```
define SetBP2
set pagination off
set $ref=_Unwind_RaiseException
set $adr1_execute_cfa_program=$ref+84-5192
set $adr2_execute_stack_op=$ref+84-5192+2240

set $adr3_execute_stack_ins=$ref+84-2836+4
set $adr4_execute_stack_ret=$ref+84-2836-28+4

set $adr5_exec_stack_push_res=$adr3_execute_stack_ins + 640

set $adr6_exec_stack_ins_bra=$adr5_exec_stack_push_res+592

b _Unwind_RaiseException
commands
print "Unwind_RaiseException"
continue
end

b *($adr2_execute_stack_op)
commands
print "execute_stack_op"
print $x1-$x0
continue
end

b *($adr1_execute_cfa_program)
commands
print "execute_cfa_prog"
print $x1-$x0
continue
end

b *($adr3_execute_stack_ins)
commands
print "stack_ins"
print /x $w7
print /x $x0
continue
end

b *($adr4_execute_stack_ret)
commands
print "stack_ret"
print /x $x0
continue
end

b *($adr5_exec_stack_push_res)
commands
print "push_res"
print /x $x6
end

b *($adr6_exec_stack_ins_bra)
commands
print "ins_bra"
```

```
print  $x0 - 0x400258
end

b __gxx_personality_v0
commands
print "gxx_personality"
continue
end

b _Unwind_SetIP
end
```

```
(gdb) b *0x402e34
Breakpoint 1 at 0x402e34
(gdb) run  1234
Starting program: /root/safe_01/decrypted_file 1234

Breakpoint 1, 0x0000000000402e34 in ?? ()

(gdb) SetBP2
Breakpoint 2 at 0xffff8ca5543c
Breakpoint 3 at 0xffff8ca548b4
Breakpoint 4 at 0xffff8ca53ff4
Breakpoint 5 at 0xffff8ca5492c
Breakpoint 6 at 0xffff8ca54910
Breakpoint 7 at 0xffff8ca54bac
Breakpoint 8 at 0xffff8ca54dfc
Breakpoint 9 at 0xffff8cbb3914
Breakpoint 10 at 0xffff8ca53e5c

(gdb) c
Continuing.

Breakpoint 12, 0x0000ffff9616e43c in _Unwind_RaiseException () from /lib64/libgcc_s.so.1
$64 = "Unwind_RaiseException"

Breakpoint 13, 0x0000ffff9616d8b4 in ?? () from /lib64/libgcc_s.so.1
$65 = "execute_stack_op"
$66 = 3

Breakpoint 15, 0x0000ffff9616d92c in ?? () from /lib64/libgcc_s.so.1
$67 = "stack_ins"
$68 = 0x2f
$69 = 0x403213

Breakpoint 15, 0x0000ffff9616d92c in ?? () from /lib64/libgcc_s.so.1
$70 = "stack_ins"
$71 = 0x6f
$72 = 0x400258
```

Le premier opcode executé est 0x2F DW_OP_skip qui déplace le pointeur d’instruction à l’adresse 0x400258 où se trouve le programme caché dans la section .gnu.hash.

Pour comprendre le fonctionnement du programme, on écrit un désassembleur de Dwarf Opcodes. Le code du désassembleur *dwarf_disass* est disponible en annexe.

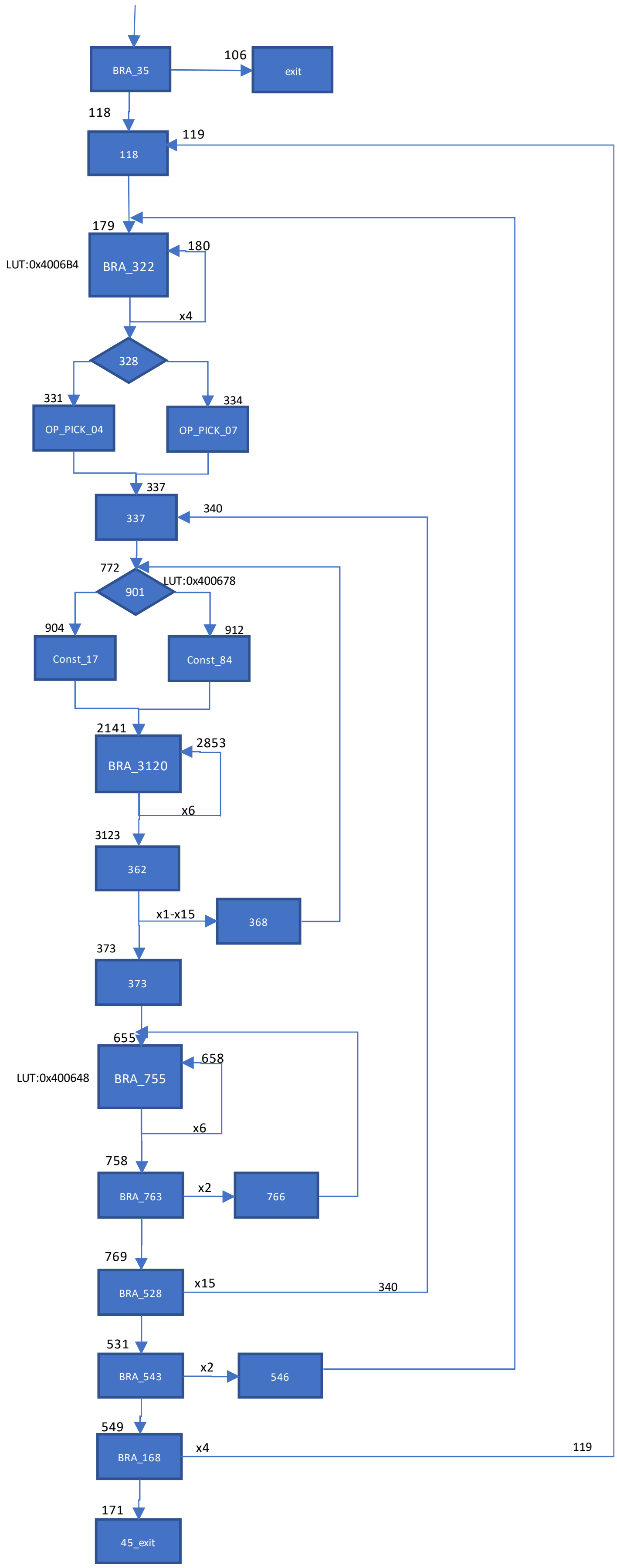
```
000: [000] 0x6f, DW_OP_reg31,                    (0x400258)
001: [001] 0x08, DW_OP_const1u, A8
002: [003] 0x22, DW_OP_plus,
```

```
003: [004] 0x06, DW_OP_deref,
004: [005] 0x08, DW_OP_const1u, 08
005: [007] 0x22, DW_OP_plus,
006: [008] 0x06, DW_OP_deref,
007: [009] 0x12, DW_OP_dup,
008: [010] 0x06, DW_OP_deref,
009: [011] 0x16, DW_OP_swap,
010: [012] 0x08, DW_OP_const1u, 08
011: [014] 0x22, DW_OP_plus,
012: [015] 0x12, DW_OP_dup,
013: [016] 0x06, DW_OP_deref,
014: [017] 0x16, DW_OP_swap,
015: [018] 0x08, DW_OP_const1u, 08
016: [020] 0x22, DW_OP_plus,
017: [021] 0x12, DW_OP_dup,
018: [022] 0x06, DW_OP_deref,
019: [023] 0x16, DW_OP_swap,
020: [024] 0x08, DW_OP_const1u, 08
021: [026] 0x22, DW_OP_plus,
022: [027] 0x12, DW_OP_dup,
023: [028] 0x06, DW_OP_deref,
024: [029] 0x16, DW_OP_swap,
025: [030] 0x08, DW_OP_const1u, 08
026: [032] 0x22, DW_OP_plus,
027: [033] 0x94, DW_OP_deref_size, 01                    ===> Stack : 32 bytes from argv[1]... / IW3, IW2, IW1, IW0
028: [035] 0x28, DW_OP_bra, 44 00 [Jump_addr= 106]       ====> Check char [33] !=0 ==> Exit ("Not good").


029: [038] 0x15, DW_OP_pick, 03                          ====> Push stack[top-3]
030: [040] 0x15, DW_OP_pick, 03
031: [042] 0x2f, DW_OP_skip, 49 00 [Jump_addr= 118]      ====> IW1, IW0, IW3, IW2, IW1, IW0


032: [045] 0x0e, DW_OP_const8u, 24 1C 8E 8E A6 02 83 65  ====> check_flag_result()
033: [054] 0x27, DW_OP_xor,
034: [055] 0x16, DW_OP_swap,
035: [056] 0x0e, DW_OP_const8u, 47 53 2E 61 F1 64 75 DC
036: [065] 0x27, DW_OP_xor,
037: [066] 0x22, DW_OP_plus,
038: [067] 0x16, DW_OP_swap,
039: [068] 0x0e, DW_OP_const8u, 13 C6 6E A8 74 9B C6 D9
040: [077] 0x27, DW_OP_xor,
041: [078] 0x22, DW_OP_plus,
042: [079] 0x16, DW_OP_swap,
043: [080] 0x0e, DW_OP_const8u, D5 AE 6A E7 36 0B 85 65
044: [089] 0x27, DW_OP_xor,
045: [090] 0x22, DW_OP_plus,
046: [091] 0x28, DW_OP_bra, 0C 00 [Jump_addr= 106]       ===> Check Stack content : 24 1C 8E 8E A6 02 83 65 , 47 53 2E
61 F1 64 75 DC,
047: [094] 0x0e, DW_OP_const8u, 98 30 40 00 00 00 00 00  ====> if == exit('That's good flag')
048: [103] 0x2f, DW_OP_skip, FF 7F [Jump_addr= 32873]
049: [106] 0x0e, DW_OP_const8u, B8 30 40 00 00 00 00 00
050: [115] 0x2f, DW_OP_skip, FF 7F [Jump_addr= 32885]    ==> Exit ("Not good").


051: [118] 0x30, DW_OP_lit0,                        ===> Push 0 on stack.  ====> 0, IW1, IW0, IW3, IW2, IW1, IW0
052: [119] 0x17, DW_OP_rot,                         ===> stack[top] = stack[top-1]; stack[top-1]= stack[top-2]; stack[top-2] =
stack[top]; ====> IW1, IW0, 0, IW3, IW2, IW1, IW0
053: [120] 0x30, DW_OP_lit0,                        ===> 0, IW1, IW0, 0, IW3, IW2, IW1, IW0
054: [121] 0x15, DW_OP_pick, 05
055: [123] 0x15, DW_OP_pick, 05
056: [125] 0x2f, DW_OP_skip, 33 00 [Jump_addr= 179] ===> IW3, IW2, 0, IW1, IW0, 0, IW3, IW2, IW1, IW0

057: [128] 0x15, DW_OP_pick, 04
058: [130] 0x15, DW_OP_pick, 04
059: [132] 0x2f, DW_OP_skip, CA 00 [Jump_addr= 337]
```

Le début du programme place sur la pile le flag passé en paramètre. Il vérifie que l'octet 33 est à 0 (le flag doit donc être une chaine de 32 caractères). Si ce n'est pas le cas, le programme place sur la pile la valeur 0x4030B8 (l'adresse de la chaine « Not good ») avant de se terminer.

Si la longueur du flag est correcte, le programme effectue un calcul sur les valeurs en entrée et vérifie que le résultat est égal à des constantes (opcodes de 45 à 115). Si les valeurs sont bien égales, il retourne sur la pile 0x403098 « That's good flag » ou bien 0x4030B8 « Not good » dans le cas contraire.

On représente la structure des boucles du programme sur le schéma ci-dessous.



NB : Le bloc d’instruction entre 2141 et 3123 effectue un calcul complexe qui ne dépend pas des valeurs en entrée mais uniquement de la constante placée sur la pile avant l’appel en 2141. Il y a deux valeurs possibles pour la constante en fonction du test 901. On peut donc remplacer le bloc d’instruction 2141-3123 par le résultat du calcul…

Après un travail fastidieux, on écrit un programme en C qui effectue le même calcul que le programme DWARF.

On peut ensuite inverser l’algorithme pour trouver la valeur en entrée correspondant au résultat attendu (constantes du bloc d’instruction 45-115). (Le programme InvHash est disponibel en annexe).

On fini par trouver :

```
#o0 =77447b4349545353
#o1 =315f4d565f667234
#o2 =695f6c306f635f73
#o3 =7d74495f745f6e73
Key=SSTIC{Dw4rf_VM_1s_co0l_isn_t_It}
```

On a notre flag: SSTIC{Dw4rf_VM_1s_co0l_isn_t_It}.

5. ARM TrustZone

1. Programme safe_02

Le decrypted_file dans safe_02 est un également programme de type « crackme ». On doit trouver le flag qui est accepté par le programme.

```
# ./decrypted_file
usage: ./decrypted_file [32-bytes-key-hex-encoded]
# ./decrypted_file 1234567890123456789012345678901234567890123456789012345678901234
Loose
```

On decompile le programme avec l'outil *Ghidra*, on trouve que la structure du programme est assez simple :

```
int main(int argc, char *argv[])
{

	long key;
	int ret;

	int fd;

	struct _param
	{
		uchar *p1;
		ulong p2;
	} param;

	if ((argc ==2) && strlen(argv[1] ) == 64 	) {

		ret = decodeHex(argv[1], &key);

		if (ret == 0) {
			printf("Can't decode hex\n");
			return(1);
		}

		fd = open("/dev/sttic");

		param.p1 = &DAT_0044dbd8; // Data : 0x101010 bytes
		param.p2 = 0x101010;
		ioctl(fd, 0xc0105300, &param);

		param.p1 = &key;
		param.p2 = 0x20;
		ioctl(fd, 0xc0105301, &param);

		do {
			ioctl(fd, 0xc0105302, NULL);
			ret = ioctl(fd, 0xc0105303, NULL);

			if ((ret & 0xFFFF) ==1) {
				if ((ret & 0xFFFF0000) ==0) {
					printf("Win\n");
```

```
                        } else {
                                printf("Loose\n");
                        }
                }

        } while( (ret & 0xFFFF) != 0xFFFF);

        printf("Failure\n");

    } else {
        printf("Usage ..\n");
        return(1);
    }


}
```

Après avoir décodé la chaine de 32 octets en hexadécimal, le programme effectue les opérations suivantes :

1/ Envoi un bloc de données de 0x101010 octets au device /dev/sstic par un ioctl(0xc0105300).

2/ Envoi la clef de 32 octets au device /dev/sstic par un ioctl(0xc0105301).

3/ Boucle :

```
        ioctl(0xc0105302)

        ret = ioctl(0xc0105303)

        Si (ret & 0xFFFF0000 ==0)

                Printf(« Win »)

        Sinon

                Printf(« Loose »)

  Tant que (ret & 0xFFFF != 0xFFFF)
```

2. module sstic.ko

On décompile le module *sstic.ko* avec l'outil *Ghidra* pour comprendre ce que font les ioctls.

On analyse la fonction *sstic_ioctl.*

C'est essentielement un adapter qui transforme les appels ioctl en appels SMC vers le secure monitor.

NB : __arm_smccc_smc est wrapper dans le kernel linux pour les appels SMC.

```
/**
 * arm_smccc_smc() - make SMC calls
 * @a0-a7: arguments passed in registers 0 to 7
 * @res: result values from registers 0 to 3
 *
 * This function is used to make SMC calls following SMC Calling Convention.
 * The content of the supplied param are copied to registers 0 to 7 prior
 * to the SMC instruction. The return values are updated with the content
 * from register 0 to 3 on return from the SMC instruction.
 */
asmlinkage void arm_smccc_smc(unsigned long a0, unsigned long a1,
                        unsigned long a2, unsigned long a3, unsigned long a4,
                        unsigned long a5, unsigned long a6, unsigned long a7,
                        struct arm_smccc_res *res);
```

ioctl(0xc0105300) : Send Data

- ⇨ __arm_smccc_smc (0x83010004, buf, len);

Ioctl(0xc0105301): Send Key

- ⇨ for (lVar4=0; lVar4<8; lVar4++)
  __arm_smccc_smc(0xf2005003, lVar4, (ulonglong)*(uint *)(KeyBuffer + lVar4 * 4));

Ioctl(0xc0105302): Fetch Instruction

__arm_smccc_smc(0xf2005001);

Ioctl(0xc0105303): Run instruction & Return Status

__arm_smccc_smc(0xf2005002);

Ioctl(0xc1105305) : READ_AES_KEY in keystore

__arm_smccc_smc(0x83010005,pvVar8);

Ioctl(0xc1105306) : ADD_KEY in keystore

__arm_smccc_smc(0x83010006,pvVar8);

NB: Les deux derniers appels READ_AES_KEY and ADD_KEY sont utilisés par les outils de gestion du keystore.

3. Boot loaders

Les messages de la séquence de boot nous donnent des informations sur le contenu de la flash.

Un Bootloader BL1 décrypte et démarre un bootloader BL2.

Le bootloader BL2 décrypte 3 images.

Ensuite le bootloader BL1 démarre un bootloader BL31 qui démarre un bootloader BL32.

Puis le Secure OS démarre.

Et enfin Linux boot.

```
##########################################
#     virtual environment detected     #
#         QEMU 3.1+ is needed         #
##########################################
NOTICE:  Booting SSTIC ARM Trusted Firmware
KEYSTORE: AES Key already decrypted
NOTICE:  Loading image id=1
NOTICE:  BL1: Booting BL2
KEYSTORE: BL2 got key : key_type:0x02, key_len:0x20
HEXDUMP :
-----------------------------------------------
53 53 54 49 43 7b 61 39 34 37 64 36 39 38 30 63
63 66 37 62 38 37 63 62 38 64 37 63 32 34 36 7d
-----------------------------------------------
NOTICE:  Loading image id=3
KEYSTORE: BL2 got key : key_type:0x02, key_len:0x20
HEXDUMP :
-----------------------------------------------
53 53 54 49 43 7b 44 77 34 72 66 5f 56 4d 5f 31
73 5f 63 6f 30 6c 5f 69 73 6e 5f 74 5f 49 74 7d
-----------------------------------------------
NOTICE:  Loading image id=4
KEYSTORE: BL2 got key : key_type:0x02, key_len:0x20
HEXDUMP :
-----------------------------------------------
53 53 54 49 43 7b 61 39 34 37 64 36 39 38 30 63
63 66 37 62 38 37 63 62 38 64 37 63 32 34 36 7d
-----------------------------------------------
NOTICE:  Loading image id=5
NOTICE:  BL1: Booting BL31
NOTICE:  BL31: Initializing BL32
NOTICE:  Booting Secure-OS
UEFI firmware (version  built at 00:01:39 on Feb 25 2019)
EFI stub: Booting Linux Kernel...
EFI stub: Generating empty DTB
EFI stub: Exiting boot services and installing virtual address map...
[    0.000000] Booting Linux on physical CPU 0x0000000000 [0x411fd070]
```

On cherche à obtenir le contenu déchiffré de flash.bin.

En ajoutant l’option ‘-gdb tcp ::1234’ à la ligne de commande de qemu, on peut attacher gdb pour tracer l’exécution du système.

```
qemu-system-aarch64 -S -gdb tcp::1234 -nographic -machine virt,secure=on -cpu max -smp 1 -m 1
024 -bios rom.bin -semihosting-config enable,target=native -device loader,file=./flash.bin,addr=0x04000000
```

Après analyse de rom.bin, on trouve que la fonction de déchiffrement pour décrypter BL2 est FUN_00002c64(). On peut mettre un point d'arrêt à la fin de cette fonction pour récuperer le contenu de BL2 en clair.

```
(gdb) target remote localhost:1234
(gdb) b *0x2c98
Breakpoint 1 at 0x2c98
(gdb) c
Continuing.
Breakpoint 1, Python Exception <type 'exceptions.NameError'> Installation error: gdb.execute_unwinders function is missing:
0x0000000000002c98 in ?? ()
(gdb) dump binary memory BL2.bin 0xe00b000  0xe00b000 +0x9440
```

De la même manière on trouve que la fonction de déchiffrement pour décrypter les images chargées par BL2 est FUN_0000207c(). En mettant un point d'arrêt à la fin de cette fonction, on peut récuperer les images en clair.

```
(gdb) b *0xe00d0bc
Breakpoint 2 at 0xe00d0bc
(gdb) c
Continuing.
Breakpoint 2, Python Exception <type 'exceptions.NameError'> Installation error: gdb.execute_unwinders function is missing:
0x000000000e00d0bc in ?? ()
(gdb) dump binary memory BL31.bin 0xe030000  0xe030000+0x90b0
(gdb) c
Continuing.
Breakpoint 2, Python Exception <type 'exceptions.NameError'> Installation error: gdb.execute_unwinders function is missing:
0x000000000e00d0bc in ?? ()
(gdb) dump binary memory BL32.bin 0xe200000 0xe205380
(gdb) c
Continuing.
Breakpoint 2, Python Exception <type 'exceptions.NameError'> Installation error: gdb.execute_unwinders function is missing:
0x000000000e00d0bc in ?? ()
(gdb) dump binary memory Linux_img.bin 0x60000000 0x60000000+  0x2220030
```

4. Security monitor

Le point d'entrée du secure monitor est situé à l'adresse VBAR_EL3 + 0x400.

On peut utiliser gdb pour tracer les appels à SMC.

```
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
(gdb) set pagination off
(gdb)
(gdb) set logging on
Copying output to gdb.txt.
(gdb) print /x $VBAR_EL3
$5965 = 0xe037000
(gdb) b *0x000000000e037400
Breakpoint 1 at 0xe037400
(gdb) commands
Type commands for breakpoint(s) 1, one per line.
End with a line saying just "end".
>silent
>echo \nSMC:\n
>print /x $x0
>print /x $x1
>print /x $x2
>print /x $ELR_EL3
>continue
>end
(gdb) info breakpoints
Num     Type          Disp Enb Address            What
1       breakpoint    keep y   0x000000000e037400
        breakpoint already hit 91145 times
        silent
        print "SMC:"
        print /x $x0
        print /x $x1
        print /x $ELR_EL3
        continue
2     breakpoint    keep y   0x000000000e035a20
        breakpoint already hit 96360 times
        silent
        print "eret"
        print /x $x0
        print /x $x1
        print /x $x2
        print /x $x3
        print /x $ELR_EL3
        continue
 (gdb) c
==> Load prog
$1096302 = "SMC:"                      ==> Load Prog.
$1096303 = 0x83010004
$1096304 = 0x7e200000
$1096305 = 0xffff00001009395c

$1096306 = "eret"
$1096307 = 0x0
$1096308 = 0x7e200000
$1096309 = 0xffff80003fde6440
$1096310 = 0x4000000000000000
$1096311 = 0xffff00001009395c          ==> Ret Linux

===> Load Key
$1096312 = "SMC:"                      ==> Load Key (0)
$1096313 = 0xf2005003
$1096314 = 0x0 //Ki
$1096315 = 0xffff00001009395c

$1096316 = "eret"
$1096317 = 0xf2005003
```

```
$1096318 = 0x0        //Ki
$1096319 = 0x78563412        // Key value
$1096320 = 0x0
$1096321 = 0xe200094                ==> Secure OS.

$1096322 = "SMC:"
$1096323 = 0xf2001000
$1096324 = 0x0        //Ki
$1096325 = 0xe2022e0                ==> Secure OS.

$1096326 = "eret"
$1096327 = 0x0                //Ki
$1096328 = 0x78563412        //key val
$1096329 = 0x78563412
$1096330 = 0x0
$1096331 = 0xe2022e0                ==> Secure OS.

$1096332 = "SMC:"
$1096333 = 0x83010002
$1096334 = 0xf
$1096335 = 0xe200d4c                ==> Secure OS.

$1096336 = "eret"
$1096337 = 0x0
$1096338 = 0xf
$1096339 = 0x78563412
$1096340 = 0x0
$1096341 = 0xe200d4c                ==> Secure OS.

$1096342 = "SMC:"
$1096343 = 0xf2005003
$1096344 = 0x0        //0x0
$1096345 = 0xe2001e8

$1096346 = "eret"
$1096347 = 0x0
$1096348 = 0x0
$1096349 = 0x0
$1096350 = 0x4000000000000000
$1096351 = 0xffff00001009395c==> Ret Linux


================> Load instruction
$1096672 = "SMC:"
$1096673 = 0xf2005001
$1096674 = 0x0
$1096675 = 0xffff00001009395c

$1096676 = "eret"
$1096677 = 0xf2005001
$1096678 = 0x0
$1096679 = 0x0
$1096680 = 0x0
$1096681 = 0xe200094        => Secure OS

$1096682 = "SMC:"
$1096683 = 0xf2001000
$1096684 = 0x0
$1096685 = 0xe2022e0

$1096686 = "eret"
$1096687 = 0x0
$1096688 = 0x0
$1096689 = 0x0
$1096690 = 0x0
$1096691 = 0xe2022e0        => Secure OS

$1096692 = "SMC:"
$1096693 = 0x83010001
$1096694 = 0xf
$1096695 = 0xe200c68
```

```
$1096696 = "eret"
$1096697 = 0x0                          => Instruction pointer.
$1096698 = 0xf
$1096699 = 0x0
$1096700 = 0x0
$1096701 = 0xe200c68                    => Secure OS

$1096702 = "SMC:"
$1096703 = 0xf2005001
$1096704 = 0x0
$1096705 = 0xe2001e8

$1096706 = "eret"
$1096707 = 0x0
$1096708 = 0x0
$1096709 = 0x0
$1096710 = 0x4000000000000000
$1096711 = 0xffff00001009395c
$1096712 = 0xf2005002

=====

$1096713 = "SMC:"
$1096714 = 0xf2005002
$1096715 = 0x0
$1096716 = 0xffff00001009395c           => Linux

$1096717 = "eret"
$1096718 = 0xf2005002
$1096719 = 0x0
$1096720 = 0x0
$1096721 = 0x0
$1096722 = 0xe200094                    => Secure OS

$1096723 = "SMC:"
$1096724 = 0xf2001000
$1096725 = 0x0
$1096726 = 0xe2022e0

$1096727 = "eret"
$1096728 = 0x0
$1096729 = 0x0
$1096730 = 0x0
$1096731 = 0x0
$1096732 = 0xe2022e0

$1096733 = "SMC:"
$1096734 = 0x83010002
$1096735 = 0x4
$1096736 = 0xe20224c

$1096737 = "eret"
$1096738 = 0x0
$1096739 = 0x4
$1096740 = 0x0
$1096741 = 0x0
$1096742 = 0xe20224c

$1096743 = "SMC:"
$1096744 = 0x83010001
$1096745 = 0xf
$1096746 = 0xe20224c

$1096747 = "eret"
$1096748 = 0x0
$1096749 = 0xf
$1096750 = 0x0
$1096751 = 0x0
$1096752 = 0xe20224c
```

```
$1096753 = "SMC:"
$1096754 = 0x83010002
$1096755 = 0xf
$1096756 = 0xe20224c

$1096757 = "eret"
$1096758 = 0x0
$1096759 = 0xf
$1096760 = 0x0
$1096761 = 0x0
$1096762 = 0xe20224c


$1096763 = "SMC:"
$1096764 = 0xf2005002
$1096765 = 0x0
$1096766 = 0xe2001e8

$1096767 = "eret"
$1096768 = 0x0
$1096769 = 0x0
$1096770 = 0x0
$1096771 = 0x4000000000000000
$1096772 = 0xffff00001009395c             => Ret Linux
```

On retrouve les appels provenant du module sstic.ko. (Les autres appels au SMC proviennent du Secure OS).

- SMC(0x83010004) : Chargement du programme dans le secureOS.
- SMC(0xf2005003) : Chargement de 32 bits de la clef dans le secure OS.
    - SMC(0xf2001000)
    - SMC(0x83010002)
    - SMC(0xf2005003)
- SMC(0xf2005001) : Decodage de la prochaine instruction
    - SMC(0xf2001000)
    - SMC(0x83010001)   => Lecture du pointeur d’instruction
    - SMC(0xf2005001)
- SMC(0xf2005002) : Execution de l’instruction courante
    - SMC(0xf2001000)
    - SMC(0x83010001)
    - SMC(0x83010002)
    - SMC(0xf2005002)

On remarque que la valeur de retour de l’appel SMC(0x83010001) qui a lieu après chaque appel SMC(0xf2005001) semble être un pointeur d’instruction. Cette valeur augmente de 3 à chaque itération et parfois revient en arrière (comme pour l’execution de boucles…)

On va désassembler le code du SMC (qui est en fait dans le « bootloader » BL31) pour comprendre ce que font ces instructions.

Les opérations 0x8301XXXX du SMC sont gérées dans la fonction *BL31_FUN_01034(uint uParm1,undefined8 *puParm2,ulonglong uParm3)*

Pour l'opération 0x83010001, le code est le suivant

```
                switchD_000011bc::caseD_1                  XREF[1]:     000011bc (j)
     000011c0 73  1e  00  12    and       w19 ,w19 ,#0xff
     000011c4 7f  3e  00  71    cmp       w19 ,#0xf
     000011c8 28  03  00  54    b.hi      LAB_0000122c
     000011cc 73  1e  7c  d3    ubfiz     x19 ,x19 ,#0x4 ,#0x8
     000011d0 c0  00  00  b0    adrp      x0,0x1a000
     000011d4 00  60  17  91    add       x0,x0,#0x5d8
     000011d8 e1  43  01  91    add       x1,sp,#0x50
     000011dc 73  02  00  8b    add       x19 ,x19 ,x0
     000011e0 00  53  3c  d5    mrs       x0,fpexc32_el2
     000011e4 01  1c  04  4e    mov       v1.S[0x0 ],w0
     000011e8 00  30  3c  d5    mrs       x0,dacr32_el2
     000011ec 01  1c  0c  4e    mov       v1.S[0x1 ],w0
     000011f0 20  50  3c  d5    mrs       x0,ifsr32_el2
     000011f4 01  1c  14  4e    mov       v1.S[0x2 ],w0
     000011f8 20  11  3e  d5    mrs       x0,sder32_el3
     000011fc 01  1c  1c  4e    mov       v1.S[0x3 ],w0
     00001200 60  7a  40  4c    ld1       {v0.4S},[ x19 ]
     00001204 20  48  28  4e    aese      v0.16B,v1.16B
     00001208 00  1c  21  6e    eor       v0.16B,v0.16B,v1.16B
     0000120c 20  78  00  4c    st1       {v0.4S},[ x1=>local_100 ]
     00001210 82  00  80  d2    mov       x2,#0x4
     00001214 21  14  00  91    add       x1,x1,#0x5
     00001218 e0  23  01  91    add       x0,sp,#0x48
     0000121c 95  13  00  94    bl        FUN_00006070                         undefined FUN_00006070()
     00001220 e0  4b  40  b9    ldr       w0,[sp, #local_108 ]
```

L'appel SMC 0x83010001 est utilisé pour lire un registre de 32 bits. l'index du registre à lire est mis dans x1 lors de l'appel SMC. Il y a 16 registres disponible. Le contenu des registres est sauvegardé crypté en mémoire. Ils sont à l'adresse (0xe04a5d8 + reg_num * 16)

Le code ci-dessus va decrypté le contenu du registre dont l'index est en $w19. La clef de déchiffrement est dans les registres fpexc32_el2, dacr32_el2, ifsr32_el2, sder32_el3 (dont l'usage a été detourné).

```
0xe031204:   aese    v0.16b, v1.16b                          // V0 = AES_SubBytes(AES_ShiftRow(V0 ^ V1))
 0xe031208:   eor     v0.16b, v0.16b, v1.16b                 // V0 = V0 ^ V1
 0xe03120c:   st1     {v0.4s}, [x1]
```

On anlyse les autres appels SMC(0x8301XX) pour comprendre leur fonction :

```
SMC(0x83010001, p2) : Read Register P2     ==> Return value in $x0
SMC(0x83010002, p2, P3) : Write in Register P2, value P3.

SMC(0x83010011, p2)  :
            Reg[p2] := Reg[p2] -1   (Reg offset: 0x5d8)
            Reg[0x0F] := Reg[0x0F] + 3 (Reg' offset: 0x6c8)

SMC(0x83010012, p2, p3)  :
            Reg[p2] := Reg[p2] + Reg[p3]
```

```
        Reg[0x0F] := Reg[0x0F] + 3 (Reg' offset: 0x6c8)

SMC(0x83010013, p2, p3)  :
        Reg[p2] := Reg[p2] - Reg[p3]
        Reg[0x0F] := Reg[0x0F] + 3 (Reg' offset: 0x6c8)

SMC(0x83010016, p2, p3)  :
        Reg[p2] := Reg[p2] ^ Reg[p3]
        Reg[0x0F] := Reg[0x0F] + 3 (Reg' offset: 0x6c8)

SMC(0x83010022, p2, p3)  :
        Reg[p2] := Reg[p2] + p3
        Reg[0x0F] := Reg[0x0F] + 3 (Reg' offset: 0x6c8)

SMC(0x83010023, p2, p3)  :
        Reg[p2] := Reg[p2] - p3
        Reg[0x0F] := Reg[0x0F] + 3 (Reg' offset: 0x6c8)

SMC(0x83010027, p2, p3)  :
        Reg[p2] := Reg[p2] & p3
        Reg[0x0F] := Reg[0x0F] + 3 (Reg' offset: 0x6c8)
```

5. Secure OS Virtual machine

1. Mémoire protégée

On a vu que la lecture du pointeur d'instruction était effectué par un appel à SMC(0x8301001, 0xf) la valeur du registre ELR_EL3 nous donne l'adresse du code qui a appelé le SMC. C'est 0xe200c68-4.

```
$1096692 = "SMC:"
$1096693 = 0x83010001
$1096694 = 0xf
$1096695 = 0xe200c68
```

On va analyser le code du Secure OS (qui est en fait dans le « bootloader » BL32) qui a appelé le SMC.

```
(gdb) x /30i 0xe200c50
   0xe200c50:  mov    x0, #0x1                    // #1
   0xe200c54:  mov    x1, #0xf                    // #15
   0xe200c58:  movk   x0, #0x8301, lsl #16
   0xe200c5c:  mov    x2, #0x0                    // #0
   0xe200c60:  mov    x3, #0x0                    // #0
   0xe200c64:  smc    #0x0
   0xe200c68:  adrp   x1, 0xe215000
   0xe200c6c:  mov    x2, #0x0                    // #0
   0xe200c70:  str    w0, [x1, #2688]
   0xe200c74:  and    x0, x0, #0xffffffff
   0xe200c78:  mov    x1, #0x0                    // #0
=> 0xe200c7c:  ldr    w0, [x0]
   0xe200c80:  adrp   x1, 0xe215000
   0xe200c84:  and    w0, w0, #0xffffff
   0xe200c88:  str    w0, [x1, #2064]
   0xe200c8c:  mov    x19, #0x0                   // #0
   0xe200c90:  mov    x20, #0x0                   // #0
   0xe200c94:  b      0xe200cb0
```

Après l'appel au SMC(0x8301001), la valeur du pointeur d'instruction est sauvegardée à l'adresse `0xe215000`.

Ensuite le secure OS va executer *« ldr   w0, [x0] »* pour charger le code de l'instruction.

Cet appel va déclencher une « page fault exception » car l'adresse x0 (contenant la valeur du pointeur d'instruction du bytecode) n'est pas mappé dans l'espace d'adressage du SecureOS.

On arrive à l'adresse du handle d'exception : $VBAR+0x200 = `0xe203200`.

Le handler d'exception va récuperer l'adresse qui a déclenché l'exception dans le registre far_el1 puis il va appeler la fonction FUN_00000e84() pour decrypter la mémoire protegée.

Le code utilisé pour decrypter la mémoire protegé est le suivant

```
0xe200ee8:  ld1    {v0.4s}, [x23]
   0xe200eec:  rev32  v0.16b, v0.16b
   0xe200ef0:  ld1    {v1.4s}, [x22]
   0xe200ef4:  mov    v2.s[0], w19
   0xe200ef8:  mov    v2.s[1], w19
   0xe200efc:  mov    v2.s[2], w19
   0xe200f00:  mov    v2.s[3], w19
   0xe200f04:  eor    v1.16b, v1.16b, v2.16b
   0xe200f08:  sm4ekey v0.4s, v0.4s, v1.4s
```

```
0xe200f10:   mov     v1.s[1], v0.s[2]
   0xe200f14:   mov     v1.s[2], v0.s[1]
   0xe200f18:   mov     v1.s[3], v0.s[0]
   0xe200f1c:   ld1     {v0.4s}, [x0]
   0xe200f20:   rev32   v0.16b, v0.16b
   0xe200f24:   sm4e    v0.4s, v1.4s
   0xe200f28:   rev32   v0.16b, v0.16b
   0xe200f2c:   mov     v1.s[0], v0.s[3]
   0xe200f30:   mov     v1.s[1], v0.s[2]
   0xe200f34:   mov     v1.s[2], v0.s[1]
   0xe200f38:   mov     v1.s[3], v0.s[0]
   0xe200f3c:   st1     {v1.4s}, [x20]
   0xe200f40:   cmp     w21, #0xc
   0xe200f44:   b.ls    0xe200fb0  // b.plast
```

La mémoire est decrypté par l'algorithme SM4 : « 0xe200f24:   sm4e   v0.4s, v1.4s ».

La mémoire protegée est stocké cryptée à partir de l'adresse 0x413000.

On retrouve à cette adresse le contenu du bloc de donnée (0x101010 bytes) qui a été transferé au secureOS par l'appel SMC(0x83010004). (En fait pas complétement, il manque la deuxième page de 4K qui n'est pas mappé dans l'espace mémoire du SecureOS. C'est une protection anti-debug quand qemu est lancé avec l'option gdb).

```
(gdb) x /16bx $x0
0x413000:      0x30    0x4b    0x4e    0x8e    0x02    0xd9    0x64    0x64
0x413008:      0xaf    0x65    0xc5    0xdd    0x47    0x6f    0x8a    0xaa
```

Extraction de la mémoire protégée :

Pour extraire le contenu de la mémoire protégée en version decrypté sans avoir à écrire de code, on va utiliser gdb.

On détourne la fonction de lecture d'instruction pour décrypter une zone de la mémoire protegée. Les commandes gdb suivantes permettent de decrypter la zone mémoire situées entre les adresses $a et $b.

```
b *0x000000000e200c80
commands
silent
print /x $x0
set $pc=0xe200c7c
set $x0=$a
set $a=$a+4
if $a<$b
continue
end
end
```

On utilise cette méthode pour dumper en clair le bytecode du programme.

```
Breakpoint 1 at 0xe200c7c
Breakpoint 2 at 0xe200c80
Type commands for breakpoint(s) 2, one per line.
End with a line saying just "end".
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000e200c7c
```

```
2      breakpoint     keep y   0x000000000e200c80
       silent
       print /x $x0
       set $pc=0xe200c7c
       set $x0=$a
       set $a=$a+4
       if $a<$b
         continue
       end
Continuing.

Breakpoint 1, 0x000000000e200c7c in ?? ()
$1 = "B="
$2 = 0x1bc
$3 = "A="
$4 = 0x0
$5 = 0x0
$6 = "======"
Continuing.
$7 = 0x100d0010
$8 = 0x100d0010
$9 = 0x204d00
$10 = 0xf40102d
$11 = 0x204f4010
$12 = 0x42f40
$13 = 0x410000f
$14 = 0x104c0010
$15 = 0x5c00
$16 = 0x2d0002b0
$17 = 0x10045000
$18 = 0x40104c40
$19 = 0xb040005c
$20 = 0x2d0002
$21 = 0x80100490

…..

$115 = 0x12c0002f
$116 = 0x9ec189
$117 = 0x40000010
$118 = 0xa5
$119 = "======"
```

NB : La clef entré en paramètre est stockée cryptée dans la zone mémoire : [0x100020 - 0x10003F]

2. Machine virtuelle

En continuant l'analyse du secureOS, on trouve que l'interpreteur de byte code est implementé par la fonction FUN_000005a4() (0xe2005a4).

Le code ci-dessous correspond à l'opcode 0.

```
                 switchD_0000095c::caseD_0                XREF[1]:     0000095c (j)
00000960 80 0e 40 92   and     x0,x20 ,#0xf
00000964 94 06 00 11   add     w20 ,w20 ,#0x1
00000968 73 3e 40 92   and     x19 ,x19 ,#0xffff
0000096c 03 00 80 d2   mov     x3,#0x0
00000970 02 00 80 d2   mov     x2,#0x0
00000974 e1 03 13 aa   mov     x1,x19
00000978 24 00 00 b0   adrp    x4,0x5000
0000097c 84 c0 08 91   add     x4=>DAT_00005230 ,x4,#0x230             = 01h
00000980 09 07 00 94   bl      FUN_000025a4                     undefined FUN_000025a4()
00000984 24 00 00 b0   adrp    x4,0x5000
00000988 03 00 80 d2   mov     x3,#0x0
0000098c 84 e0 01 91   add     x4=>DAT_00005078 ,x4,#0x78
00000990 02 00 80 d2   mov     x2,#0x0
00000994 01 00 80 d2   mov     x1,#0x0
00000998 e0 01 80 d2   mov     x0,#0xf
0000099c 02 07 00 94   bl      FUN_000025a4                     undefined FUN_000025a4()
```

Pour rendre le code plus compliqué à analyser l'interpreteur comporte du code AARCH32.

La fonction FUN_25a4() est utilisé pour basculer en AARCH32. L'adresse du code en 32 bits est mise dans le registre elr_el1.

```
000025a4 ff 83 02 d1   sub     sp,sp,#0xa0
000025a8 e0 07 00 a9   stp     x0,x1,[sp]=>local_a0
000025ac e2 0f 01 a9   stp     x2,x3,[sp, #local_90 ]
000025b0 e4 17 02 a9   stp     x4,x5,[sp, #local_80 ]
000025b4 e6 1f 03 a9   stp     x6,x7,[sp, #local_70 ]
000025b8 e8 27 04 a9   stp     x8,x9,[sp, #local_60 ]
000025bc ea 2f 05 a9   stp     x10 ,x11 ,[sp, #local_50 ]
000025c0 ec 37 06 a9   stp     x12 ,x13 ,[sp, #local_40 ]
000025c4 ee 3f 07 a9   stp     x14 ,x15 ,[sp, #local_30 ]
000025c8 f0 47 08 a9   stp     x16 ,x17 ,[sp, #local_20 ]
000025cc f2 7b 09 a9   stp     x18 ,x30 ,[sp, #local_10 ]
000025d0 24 40 18 d5   msr     elr_el1 ,x4
000025d4 e6 ff ff 97   bl      FUN_0000256c                     undefined FUN_0000256c()
000025d8 06 3a 80 52   mov     w6,#0x1d0
000025dc 06 40 18 d5   msr     spsr_el1 ,x6
000025e0 e0 07 40 a9   ldp     x0,x1,[sp]=>local_a0
000025e4 e2 0f 41 a9   ldp     x2,x3,[sp, #local_90 ]
000025e8 e4 17 42 a9   ldp     x4,x5,[sp, #local_80 ]
000025ec e6 1f 43 a9   ldp     x6,x7,[sp, #local_70 ]
000025f0 e8 27 44 a9   ldp     x8,x9,[sp, #local_60 ]
000025f4 ea 2f 45 a9   ldp     x10 ,x11 ,[sp, #local_50 ]
000025f8 ec 37 46 a9   ldp     x12 ,x13 ,[sp, #local_40 ]
000025fc ee 3f 47 a9   ldp     x14 ,x15 ,[sp, #local_30 ]
00002600 f0 47 48 a9   ldp     x16 ,x17 ,[sp, #local_20 ]
00002604 f2 7b 49 a9   ldp     x18 ,x30 ,[sp, #local_10 ]
00002608 ff 83 02 91   add     sp,sp,#0xa0
0000260c e0 03 9f d6   eret
```

Pour traiter l'opcode 0, l'interpreteur va invoquer le code 32 bits à l'adresse 0x5230.

```
00005230 01 20 a0 e1   cpy     r2,r1
```

```
00005234 00  10  a0  e1    cpy       r1,r0
00005238 01  03  08  e3    movw      r0,#0x8301
0000523c 00  08  a0  e1    mov       r0,r0, lsl #0x10
00005240 02  00  80  e2    add       r0,r0,#0x2
00005244 38  13  00  ef    swi       0x1338
00005248 37  13  00  ef    swi       0x1337
```

L'instruction SWI déclenche une exception software interrupt. Le handler (code en AARCH64) teste la valeur de de l'appel SWI. La valeur 0x1338 est utilisé pour invoquer un appel SMC. La valeur 0x1337 est utilisé pour retourner au code de l'appelant de la fonction FUN_25a4().

Ici la fonction 5230 est simplement utilisée pour appeler SMC(0x8301002, r0, r1) c'est-à-dire pour écrire la valeur r1 dans le registre r0.

La deuxième fonction AARCH32 appelée pour l'opcode 0 se trouve en 0x5078. Elle est utilisée pour avancer le pointeur d'instruction de 3 octets. SMC(0x8301001, 0xF) retourne le pointeur d'instruction, on ajoute 3 et SMC(0x8301002,0xF, r2) écrit la nouvelle valeur du pointeur d'instruction.

```
00005078 00  80  a0  e1    cpy       r8,r0
0000507c 00  10  a0  e1    cpy       r1,r0
00005080 01  03  08  e3    movw      r0,#0x8301
00005084 00  08  a0  e1    mov       r0,r0, lsl #0x10
00005088 01  00  80  e2    add       r0,r0,#0x1
0000508c 38  13  00  ef    swi       0x1338
00005090 00  10  a0  e1    cpy       r1,r0
00005094 03  10  81  e2    add       r1,r1,#0x3
00005098 08  00  a0  e1    cpy       r0,r8
0000509c 01  20  a0  e1    cpy       r2,r1
000050a0 00  10  a0  e1    cpy       r1,r0
000050a4 01  03  08  e3    movw      r0,#0x8301
000050a8 00  08  a0  e1    mov       r0,r0, lsl #0x10
000050ac 02  00  80  e2    add       r0,r0,#0x2
000050b0 38  13  00  ef    swi       0x1338
000050b4 37  13  00  ef    swi       0x1337
```

Après analyse des autres instructions de l’interpreteur on trouve :

La machine virtuelle implementée par le secure OS comporte :

16 registres de 32 bits : R[0] à R[15].

R[15] est le pointeur d’instruction.

Les instructions sont codées sur 3 octets.

opcode = (ins>>20)&0xFF

opc2 = (ins >> 18) & 3

r0 = (ins >> 14) & 0xF

r1 = (ins >> 10) & 0xF

p0 = ins & 0x3FFF

On a les instructions suivantes :

opcode	opc2	Opération
0	0	R[r0] := R[r1]
	1	R[r0] := MEM[R[r1]] // MEM désigne la zone mémoire protegée
	2	MEM[R[r0]] := R[r1]
	3	R[r0] := p0
1	-	R[r0] -= 1
2	0	R[r0] += R[r1]
	3	R[r0] += p0
3	0	R[r0] -= R[r1]
	3	R[r0] -= p0
4	3	R[r0] <<= p0
5	3	R[r0] >>= p0
6	0	R[r0] ^= R[r1]
7	3	R[r0] &= p0
8	-	R[15] = p0 // JUMP p0
9	-	If (R[r0] != 0) R[15] = p0 // JNZ p0
10	-	Return(R[0]) // EXIT
11	0	R[r0] := ((R[r0] <<8)&0xFF) | (R[r0] >> 8) // SWAP Bytes
12	-	SetKey(R[r0])
13	-	NOP
14	-	If ((*0x9010000>5) R[15] = p0 // ANTI-DEBUG

L’instruction 14 est une instruction Anti-debug. L’adresse 0x9010000 contient un compteur en secondes. Si le temps d’execution du programme dépasse 5 secondes, cette instruction en effectuant un JUMP va modifier le flot d’execution pour rendre la sortie du programme invalide.

L’instruction 12 est utilisé pour calculer la valeur attendue du programme (stocké en mémoire aux adresses 0x100000-0x10001F). Après avoir chiffré la clef donnée en paramètre au programme safe2, le programme compare le résultat chiffé avec la valeur attendue (mise à jour par l’instruction 12).

6. Byte code program

On peut maintenant désassembler le programme téléchargé dans le secureOS. On va utiliser pour cela le programme disass.py disponible en annexe.

```
R[0x4] = 0x10
R[0x4] <<= 0x10
R[0x4] += 0x20
R[0xd] = 0x10
R[0xd] <<= 0x10
R[0xd] += 0x20
R[0xc] = 0x4


        R[0x0] = MEM[R[0x4]]
        R[0x0] <<= 0x10
        R[0x0] >>= 0x10
        R[0x0] = (R[0x0] & 0xFF ) <<8) | (R[0x0] >> 8)

        R[0x4] += 0x2
        R[0x1] = MEM[R[0x4]]
        R[0x1] <<= 0x10
        R[0x1] >>= 0x10
        R[0x1] = (R[0x1] & 0xFF ) <<8) | (R[0x1] >> 8)

        R[0x4] += 0x2
        R[0x2] = MEM[R[0x4]]
        R[0x2] <<= 0x10
        R[0x2] >>= 0x10
        R[0x2] = (R[0x2] & 0xFF ) <<8) | (R[0x2] >> 8)

        R[0x4] += 0x2
        R[0x3] = MEM[R[0x4]]
        R[0x3] <<= 0x10
        R[0x3] >>= 0x10
        R[0x3] = (R[0x3] & 0xFF ) <<8) | (R[0x3] >> 8)


        R[0xe] = 0x20
        R[0x7] = 0x7
        NOP


                R[0xe] -= 1
                R[0x4] = R[0x1]
                R[0x5] = R[0x4]
                R[0x4] >>= 0x8
                R[0x4] &= 0xff
                R[0x5] &= 0xff
                R[0xb] = R[0x5]
                R[0xb] <<= 0x8

                R[0xa] = R[0x7]
                R[0xa] <<= 0x10
                R[0xa] += R[0xb]
                R[0xa] += R[0x4]
                R[0xa] += 0x1000

                R[0x6] = MEM[R[0xa]]
                R[0x6] &= 0xff

                if (R[0x7] == 0)
                        R[0x7] = 0xa
                R[0x7] -= 1

                R[0xb] = R[0x4]
```

```
R[0xb] <<= 0x8

R[0xa] = R[0x7]
R[0xa] <<= 0x10
R[0xa] += R[0xb]
R[0xa] += R[0x6]
R[0xa] += 0x1000

R[0x5] = MEM[R[0xa]]
R[0x5] &= 0xff

if (R[0x7] == 0)
        R[0x7] = 0xa
R[0x7] = 0xa
R[0x7] -= 1

R[0xb] = R[0x6]
R[0xb] <<= 0x8

R[0xa] = R[0x7]
R[0xa] <<= 0x10
R[0xa] += R[0xb]
R[0xa] += R[0x5]
R[0xa] += 0x1000

R[0x4] = MEM[R[0xa]]
R[0x4] &= 0xff

if (R[0x7] == 0)
        R[0x7] = 0xa
R[0x7] -= 1

R[0xb] = R[0x5]
R[0xb] <<= 0x8

R[0xa] = R[0x7]
R[0xa] <<= 0x10
R[0xa] += R[0xb]
R[0xa] += R[0x4]
R[0xa] += 0x1000

R[0x6] = MEM[R[0xa]]
R[0x6] &= 0xff

if (R[0x7] == 0)
        R[0x7] = 0xa
R[0x7] -= 1

R[0x9] = R[0x6]
R[0x9] <<= 0x8
R[0x9] += R[0x4]

R[0x8] = R[0xe]
R[0x8] >>= 0x3
R[0x8] &= 0x1

SetKey(R[0xe])

if (R[0x8] == 0)
        R[0x8] = R[0x3]
        R[0x3] = R[0xe]
        R[0x3] += 0x1
        R[0x3] ^= R[0x0]
        if (*0x9010000<=5)
                R[0x3] ^= R[0x1];
                R[0x0] = R[0x9];
                R[0x1] = R[0x2];
                R[0x2] = R[0x8];
else
        R[0x8] = R[0x0]
        R[0x0] = R[0x9]
```

```
                R[0x1] = R[0xe]
                R[0x1] += 0x1
                R[0x1] ^= R[0x0]
                R[0x1] ^= R[0x2]

                if (*0x9010000<=5)
                    R[0x2] = R[0x3];
                    R[0x3] = R[0x8];

            if (R[0xe] == 0) R[0xf]+=3 else R[0xf] = 0x57


        R[0x0] = (R[0x0] & 0xFF ) <<8) | (R[0x0] >> 8)
        R[0x1] = (R[0x1] & 0xFF ) <<8) | (R[0x1] >> 8)
        R[0x1] <<= 0x10
        R[0x0] += R[0x1]

        MEM[R[0xd]]= R[0x0]
        R[0xd] += 0x4

        R[0x2] = (R[0x2] & 0xFF ) <<8) | (R[0x2] >> 8)
        R[0x3] = (R[0x3] & 0xFF ) <<8) | (R[0x3] >> 8)
        R[0x3] <<= 0x10
        R[0x2] += R[0x3]
        MEM[R[0xd]]= R[0x2]

        R[0xd] += 0x4
        R[0x4] = R[0xd]
        R[0xc] -= 1
        if (R[0xc] == 0) R[0xf]+=3 else R[0xf] = 0x15


    R[0xc] = 0x10
    R[0xc] <<= 0x10
    R[0xb] = 0x20

    R[0xd] -= 0x20
    R[0x4] = 0x0


        R[0x0] = MEM[R[0xd]]
        R[0x0] &= 0xff
        R[0x1] = MEM[R[0xc]]
        R[0x1] &= 0xff
        R[0x0] -= R[0x1]
        if (R[0x0] != 0)
            R[0x4] = 0x1
        R[0xd] += 0x1
        R[0xc] += 0x1

        R[0xb] -= 1
        if (R[0xb] == 0) R[0xf]+=3 else R[0xf] = 0x189

    R[0x0] = R[0x4]
    R[0]; EXIT()
```

On peut maintenant écrire un programme C équivalent et inverser la fonction. (Programme Decrypt.c disponible en annexe).

Valeures attendues

Il est nécessaire de connaitre le résultat attendu de la fonction de chiffrement. Cette valeur est disponible dans la mémoire protégée entre les adresses 0x10 0000 et 0x10 001F après l'execution de la première partie du programme. Pour obtenir les valeurs déchiffrées, on va utiliser gdb. On place un point d'arrêt dans l'interpreteur sur le traitement de l'instruction de fin de programme (opcode :0xA) par exemple à l'adresse 0xe200704. Puis on utilise les commandes gdb pour détourner le code de lecture des instructions du bytecode (comme presenté au paragraphe 5.1).

```
1      breakpoint     keep y   0x000000000e200704
       breakpoint already hit 8 times
2      breakpoint     keep n   0x000000000e200c7c
       breakpoint already hit 2 times
3      breakpoint     keep n   0x000000000e200c80
       breakpoint already hit 102 times
       silent
       print /x $x0
       set $pc=0xe200c7c
       set $x0=$a
       set $a=$a+4
       if $a<$b
         continue
       end

(gdb) ena 2 3
(gdb) set $a=0x100000
(gdb) set $b=$a+0x44
(gdb) set $x0=$a
(gdb) set $pc=0x000000000e200c7c
```

On trouve les valeurs.

```
$8 = 0x612e7270
$9 = 0x6766722e
$10 = 0x666e632e
$11 = 0x2e76662e
$12 = 0x76706e73
$13 = 0x66407279
$14 = 0x70766766
$15 = 0x7465622e
```

En ASCII ces valeurs donnent : « pr.a.rfg.cnf.fv.snpvyr@ffgvp.bet »

Soit après ROT13 « ce.n.est.pas.si.facile@sstic.org » !

Lookup Table

Il reste un dernier problème à regler. Le programme utilise une lookup table dans l'algorithme de chiffrement. Cette table est constituée des données qui se trouvent à la suite du programme dans le bloc de données de 0x101010 octets chargés dans le SecureOS.

Hors on a vu que ces données ne sont pas correctement mappées dans l'espace mémoire du SecureOS (il manque la deuxième page de 4K). C'est une protection anti debug qui est activée quand qemu est lancé avec l'option gdb.

(NB : Les données sont correctement mappés dans l'espace du Secure Monitor où elles ont été recopiées par l'appel SMC(0x83010004)).

A cause de ce décalage, la table de lookup decryptée est fausse et la fonction de chiffrement n'est pas inversible.

Il faudrait regarder comment sont configurées les pages tables referencés par le registre ttbr0_el1.

(Mais je n'ai pas eu le temps et ni le courage de chercher).

La solution la plus simple que j'ai trouvé a été de crée un fichier decrypted_file_hacked avec la table de lookup deplacée de 4K vers la fin. (On va perdre les 4K à la fin de la table mais elles ne sont pas utilsées par le programme).

```
dd if=decrypted_file of=file_enc.bin bs=1 skip=318424 count=1052688
dd if=file_enc.bin of=decrypted_file_hacked bs=1 skip=4096 seek=326616 count=1044496
dd if=decrypted_file of=decrypted_file_hacked bs=1 skip=1371112 seek=1371112 count=161320
```

Maintenant avec le programme decrypted_file_hacked, la table de lookup apparait correctement dans l'espace mémoire du secureOS. On peut extraire la table décryptée et utiliser le programme de déchiffrement.

Après avoir regler ce dernier problème, le programme decrypt.c permet d'obtenir le flag du programme safe_02.

SSTIC{acadaa8b5b55306fb3c6dfc3b2d1c80770084644225febd71a9189aa26ec740e}

```
============================
dkeys0= 8baaadac
dkeys1= 6f30555b
dkeys2= c3dfc6b3
dkeys3= 7c8d1b2
dkeys4= 44460870
dkeys5= d7eb5f22
dkeys6= aa89911a
dkeys7= e74ec26

acadaa8b 5b55306f b3c6dfc3 b2d1c807 70084644 225febd7 1a9189aa 26ec740e

acadaa8b5b55306fb3c6dfc3b2d1c80770084644225febd71a9189aa26ec740e

# ./add_key.py acadaa8b5b55306fb3c6dfc3b2d1c80770084644225febd71a9189aa26ec740e
[+] Key with key_id 00000004 ok
[+] Key added into keystore
[+] Envoyez le flag SSTIC{acadaa8b5b55306fb3c6dfc3b2d1c80770084644225febd71a9189aa26ec740e} à l'adresse challenge2019@sstic.org
pour valider votre avancée
[+] Container /root/safe_03/.encrypted decrypted to /root/safe_03/decrypted_file
```

6. Android image

Le fichier decrypted_file dans safe_03 contient une image Android.

Après quelques recherches, on finit par trouver l'adresse email recherché dans le fichier

`data/com.google.android.apps.messaging/databases/bugle_db`

```
$ sqlite3 data/com.google.android.apps.messaging/databases/bugle_db
SQLite version 3.11.0 2016-02-15 17:29:24
Enter ".help" for usage hints.
sqlite> select snippet_text from conversations;
Bien noté, je pense cibler le stand des huîtres pour plus de discrétion et d'efficacité.
Link previews are on.<br />Learn more or turn off in <u>Settings</u>.
Bonjour Kévin. Désolé finalement nous n'avons plus besoin de vous le 13/06/2018, car nous avons déjà trop de personnel. Nous vous
recontacterons pour une future mission.
Agence Interim'expert
Bien reçu. Bonne journée.
Bravo. Ce billet "Pentest et pentesteur" nous a paru un peu gros lors de la publication, mais cela semble marcher au-delà de nos
espérances. Continuez à alimenter la conversation, nous avons déjà constaté une perte d'attention chez la plupart des gens visés.
Mission accomplie, comme d'habitude la perspective d'une tournée de shooter au cactus a suffi à corrompre le CO et à chosir la date de
publication du challenge. Pour être sûr que les experts sont toujours occupés, j'ai redirigé l'adresse
9e915a63d3c4d57eb3da968570d69e95@challenge.sstic.org vers votre boîte mail. Tant que vous ne voyez passer aucun mail, la voie est
libre...
sqlite>
```

On trouve l'adresse email : 9e915a63d3c4d57eb3da968570d69e95@challenge.sstic.org.

7. Annexes

1. Find_key.py

```
import matplotlib.pyplot as plt
import numpy as np


def compute_diff(data, dt_strt, dt_lg, ref_st, ref_lg):
		res = np.zeros(dt_lg, dtype=float)
		for i in range(dt_strt, dt_strt+ dt_lg):
				delta = 0.0
				for j in range(0, ref_lg):
						delta = delta + abs(data[i+j] - data[ref_st + j])

				res[i-dt_strt] = delta

		return (res)


def find_key(diffs):
		recovered_key = 0x0000
		bitnum = 1024

		diffs = np.array(diffs)
		loc = np.where(diffs < 3)

		#Get actual list
		loc = loc[0]

		print len(loc)

		for i in range(0, len(loc)-1):
				delta = loc[i+1]-loc[i]
				bitnum -= 1
				#print delta

				#if delta > 700:
				if delta > 800:
						recovered_key |= 1<<bitnum

		print("Key = %04x"%recovered_key)

		return recovered_key


def bytes_to_int(bytes):
   result = 0
   for b in bytes:
      result = result * 256 + int(b)

   return result

def int_to_bytes(value, length):
   result = []

   for i in range(0, length):
      result.append((value >> (i * 8)) & 0xff)
   #result.reverse()

   return result

def swap_byte(b):
		res = 0
		for i in range(0,8):
				v = b & 1
				b = b >> 1
```

```
                    res = res <<1
                    res |= v
          return (res)

data = np.load('arr_0.npy')






d_start = 700000
d_length = 800000

r_start = 710000 + 750
r_length = 170



diffs = compute_diff(data, d_start, d_length, r_start, r_length)

k = find_key(diffs)

kb = int_to_bytes(k, 128)
print kb
for i in range(0,len(kb)):
          print "%02X"%swap_byte(kb[i]),
print
```

2. Secure_device

```
#include <stdio.h>
#include <string.h>
#include "sha256.h"


unsigned char secure_device_int(unsigned char A, unsigned char B, unsigned char op, unsigned char buttons)
{
          int bt4, bt1, bt2, bt3;
          int op0, op1;

          unsigned char IA, IB;

          unsigned char out;

          bt1 = buttons & 1;
          bt2 = buttons & 2;
          bt3 = buttons & 4;
          bt4 = buttons & 8;

          op0 = op & 1;
          op1 = (op & 2)>>1;

          if (bt3 != 0)
                    IA = ((A <<1) | (A >>7 )) & 0xFF ;
                    //IA = (((A <<1)&0xFF) | ((A >>7 )&0xFF)) & 0xFF ;
          else
                    IA = A;



          if (bt4 != 0)
                    IB = ((B <<1) | (B >>7 )) & 0xFF ;
                    //IB = (((B <<1)&0xFF) | ((B >>7 )&0xFF)) & 0xFF ;
          else
                    IB = B;
```

```
	if (bt1 != 0)
		op0 = 1- op0;

	if (bt2 != 0)
		op1 = 1- op1;

	if (op0 != 0 && op1 == 0)
		out = IA | IB;
	else if (op0 == 0 && op1 == 0)
		out = IA & IB;
	else if (op0 == 0 && op1 != 0)
		out = IA ^ IB;
	else {
		out = IA + IB;
	}

	return (out);
}
/*******************************/
unsigned char g_buttons=0;
//unsigned char g_buttons=0x0F;
/*******************************/
unsigned char secure_device(unsigned char A, unsigned char B, unsigned char op)
{
	unsigned char res;

	res = secure_device_int( A,  B, op, g_buttons);

	//printf("%x\n",res);
	return(res);
}
/*******************************/
unsigned char init()
{
	unsigned char r;

	r = secure_device(0x46,0x92,0);
	r = secure_device(0xdf,r,2);
	r = secure_device(0x3e,r,0);
	r = secure_device(0x3a,r,3);
	r = secure_device(0x36,r,2);
	r = secure_device(0x8e,r,2);
	r = secure_device(0xc9,r,3);
	r = secure_device(0xe7,r,1);
	r = secure_device(0x29,r,2);
	r = secure_device(0xc2,r,2);
	r = secure_device(0x79,r,0);
	r = secure_device(0x2a,r,2);
	r = secure_device(0x4c,r,3);
	r = secure_device(0xde,r,0);
	r = secure_device(0x88,r,0);
	r = secure_device(0x8b,r,2);
	r = secure_device(0x97,r,3);
	r = secure_device(0x6a,r,2);
	r = secure_device(0x60,r,1);
	r = secure_device(0x0f,r,0);
	r = secure_device(0x5b,r,3);
	r = secure_device(0xd0,r,2);
	r = secure_device(0xa9,r,1);
	r = secure_device(0xe3,r,3);
	r = secure_device(0xd0,r,1);
	r = secure_device(0x27,r,0);
	r = secure_device(0x90,r,0);
	r = secure_device(0x3b,r,1);
	r = secure_device(0x66,r,2);
	r = secure_device(0xe2,r,0);
	r = secure_device(0x24,r,3);
	r = secure_device(0xee,r,1);
	r = secure_device(0xf2,r,3);
	return r;
```

```
}
/*******************************/
unsigned char step1()
{
        unsigned char r;

        r = secure_device(0x35,0x27,3);
        r = secure_device(0x7e,r,3);
        r = secure_device(0x66,r,2);
        r = secure_device(0x8,r,1);
        r = secure_device(0x13,r,0);
        r = secure_device(0x1f,r,1);
        r = secure_device(0xa,r,2);
        r = secure_device(0xd3,r,0);
        r = secure_device(0xc6,r,3);

        return r;
}
unsigned char step2()
{
        unsigned char r;

        r= secure_device(0xde,0xab,0);
        r= secure_device(0x67,r,3);
        r= secure_device(0x2a,r,2);
        r= secure_device(0x6d,r,1);
        r= secure_device(0x4a,r,3);
        r= secure_device(0xe7,r,0);
        r= secure_device(0x1c,r,1);
        r= secure_device(0x35,r,0);
        r= secure_device(0xde,r,3);
        r= secure_device(0xf7,r,0);
        r= secure_device(0xda,r,2);
        return r;
}

unsigned char step3()
{
        unsigned char r;

        r = secure_device(0x14,0x23,3);
        r = secure_device(0x72,r,0);
        r = secure_device(0x48,r,3);
        r = secure_device(0x53,r,1);
        r = secure_device(0xa7,r,0);
        r = secure_device(0x5f,r,1);
        r = secure_device(0x3,r,3);
        r = secure_device(0xb7,r,3);
        r = secure_device(0x73,r,1);
        r = secure_device(0x37,r,3);
        r = secure_device(0xc5,r,2);
        r = secure_device(0xa4,r,1);
        r = secure_device(0x30,r,0);
        r = secure_device(0xdd,r,2);
        return r;
}

unsigned char step4()
{
        unsigned char r;

        r = secure_device(0xb0,0x42,2);
        r = secure_device(0xbc,r,2);
        r = secure_device(0xfc,r,2);
        r = secure_device(0x54,r,3);
        r = secure_device(0x30,r,2);
        r = secure_device(0x97,r,1);
        r = secure_device(0xe8,r,2);
        r = secure_device(0xd6,r,0);
        r = secure_device(0x26,r,0);
        r = secure_device(0xeb,r,0);
```

```
	r = secure_device(0x68,r,1);
	r = secure_device(0x26,r,0);
	r = secure_device(0x9,r,3);
	r = secure_device(0x2a,r,2);
	r = secure_device(0xa9,r,3);
	return r;
}

unsigned char step5()
{
	unsigned char r;

	r = secure_device(0xff,0x12,0);
	r = secure_device(0xfd,r,1);
	r = secure_device(0xe5,r,1);
	r = secure_device(0x26,r,3);
	r = secure_device(0x85,r,3);
	r = secure_device(0x63,r,1);
	r = secure_device(0x93,r,3);
	r = secure_device(0xba,r,2);
	r = secure_device(0x97,r,0);
	r = secure_device(0xab,r,1);
	r = secure_device(0x6e,r,3);
	r = secure_device(0xfd,r,0);
	r = secure_device(0x4c,r,3);
	r = secure_device(0x50,r,0);
	r = secure_device(0xa,r,2);
	r = secure_device(0xfc,r,3);
	r = secure_device(0xe3,r,2);
	r = secure_device(0xa6,r,3);
	r = secure_device(0x64,r,2);
	r = secure_device(0x8e,r,3);
	r = secure_device(0xc1,r,1);
	return r;
}

unsigned char step6()
{
	unsigned char r;

	r = secure_device(0x90,0x77,1);
	r = secure_device(0x8e,r,0);
	r = secure_device(0xbd,r,2);
	r = secure_device(0x39,r,2);
	r = secure_device(0x4c,r,2);
	r = secure_device(0xc5,r,2);
	r = secure_device(0xb6,r,3);
	r = secure_device(0x93,r,1);
	r = secure_device(0x9f,r,3);
	r = secure_device(0xd6,r,3);
	r = secure_device(0x6e,r,2);
	r = secure_device(0x39,r,3);
	r = secure_device(0x40,r,1);
	r = secure_device(0x14,r,2);
	r = secure_device(0xe6,r,3);
	return r;
}

unsigned char step7()
{
	unsigned char r;

	r = secure_device(0xf,0xab,3);
	r = secure_device(0xa2,r,1);
	r = secure_device(0x7c,r,0);
	r = secure_device(0x34,r,1);
	r = secure_device(0x14,r,1);
	r = secure_device(0xe7,r,0);
	r = secure_device(0xb9,r,0);
	r = secure_device(0xf1,r,2);
	r = secure_device(0xd5,r,1);
```

```
	r = secure_device(0x4e,r,2);
	r = secure_device(0xe,r,2);
	r = secure_device(0x6,r,0);
	r = secure_device(0x7d,r,2);
	r = secure_device(0x87,r,3);
	r = secure_device(0xbc,r,0);
	r = secure_device(0xd4,r,3);
	r = secure_device(0x8a,r,1);
	r = secure_device(0xe7,r,3);
	r = secure_device(0x9e,r,1);
	r = secure_device(0x58,r,0);
	r = secure_device(0x24,r,2);
	r = secure_device(0x44,r,3);
	r = secure_device(0xc9,r,1);
	r = secure_device(0xd4,r,1);
	r = secure_device(0x1d,r,3);
	r = secure_device(0xcd,r,0);
	r = secure_device(0xde,r,1);
	r = secure_device(0x54,r,0);
	r = secure_device(0x5e,r,2);
	r = secure_device(0x46,r,1);
	r = secure_device(0x21,r,0);
	r = secure_device(0xff,r,1);
	r = secure_device(0x51,r,0);
	r = secure_device(0x78,r,1);
	r = secure_device(0x2f,r,3);
	r = secure_device(0xed,r,2);
	r = secure_device(0x4b,r,3);
	r = secure_device(0x4d,r,2);
	return r;
}

unsigned char step8()
{
	unsigned char r;

	r = secure_device(0x88,0x74,0);
	r = secure_device(0x48,r,2);
	r = secure_device(0x11,r,2);
	r = secure_device(0x76,r,0);
	r = secure_device(0x2b,r,3);
	r = secure_device(0xf8,r,2);
	return r;
}
/*******************************/
unsigned char kelts[8][16];

int build_keyelts()
{
	int i;

	for (i=0; i <16; i++) {
		g_buttons = i;
		kelts[0][i] = step1();
	}
	for (i=0; i <16; i++) {
		g_buttons = i;
		kelts[1][i] = step2();
	}
	for (i=0; i <16; i++) {
		g_buttons = i;
		kelts[2][i] = step3();
	}
	for (i=0; i <16; i++) {
		g_buttons = i;
		kelts[3][i] = step4();
	}
	for (i=0; i <16; i++) {
		g_buttons = i;
		kelts[4][i] = step5();
	}
```

```
		for (i=0; i <16; i++) {
			g_buttons = i;
			kelts[5][i] = step6();
		}
		for (i=0; i <16; i++) {
			g_buttons = i;
			kelts[6][i] = step7();
		}
		for (i=0; i <16; i++) {
			g_buttons = i;
			kelts[7][i] = step8();
		}

}
/*****/
int show_keyelts()
{
		int i,j;
		for (i=0; i<8; i++) {
			for (j=0; j<16; j++) {
				printf("%02X ",kelts[i][j]);
			}
			printf("\n");
		}
}
/*****/
int inc_tab(int *idx)
{
		int i,j;
		int res=0;

		i=0;
		while (idx[i] == 15 && i <8)
			i++;

		if (i>5) {
			for (j=0; j<8; j++)
				printf("%d ",idx[j]);
			printf("\n");
		}

		if (i<8) {
			idx[i] ++;
			for (j=0; j<i; j++)
				idx[j] = 0;
		}
		else
			res = 1;

		return (res);
}
/*****/
unsigned char hash_ref[SHA256_BLOCK_SIZE]={
0x00,0xc8,0xbb,0x35,0xd4,0x4d,0xcb,0xb2,0x71,0x2a,0x11,0x79,0x9d,0x8e,0x13,0x16,0x04,0x5d,0x64,0x40,0x4f,0x33,0x7f,0x4f,0xf6,0x53,
0xc2,0x76,0x07,0xf4,0x36,0xea  };
/*****/
int check_key(int *idx)
{
		int i;
		unsigned char key[8];
    SHA256_CTX ctx;
		unsigned char hash[SHA256_BLOCK_SIZE];
		int pass;

		for (i=0; i<8; i++)
			key[i] = kelts[i][idx[i]];


    sha256_init(&ctx);
    sha256_update(&ctx, key, 8);
    sha256_final(&ctx, hash);
```

```
	pass = !memcmp(hash_ref, hash, SHA256_BLOCK_SIZE);
		if (pass == 1) {
			printf("Key Found\n");
			for (i=0; i<8; i++)
				printf("%02X ",key[i]);
			printf("\n");
		}

		return(pass);

}
/*****/
int brute_force()
{
		int idx[8];
		int j;
		int fin;
		int found;

		for (j=0; j<8; j++)
			idx[j] = 0;


		do {
			found = check_key(idx);
			if (found)
				break;
			fin = inc_tab(idx);
		} while(!fin) ;


}
/*******************************/
int main()
{
		unsigned char res;

		g_buttons=0xFF;
		res = init();
		printf("res=0x%02x\n", res);

		g_buttons=0;
		res = init();
		printf("res=0x%02x\n", res);

		build_keyelts();
		show_keyelts();
		brute_force();
}
```

3. dwarf_disass

```
char * inst_names[256];

char ins_undef[]= "UNDEF";

#define DW_OP(name, opcode)  inst_names[opcode]=name;
#define DW_OP_DUP(name, opcode)

load_inst() {

int i;
for (i=0; i<256; i++) {
		inst_names[i] = ins_undef;
}
```

```
DW_OP ("DW_OP_addr,", 0x03)
DW_OP ("DW_OP_deref,", 0x06)
DW_OP ("DW_OP_const1u,", 0x08)
DW_OP ("DW_OP_const1s,", 0x09)
DW_OP ("DW_OP_const2u,", 0x0a)
DW_OP ("DW_OP_const2s,", 0x0b)
DW_OP ("DW_OP_const4u,", 0x0c)
DW_OP ("DW_OP_const4s,", 0x0d)
DW_OP ("DW_OP_const8u,", 0x0e)
DW_OP ("DW_OP_const8s,", 0x0f)
DW_OP ("DW_OP_constu,", 0x10)
DW_OP ("DW_OP_consts,", 0x11)
DW_OP ("DW_OP_dup,", 0x12)
DW_OP ("DW_OP_drop,", 0x13)
DW_OP ("DW_OP_over,", 0x14)
DW_OP ("DW_OP_pick,", 0x15)
DW_OP ("DW_OP_swap,", 0x16)
DW_OP ("DW_OP_rot,", 0x17)
DW_OP ("DW_OP_xderef,", 0x18)
DW_OP ("DW_OP_abs,", 0x19)
DW_OP ("DW_OP_and,", 0x1a)
DW_OP ("DW_OP_div,", 0x1b)
DW_OP ("DW_OP_minus,", 0x1c)
DW_OP ("DW_OP_mod,", 0x1d)
DW_OP ("DW_OP_mul,", 0x1e)
DW_OP ("DW_OP_neg,", 0x1f)
DW_OP ("DW_OP_not,", 0x20)
DW_OP ("DW_OP_or,", 0x21)
DW_OP ("DW_OP_plus,", 0x22)
DW_OP ("DW_OP_plus_uconst,", 0x23)
DW_OP ("DW_OP_shl,", 0x24)
DW_OP ("DW_OP_shr,", 0x25)
DW_OP ("DW_OP_shra,", 0x26)
DW_OP ("DW_OP_xor,", 0x27)
DW_OP ("DW_OP_bra,", 0x28)
DW_OP ("DW_OP_eq,", 0x29)
DW_OP ("DW_OP_ge,", 0x2a)
DW_OP ("DW_OP_gt,", 0x2b)
DW_OP ("DW_OP_le,", 0x2c)
DW_OP ("DW_OP_lt,", 0x2d)
DW_OP ("DW_OP_ne,", 0x2e)
DW_OP ("DW_OP_skip,", 0x2f)
DW_OP ("DW_OP_lit0,", 0x30)
DW_OP ("DW_OP_lit1,", 0x31)
DW_OP ("DW_OP_lit2,", 0x32)
DW_OP ("DW_OP_lit3,", 0x33)
DW_OP ("DW_OP_lit4,", 0x34)
DW_OP ("DW_OP_lit5,", 0x35)
DW_OP ("DW_OP_lit6,", 0x36)
DW_OP ("DW_OP_lit7,", 0x37)
DW_OP ("DW_OP_lit8,", 0x38)
DW_OP ("DW_OP_lit9,", 0x39)
DW_OP ("DW_OP_lit10,", 0x3a)
DW_OP ("DW_OP_lit11,", 0x3b)
DW_OP ("DW_OP_lit12,", 0x3c)
DW_OP ("DW_OP_lit13,", 0x3d)
DW_OP ("DW_OP_lit14,", 0x3e)
DW_OP ("DW_OP_lit15,", 0x3f)
DW_OP ("DW_OP_lit16,", 0x40)
DW_OP ("DW_OP_lit17,", 0x41)
DW_OP ("DW_OP_lit18,", 0x42)
DW_OP ("DW_OP_lit19,", 0x43)
DW_OP ("DW_OP_lit20,", 0x44)
DW_OP ("DW_OP_lit21,", 0x45)
DW_OP ("DW_OP_lit22,", 0x46)
DW_OP ("DW_OP_lit23,", 0x47)
DW_OP ("DW_OP_lit24,", 0x48)
DW_OP ("DW_OP_lit25,", 0x49)
DW_OP ("DW_OP_lit26,", 0x4a)
DW_OP ("DW_OP_lit27,", 0x4b)
DW_OP ("DW_OP_lit28,", 0x4c)
```

```
DW_OP ("DW_OP_lit29,", 0x4d)
DW_OP ("DW_OP_lit30,", 0x4e)
DW_OP ("DW_OP_lit31,", 0x4f)
DW_OP ("DW_OP_reg0,", 0x50)
DW_OP ("DW_OP_reg1,", 0x51)
DW_OP ("DW_OP_reg2,", 0x52)
DW_OP ("DW_OP_reg3,", 0x53)
DW_OP ("DW_OP_reg4,", 0x54)
DW_OP ("DW_OP_reg5,", 0x55)
DW_OP ("DW_OP_reg6,", 0x56)
DW_OP ("DW_OP_reg7,", 0x57)
DW_OP ("DW_OP_reg8,", 0x58)
DW_OP ("DW_OP_reg9,", 0x59)
DW_OP ("DW_OP_reg10,", 0x5a)
DW_OP ("DW_OP_reg11,", 0x5b)
DW_OP ("DW_OP_reg12,", 0x5c)
DW_OP ("DW_OP_reg13,", 0x5d)
DW_OP ("DW_OP_reg14,", 0x5e)
DW_OP ("DW_OP_reg15,", 0x5f)
DW_OP ("DW_OP_reg16,", 0x60)
DW_OP ("DW_OP_reg17,", 0x61)
DW_OP ("DW_OP_reg18,", 0x62)
DW_OP ("DW_OP_reg19,", 0x63)
DW_OP ("DW_OP_reg20,", 0x64)
DW_OP ("DW_OP_reg21,", 0x65)
DW_OP ("DW_OP_reg22,", 0x66)
DW_OP ("DW_OP_reg23,", 0x67)
DW_OP ("DW_OP_reg24,", 0x68)
DW_OP ("DW_OP_reg25,", 0x69)
DW_OP ("DW_OP_reg26,", 0x6a)
DW_OP ("DW_OP_reg27,", 0x6b)
DW_OP ("DW_OP_reg28,", 0x6c)
DW_OP ("DW_OP_reg29,", 0x6d)
DW_OP ("DW_OP_reg30,", 0x6e)
DW_OP ("DW_OP_reg31,", 0x6f)
DW_OP ("DW_OP_breg0,", 0x70)
DW_OP ("DW_OP_breg1,", 0x71)
DW_OP ("DW_OP_breg2,", 0x72)
DW_OP ("DW_OP_breg3,", 0x73)
DW_OP ("DW_OP_breg4,", 0x74)
DW_OP ("DW_OP_breg5,", 0x75)
DW_OP ("DW_OP_breg6,", 0x76)
DW_OP ("DW_OP_breg7,", 0x77)
DW_OP ("DW_OP_breg8,", 0x78)
DW_OP ("DW_OP_breg9,", 0x79)
DW_OP ("DW_OP_breg10,", 0x7a)
DW_OP ("DW_OP_breg11,", 0x7b)
DW_OP ("DW_OP_breg12,", 0x7c)
DW_OP ("DW_OP_breg13,", 0x7d)
DW_OP ("DW_OP_breg14,", 0x7e)
DW_OP ("DW_OP_breg15,", 0x7f)
DW_OP ("DW_OP_breg16,", 0x80)
DW_OP ("DW_OP_breg17,", 0x81)
DW_OP ("DW_OP_breg18,", 0x82)
DW_OP ("DW_OP_breg19,", 0x83)
DW_OP ("DW_OP_breg20,", 0x84)
DW_OP ("DW_OP_breg21,", 0x85)
DW_OP ("DW_OP_breg22,", 0x86)
DW_OP ("DW_OP_breg23,", 0x87)
DW_OP ("DW_OP_breg24,", 0x88)
DW_OP ("DW_OP_breg25,", 0x89)
DW_OP ("DW_OP_breg26,", 0x8a)
DW_OP ("DW_OP_breg27,", 0x8b)
DW_OP ("DW_OP_breg28,", 0x8c)
DW_OP ("DW_OP_breg29,", 0x8d)
DW_OP ("DW_OP_breg30,", 0x8e)
DW_OP ("DW_OP_breg31,", 0x8f)
DW_OP ("DW_OP_regx,", 0x90)
DW_OP ("DW_OP_fbreg,", 0x91)
DW_OP ("DW_OP_bregx,", 0x92)
DW_OP ("DW_OP_piece,", 0x93)
```

```
DW_OP ("DW_OP_deref_size,", 0x94)
DW_OP ("DW_OP_xderef_size,", 0x95)
DW_OP ("DW_OP_nop,", 0x96)
/* DWARF 3 extensions.  */
DW_OP ("DW_OP_push_object_address,", 0x97)
DW_OP ("DW_OP_call2,", 0x98)
DW_OP ("DW_OP_call4,", 0x99)
DW_OP ("DW_OP_call_ref,", 0x9a)
DW_OP ("DW_OP_form_tls_address,", 0x9b)
DW_OP ("DW_OP_call_frame_cfa,", 0x9c)
DW_OP ("DW_OP_bit_piece,", 0x9d)

/* DWARF 4 extensions.  */
DW_OP ("DW_OP_implicit_value,", 0x9e)
DW_OP ("DW_OP_stack_value,", 0x9f)

/* DWARF 5 extensions.  */
DW_OP ("DW_OP_implicit_pointer,", 0xa0)
DW_OP ("DW_OP_addrx,", 0xa1)
DW_OP ("DW_OP_constx,", 0xa2)
DW_OP ("DW_OP_entry_value,", 0xa3)
DW_OP ("DW_OP_const_type,", 0xa4)
DW_OP ("DW_OP_regval_type,", 0xa5)
DW_OP ("DW_OP_deref_type,", 0xa6)
DW_OP ("DW_OP_xderef_type,", 0xa7)
DW_OP ("DW_OP_convert,", 0xa8)
DW_OP ("DW_OP_reinterpret,", 0xa9)


/* GNU extensions.  */
DW_OP ("DW_OP_GNU_push_tls_address,", 0xe0)
/* The following is for marking variables that are uninitialized.  */
DW_OP ("DW_OP_GNU_uninit,", 0xf0)
DW_OP ("DW_OP_GNU_encoded_addr,", 0xf1)
DW_OP ("DW_OP_GNU_implicit_pointer,", 0xf2)
DW_OP ("DW_OP_GNU_entry_value,", 0xf3)
DW_OP ("DW_OP_GNU_const_type,", 0xf4)
DW_OP ("DW_OP_GNU_regval_type,", 0xf5)
DW_OP ("DW_OP_GNU_deref_type,", 0xf6)
DW_OP ("DW_OP_GNU_convert,", 0xf7)
DW_OP ("DW_OP_GNU_reinterpret,", 0xf9)
DW_OP ("DW_OP_GNU_parameter_ref,", 0xfa)
DW_OP ("DW_OP_GNU_addr_index,", 0xfb)
DW_OP ("DW_OP_GNU_const_index,", 0xfc)
DW_OP ("DW_OP_GNU_variable_value,", 0xfd)
DW_OP_DUP ("DW_OP_HP_unknown", 0xe0)
DW_OP ("DW_OP_HP_is_value,", 0xe1)
DW_OP ("DW_OP_HP_fltconst4,", 0xe2)
DW_OP ("DW_OP_HP_fltconst8,", 0xe3)
DW_OP ("DW_OP_HP_mod_range,", 0xe4)
DW_OP ("DW_OP_HP_unmod_range,", 0xe5)
DW_OP ("DW_OP_HP_tls,", 0xe6)
DW_OP ("DW_OP_PGI_omp_thread_num,", 0xf8)
DW_OP ("DW_OP_AARCH64_operation,", 0xea)
}
```

```
#include <stdio.h>

#include "dwarf_disass.h"
#include "dwarf_oplg.h"

#include "dec_dwarf_code.h"
```

```
#define MAXSZ 2048
unsigned char code[MAXSZ];



/************************************/
int disas(unsigned char *cde, int lg)
{
        int ptr=0;
        int opcode;
        int lg_oper;
        int cnt=0;
        int i;
        short of7;
        int jump_addr;

        while (ptr < lg) {

                opcode = cde[ptr];
                lg_oper = op_lg[opcode];

                printf("%03d: [%03d] 0x%02x, %s ", cnt, ptr, opcode, inst_names[opcode]);
                cnt++;

                ptr++;
                for (i=0; i<lg_oper; i++) {
                        printf("%02X ",cde[ptr +i]);
                }
                if ((opcode == 0x28 ) || (opcode == 0x2f)) {
                        of7 = cde[ptr+1] * 256 + cde[ptr];
                        jump_addr = ptr + of7 + lg_oper;
                        printf("[Jump_addr= %03d] ",jump_addr);
                }
                printf("\n");
                ptr += lg_oper;



        }

}
/************************************/
int main( int argc, char *argv[])
{

        load_inst();
        load_ins_lg();

        disas(prog2, sizeof(prog2));

}
```

4. InvHashC

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


#include "../dec_dwarf_code.h"

typedef unsigned long long uint64;

unsigned int * LUT3 = (unsigned int *) (prog2+0x4006B4-0x400258);

unsigned int * LUT0 = (unsigned int *) (prog2+0x400678-0x400258);
```

```
unsigned int * LUT2 = (unsigned int *) (prog2+0x400648-0x400258);

#define ROTL(A, r) ((((A)<<r) | ((A)>>(32-r))) & 0xFFFFFFFF)
#define ROTR(A, r) ((((A)>>r) | ((A)<<(32-r))) & 0xFFFFFFFF)

/*********************************************/
int Inv_fun_769(uint64 *A, uint64 B, uint64 C, uint64 D, uint64 res )
{
		uint64 AL, AH;
		uint64 VL, VH;
		uint64 V;

		printf("\tres=%llx\n",res);
		printf("\tB=%llx\n",B);
		printf("\tC=%llx\n",C);
		printf("\tD=%llx\n",D);


		res ^= B;
		VL = res & 0xFFFFFFFF;
		VH = (res>>32);


		C &= 0xFFFFFFFF;
		D &= 0xFFFFFFFF;
	AL = ROTL(VL , 6) ^D;
		AH = ROTL(VH, 14) ^ C ^ D;

		*A = (AH << 32) | AL;

		printf("\tA=%llx\n",*A);

}
/*********************************************/
int fun_769(uint64 A, uint64 B, uint64 C, uint64 D, uint64 *res )
{
		uint64 AL, AH;
		uint64 V;

		AL = A & 0xFFFFFFFF;
		AH = (A>>32);

		V = B ^ (ROTR(AL ^ D, 6) | (ROTR(AH ^ C ^ D, 14) <<32)) ;

		*res = V;
}
/*********************************************/
/*********************************************/
int Inv_fun_766 (uint64 *A, uint64 B, uint64 C, uint64 D, uint64 res)
{
		uint64 AL, AH;
		uint64 VL, VH;

		res ^= B;

		VL = res & 0xFFFFFFFF;
		VH = (res>>32);

		C &= 0xFFFFFFFF;
		D &= 0xFFFFFFFF;

		AL = ROTR (VL ^ D, 4) ^ C;
		AH = ROTR(VH ^ C, 14) ^ D ;

		*A = (AH << 32) | AL;

}
/*********************************************/
int fun_766 (uint64 A, uint64 B, uint64 C, uint64 D, uint64 *resA, uint64 *resB, uint64 *res)
{
		uint64 AL, AH;
```

```
	uint64 V;

	AL = A & 0xFFFFFFFF;
	AH = (A>>32);



	V = B ^ ( ROTL(AL ^ C, 4)  ^ D  | ((ROTL(AH ^ D, 14) ^ C)<<32)  );


	*resA = V & 0xFFFFFFFF;
	*resB = V >>32;
	*res = V;

}
/*********************************************/
/*********************************************/
int Inv_fun_755 (uint64 *pA, uint64 *pB, int cpt)
{
	uint64 A,B;
	uint64 VH,VL;
	uint64 C0, C1;

	VH = *pA;  //  (C1 | VL) ^ B
	VL = *pB;  //  (A ^ (C0 +B)))


	C0 = LUT2[2*cpt];
	C1 = LUT2[2*cpt+1];
	B = (C1 | VL ) ^ VH ;
	A = (C0 + B) ^ VL ;

	*pA = A;
	*pB = B;

}
/*********************************************/
int Inv_fun_758(uint64 A, uint64 B, uint64 *oA, uint64 *oB)
{
	int i;
	uint64 tmp;


	for (i=0; i<6; i++)
	{
		tmp = A;
		A = B;
		B = tmp;
		Inv_fun_755 (&A, &B, i);
	}
	*oA = A;
	*oB = B;
}
/*********************************************/
/*********************************************/
int fun_755 (uint64 *pA, uint64 *pB, int cpt)
{
	uint64 A,B;
	uint64 V,V2,V3;
	uint64 ResA, ResB;

	A = *pA;
	B = *pB;


	V = (A ^ (LUT2[2*cpt] + B)) & 0xFFFFFFFF ;
	V2 =  ((LUT2[2*cpt+1] | V ) ^ B ) <<32;
	V3 = (V | V2);

	ResA = ((V3) >> 32) ;
	ResB = ((V3) & 0xFFFFFFFF);
```

```
		*pA = ResA;
		*pB = ResB;
}
/********************************************/
int fun_758(uint64 A, uint64 B, uint64 *oA, uint64 *oB)
{
		int i;
		uint64 tmp;

		printf("iA_758: %llx\n", A);
		printf("iB_758: %llx\n", B);

		for (i=0; i<6; i++)
		{
				fun_755 (&A, &B, i);
				tmp = A;
				A = B;
				B = tmp;
		}
		*oA = A;
		*oB = B;
}
/********************************************/
/********************************************/
int fun_901 (uint64 *A, uint64 *B, int cpt)
{
		uint64 AL, AH, BH, BL;

		uint64 O1, O2, O3, O4, O5, O6, O7;

		AL = (*A) & 0xFFFFFFFF;
		BL = (*B) & 0xFFFFFFFF;

		AH = ((*A)>>32) & 0xFFFFFFFF;
		BH = ((*B)>>32) & 0xFFFFFFFF;

		O1 = LUT0[cpt] ^ AL;
		O2 = ((AH + 0x45786532) ^ BL) & 0xFFFFFFFF;
		O3 = ROTL(BH , 4);
		O4 = (AH + 0x45786532) & 0xFFFFFFFF;
		O5 = ((LUT0[cpt] ^ AL) - (AH + 0x45786532)) & 0xFFFFFFFF ;
		O6 = ((AH + 0x45786532) ^ BL) & 0x80000000 ;

		if (O6 == 0)
				O7 = 0x00000000818F694A;
		else
				O7 = 0x0000000060BF080F;



		*A = (O5) | ((((O7) ^ (O4)) ^ (O3)) << (0x20)) ;
		*B = (O2) | ((O3) << (0x20)) ;



}
/********************************************/
int fun_368(uint64 A, uint64 B, uint64 *oA, uint64 *oB, int nbLoop)
{
		int i;

		for (i=0; i<nbLoop; i++)
		{
				fun_901 (&A, &B, i);
		}
		*oA = A;
		*oB = B;
}
/********************************************/
```

```
/*********************************************/
int fun_322 (uint64 *pA, uint64 *pB)
{
		uint64 A,B;
		uint64 Ah,Bh;
		uint64 Al,Bl;
		uint64 V,V2,V3;
		uint64 ResA, ResB;

    A = *pA;
    B = *pB;


		Al = A & 0xFFFFFFFF;
		Ah = A >> 32;
		Bl = B & 0xFFFFFFFF;
		Bh = B >> 32;

		ResA = (((Al ^ (Ah + Bl) ) + LUT3[Bh&0xFF] ) & 0xFFFFFFFF) | ((Ah & Bl)<<32) ;

		V = (Al ^ (Ah + Bl)) & 0xFFFFFFFF  ;
		ResB = ((Bl - V ) & 0xFFFFFFFF)  |  (((V + LUT3[Bh&0xFF] ) ^ (Bh>>8) ) <<32) ;



		*pA = ResA;
		*pB = ResB;
}

/*********************************************/
int fun_328(uint64 A, uint64 B, uint64 *oA, uint64 *oB)
{
		int i;

		for (i=0; i<4; i++)
		{
				fun_322 (&A, &B);
		}

		*oA = A;
		*oB = B;
}
/*********************************************/
/*********************************************/
int Inv_Hash1( uint64 kA, uint64 kB, uint64 i0, uint64 i1, uint64 *o0, uint64 *o1)
{
		uint64 O368_A, O368_B;
		uint64 O328_A, O328_B;
		uint64 O769;
		uint64 O766;
		uint64 ResA, ResB;
		int idx;

						O766 = i0;
						O769 = i1;


						fun_328(kA, kB, &ResA, &ResB);
						printf("A1 =%llx\n", ResA);
						printf("B1 =%llx\n", ResB);
						O328_A = ResA;
						O328_B = ResB;

						for (idx = 15; idx >=1; idx--) {


								printf("\nIDX :%d\n", idx);

								fun_368(O328_A, O328_B, &ResA, &ResB, idx);
								printf("A2 =%llx\n", ResA);
								printf("B2 =%llx\n", ResB);
```

```
                                        O368_A = ResA;
                                        O368_B = ResB;


                                        printf("O_766 =%llx\n", O766);
                                        printf("O_769 =%llx\n", O769);
                                        ResA = O766 & 0xFFFFFFFF;
                                        ResB = O766 >>32;
                                        fun_758(ResA, ResB, &ResA, &ResB);
                                        printf("ResA_4_769 =%llx\n", ResA);
                                        printf("ResB_4_769 =%llx\n", ResB);
                                        Inv_fun_769(&O769, O368_B, ResA, ResB, O769 ) ;


                                        ResA = O769 & 0xFFFFFFFF;
                                        ResB = O769 >>32;
                                        fun_758(ResA, ResB, &ResA, &ResB);
                                        printf("ResA_4_766 =%llx\n", ResA);
                                        printf("ResB_4_766 =%llx\n", ResB);
                                        Inv_fun_766 (&O766, O368_A, ResA, ResB, O766) ;
                                        printf("O_766 =%llx\n", O766);
                                        printf("O_769 =%llx\n", O769);

                                }
                                *o0 = O766;
                                *o1 = O769;
}
/*********************************************/
int printAscii(uint64 i0, uint64 i1, uint64 i2, uint64 i3)
{
        char key[512];
        int i;
        unsigned char *pint;
        char *pkey=key;


        pint = (unsigned char *) &i0;
        for (i=0; i<8; i++) {
                *pkey++ = pint[i];
        }
        pint = (unsigned char *) &i1;
        for (i=0; i<8; i++) {
                *pkey++ = pint[i];
        }
        pint = (unsigned char *) &i2;
        for (i=0; i<8; i++) {
                *pkey++ = pint[i];
        }
        pint = (unsigned char *) &i3;
        for (i=0; i<8; i++) {
                *pkey++ = pint[i];
        }
        *pkey=0;


        printf("Key=%s\n",key);


}
/*********************************************/
int Inv_Hash( uint64 i0, uint64 i1, uint64 i2, uint64 i3)
{
        int idx3;

        printf("#i0 =%llx\n", i0);
        printf("#i1 =%llx\n", i1);
        printf("#i2 =%llx\n", i2);
        printf("#i3 =%llx\n", i3);

        for (idx3 = 0; idx3 <4; idx3++) {
                Inv_Hash1( i0, i1, i2, i3, &i2, &i3) ;
```

```
			printf("\n");
			Inv_Hash1( i2, i3, i0, i1, &i0, &i1) ;
		}

		printf("#o0 =%llx\n", i0);
		printf("#o1 =%llx\n", i1);
		printf("#o2 =%llx\n", i2);
		printf("#o3 =%llx\n", i3);
		printAscii(i0, i1, i2, i3) ;
}
/*******************************************/
int HashC(uint64 kA, uint64 kB, uint64 i0, uint64 i1, uint64 o0, uint64 o1 )
{
		uint64 O368_A, O368_B;
		uint64 O328_A, O328_B;
		uint64 O769;
		uint64 O766;
		uint64 ResA, ResB;
		int idx;

		O769= i1;
		O766= i0;


		fun_328(kA, kB, &O328_A, &O328_B);
		printf("A1 =%llx\n", O328_A);
		printf("B1 =%llx\n", O328_B);

		for (idx = 1; idx <16; idx++) {

			printf("\nIDX :%d\n", idx);


			fun_368(O328_A, O328_B, &O368_A, &O368_B, idx);
			printf("A2 =%llx\n", O368_A);
			printf("B2 =%llx\n", O368_B);


			ResA = O769 & 0xFFFFFFFF;
			ResB = O769 >>32;
			fun_758(ResA, ResB, &ResA, &ResB);
			printf("A3 =%llx\n", ResA);
			printf("B3 =%llx\n", ResB);


			fun_766(O766, O368_A, ResA, ResB, &ResA, &ResB, &O766);

			fun_758(ResA, ResB, &ResA, &ResB);
			printf("A4 =%llx\n", ResA);
			printf("B4 =%llx\n", ResB);
			printf("O766 =%llx\n", O766);

			fun_769(O769, O368_B, ResA, ResB, &ResA );

			printf("A5 =%llx\n", ResA);
			O769 = ResA;

		}
}
/*******************************************/
/*******************************************/
int HashC1(int argc, char *argv[])
{

		uint64 i0, i1, i2, i3;
		uint64 o0, o1, o2, o3;

		uint64 ResA, ResB;
		uint64 O368_A, O368_B;
		uint64 O328_A, O328_B;
		uint64 O769;
```

```
uint64 O766;
uint64 iA, iB;
int idx;
int idx2;
int idx3;

i0 = 0;
i1 = 0;
i2 = 0;
i3 = 0;


if (argc == 5) {
		i0 = strtoll(argv[1], NULL, 16);
		i1 = strtoll(argv[2], NULL, 16);
		i2 = strtoll(argv[3], NULL, 16);
		i3 = strtoll(argv[4], NULL, 16);
}

//O769=0;
O769= i1;

//O766=0;
O766=i0;


iA = i2;
iB = i3;

for (idx3 = 0; idx3 <4; idx3++) {

printf("################################################################################################\n");
		printf("\nIDX3 :%d\n", idx3);
		printf("#i0 =%llx\n", i0);
		printf("#i1 =%llx\n", i1);
		printf("#i2 =%llx\n", i2);
		printf("#i3 =%llx\n", i3);

		for (idx2 = 0; idx2 <2; idx2++) {
			printf("\nIDX2 :%d\n", idx2);
			printf("#\tiA =%llx\n", iA);
			printf("#\tiB =%llx\n", iB);

			fun_328(iA, iB, &ResA, &ResB);
			printf("A1 =%llx\n", ResA);
			printf("B1 =%llx\n", ResB);
			O328_A = ResA;
			O328_B = ResB;

			if (idx2 == 0) {
				O769= i1;
				O766=i0;
			} else {
				O769= i3;
				O766= i2;
			}


			for (idx = 1; idx <16; idx++) {

				printf("\nIDX :%d\n", idx);

				ResA = O328_A;
				ResB = O328_B;

				fun_368(ResA, ResB, &ResA, &ResB, idx);
				printf("A2 =%llx\n", ResA);
				printf("B2 =%llx\n", ResB);
				O368_A = ResA;
				O368_B = ResB;
```

```
                        ResA = O769 & 0xFFFFFFFF;
                        ResB = O769 >>32;
                        fun_758(ResA, ResB, &ResA, &ResB);

                        printf("A3 =%llx\n", ResA);
                        printf("B3 =%llx\n", ResB);


                        fun_766(O766, O368_A, ResA, ResB, &ResA, &ResB, &O766);
                        fun_758(ResA, ResB, &ResA, &ResB);
                        printf("A4 =%llx\n", ResA);
                        printf("B4 =%llx\n", ResB);
                        printf("O766 =%llx\n", O766);


                        fun_769(O769, O368_B, ResA, ResB, &ResA );
                        printf("A5 =%llx\n", ResA);
                        O769 = ResA;

                    }
                    iA = O766;
                    iB = O769;
                    if (idx2 == 0) {
                        o0=O766;
                        o1=O769;
                    } else {
                        o2=O766;
                        o3=O769;
                    }
                }
                i0 = o0;
                i1 = o1;
                i2 = o2;
                i3 = o3;
                printf("#o0 =%llx\n", o0);
                printf("#o1 =%llx\n", o1);
                printf("#o2 =%llx\n", o2);
                printf("#o3 =%llx\n", o3);
        }

}
/*********************************************/
int main(int argc, char *argv[])
{
        uint64 i0, i1, i2, i3;
        uint64 o0, o1, o2, o3;

        //HashC1(argc, argv) ;

        if (argc == 5) {
                i0 = strtoll(argv[1], NULL, 16);
                i1 = strtoll(argv[2], NULL, 16);
                i2 = strtoll(argv[3], NULL, 16);
                i3 = strtoll(argv[4], NULL, 16);
        } else {
                i0 = 0x65850B36E76AAED5;
                i1 = 0xD9C69B74A86EC613;
                i2 = 0xDC7564F1612E5347;
                i3 = 0x658302A68E8E1C24;


                /*i0 = 0x838a2e182b2b97b9;
                i1 = 0xcd47e1858389123c;
                i2 = 0x90259e3e33676fae;
                i3 = 0xd068a8b9d95f6da7; */

        }

        Inv_Hash( i0, i1, i2, i3) ;
```

```
}
```

5. Disass.py

```
#!/usr/bin/python
import sys
import getopt
import os
import time
import re


def load_ins(filename):
	res = list()
	with open(filename) as f:
		for line in f:
			lgs = line.strip('\n')
			m = re.search('(0x[0-9a-f]+):[ ]*(0x[0-9a-f]+)', lgs)
			if m:
				v = m.group(1)
				#print v
				iptr = int(v, 16)
				#print "0x%x"%vd
				v = m.group(2)
				#print v
				ins = int(v, 16)
				#print "0x%x"%vd
				res.append([iptr, ins])

	return res

def show_ins(ins):
	print "0x%04x"%ins

	opcode = (ins>>20)&0xFF
	p0 = ins & 0x3FFF
	p2 = (ins >> 18 ) & 3
	p3 = (ins >> 14) & 0xF
	p4 = (ins >> 10) & 0xF

	print "\topcode=0x%x"%opcode
	print "\topc2=0x%x"%p2
	print "\tP0=0x%x"%p0
	print "\tP3=0x%x"%p3
	print "\tP4=0x%x"%p4

def show_ins2(ins):
	print "0x%04x"%ins

	opcode = (ins>>20)&0xFF
	p0 = ins & 0x3FFF
	p2 = (ins >> 18 ) & 3
	p3 = (ins >> 14) & 0xF
	p4 = (ins >> 10) & 0xF

	print "\tp2=0x%04x"%p2

	print "\t",

	if opcode == 0 :
		if p2 == 3:
			print "R[0x%x] = 0x%x"%(p3,p0)
		elif p2 == 1:
			print "R[0x%x] = MEM[R[0x%x]]"%(p3,p4)
		elif p2 == 0:
```

```
                        print "R[0x%x] = R[0x%x]"%(p3,p4)
                elif p2 == 2:
                        print "MEM[R[0x%x]]= R[0x%x]"%(p3,p4)
                else:
                        print "Unknown opc2:0x%x"%p2
        elif opcode == 1:
                print "R[0x%x] -= 1"%(p3)
        elif opcode == 2:
                if p2 == 3:
                        print "R[0x%x] += 0x%x"%(p3,p0)
                elif p2 == 0:
                        print "R[0x%x] += R[0x%x]"%(p3,p4)
        elif opcode == 3:
                if p2 == 3:
                        print "R[0x%x] -= 0x%x"%(p3,p0)
                elif p2 == 0:
                        print "R[0x%x] -= R[0x%x]"%(p3,p4)
        elif opcode == 4:
                print "R[0x%x] <<= 0x%x"%(p3,p0)
        elif opcode == 5:
                print "R[0x%x] >>= 0x%x"%(p3,p0)
        elif opcode == 6:
                print "R[0x%x] ^= R[0x%x]"%(p3,p4)
        elif opcode == 7:
                print "R[0x%x] &= 0x%x"%(p3,p0)
        elif opcode == 8:
                print "R[0xf] = 0x%x // JUMP 0x%x"%(p0,p0)
        elif opcode == 9:
                print "if (R[0x%x] == 0) R[0xf]+=3 else R[0xf] = 0x%x"%(p3,p0)
        elif opcode == 0xa:
                print "R[0]; EXIT()"
        elif opcode == 0xb:
                print "R[0x%x] = (R[0x%x] & 0xFF ) <<8) | (R[0x%x] >> 8)"%(p3,p3,p3)
        elif opcode == 0xc:
                print "SetKey(R[0x%x])"%(p3)
        elif opcode == 0xd:
                print "NOP"
        elif opcode == 0xe:
                print "if (*0x9010000>5) R[0xf] = 0x%x else R[0xf]+=3"%(p0)
        else:
                print "Unknown opcode:0x%x"%opcode

def check_opc2(prog):
        t_opc = []

        for i in range(0,16):
                l = list()
                for j in range(0,4):
                        l.append(0)
                t_opc.append(l)

        for e in prog:
                ins = e[1]
                opcode = (ins>>20)&0xFF
                p2 = (ins >> 18 ) & 3
                l = t_opc[opcode]
                l[p2] +=1

        for i in range(0, len(t_opc)):
                print "%x:"%(i),
                l = t_opc[i]
                for j in range(0, len(l)):
                        print ",%x"%(l[j]),
                print

def disass(prog):
        for e in prog:
                print "0x%03x"%e[0],
                #show_ins(e[1])
                show_ins2(e[1])
```

```python
def main(argv):

        #inputfile = 'ins1_codes.txt'
        #inputfile = 'ins_end_code.txt'
        inputfile = 'ins_full_code.txt'

        prog  = load_ins(inputfile)
        print len(prog)
        disass(prog)

        check_opc2(prog)



if __name__ == '__main__':
        main(sys.argv[1:])
```

6. Decrypt.c

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


typedef unsigned int uint32;
typedef       int int32;

typedef unsigned short uint16;

#define MEM_SIZE 0x101010

unsigned char memory[MEM_SIZE];


#define SWAP_BYTES(a) ((((a)<<8) & 0xFF00) | (((a)>>8) & 0xFF))

/**********************************/
uint32 read_mem(uint32 addr)
{
        int32 of7;
        uint32 val=0;


        of7 = addr ;

        if ((of7 >=0 ) && (of7 < MEM_SIZE)) {
                val = *(uint32 *) (memory + of7);
        } else {
                printf("addr out of range: %X\n",addr);
        }

        return(val);
}
/**********************************/
uint32 SetKey(uint32 idx)
{
        return(0);
}
/**********************************/
uint32 set_mem(uint32 addr, uint32 val)
{

        int32 of7;
```

```
		of7 = addr;

		if ((of7 >=0 ) && (of7<MEM_SIZE)) {
				*(uint32 *) (memory + of7) = val;
		} else {
				printf("addr out of range: %X\n",addr);
		}
		return(0);
}
/**********************************/
uint32 Hash(uint32 RI, uint32 *salt)
{
		uint32  R[16];

		R[0x7] = *salt;

		//printf("RI=0x%x\n",RI);

						R[0x4] = (RI >> 8) & 0xff;
						R[0x5] = (RI) & 0xff;


						R[0xa] = (R[0x7] << 0x10) + 0x1000 + (R[0x5]<<8) + R[0x4];
						//printf("Addr=0x%x\n",R[0xa]);
						R[0x6] = read_mem(R[0xa]);
						//printf("\tRead VAL=0x%x\n",R[0x6]);
						R[0x6] &= 0xff;


						if (R[0x7] == 0)
								R[0x07] = 0xa;
						R[0x7] -= 1;



						R[0xa] = (R[0x7] << 0x10) + 0x1000 + (R[0x4]<<8) + R[0x6];
						R[0x5] = read_mem(R[0xa]);
						R[0x5] &= 0xff;


						if (R[0x7] == 0)
								R[0x7] = 0xa;
						R[0x7] -= 1;


						R[0xa] = (R[0x7] << 0x10) + 0x1000 + (R[0x6]<<8) + R[0x5];
						R[0x4] = read_mem(R[0xa]);
						R[0x4] &= 0xff;

						if (R[0x7] == 0)
								R[0x7] = 0xa;
						R[0x7] -= 1;


						R[0xa] = (R[0x7] << 0x10) + 0x1000 + (R[0x5]<<8) + R[0x4];
						R[0x6] = read_mem(R[0xa]);
						R[0x6] &= 0xff;

						if (R[0x7] == 0)
								R[0x7] = 0xa;
						R[0x7] -= 1;


						R[0x9] = R[0x6];
						R[0x9] <<= 0x8;
						R[0x9] += R[0x4];
```

```
			*salt = R[0x7];

			//printf("\tR9=0x%x\n",R[9]);
			return(R[0x9]);

}
/**********************************/
int pcrypt()
{

		int i_c, i_e, i_b;
		uint32  R[16];


		R[0x4] = ((0x10<<0x10) + 0x20);
		R[0xd] = ((0x10<<0x10) + 0x20);

		R[0xc] = 0x4;


		for (i_c = 4; i_c>0; i_c--) {

			R[0x0] = read_mem(R[0x4]);
			R[0x0] <<= 0x10;
			R[0x0] >>= 0x10;
			R[0x0] = ((R[0x0] & 0xFF ) <<8) | (R[0x0] >> 8);

			R[0x4] += 0x2;
			R[0x1] = read_mem(R[0x4]);
			R[0x1] <<= 0x10;
			R[0x1] >>= 0x10;
			R[0x1] = ((R[0x1] & 0xFF ) <<8) | (R[0x1] >> 8);

			R[0x4] += 0x2;
			R[0x2] = read_mem(R[0x4]);
			R[0x2] <<= 0x10;
			R[0x2] >>= 0x10;
			R[0x2] = ((R[0x2] & 0xFF ) <<8) | (R[0x2] >> 8);

			R[0x4] += 0x2;
			R[0x3] = read_mem(R[0x4]);
			R[0x3] <<= 0x10;
			R[0x3] >>= 0x10;
			R[0x3] = ((R[0x3] & 0xFF ) <<8) | (R[0x3] >> 8);


			R[0xe] = 0x20;
			R[0x7] = 0x7;

			for (i_e = 32; i_e > 0; i_e--) {


				R[0xe] -= 1;

				R[0x9] = Hash(R[0x1], &(R[0x7]));



				R[0x8] = ((R[0xe])>>0x3) & 0x1;

				SetKey(R[0xe]);


				if (R[0x8] == 0) {
					R[0x8] = R[0x3];
					R[0x3] = (R[0xe] + 1) ^ R[0x0];
					if (1) {
						R[0x3] ^= R[0x1];
						R[0x0] = R[0x9];
						R[0x1] = R[0x2];
						R[0x2] = R[0x8];
```

```
                        }
                } else {

                        R[0x8] = R[0x0];
                        R[0x0] = R[0x9];

                        R[0x1] = (R[0xe]+1) ^ R[0x0] ^ R[0x2];
                        if (1) {
                                R[0x2] = R[0x3];
                                R[0x3] = R[0x8];
                        }
                }


        }

        R[0x0] = ((R[0x0] & 0xFF ) <<8) | (R[0x0] >> 8);
        R[0x1] = ((R[0x1] & 0xFF ) <<8) | (R[0x1] >> 8);
        R[0x1] <<= 0x10;
        R[0x0] += R[0x1];
        set_mem(R[0xd],  R[0x0]) ;
        R[0xd] += 0x4;

        R[0x2] = ((R[0x2] & 0xFF ) <<8) | (R[0x2] >> 8);
        R[0x3] = ((R[0x3] & 0xFF ) <<8) | (R[0x3] >> 8);
        R[0x3] <<= 0x10;
        R[0x2] += R[0x3];
        set_mem(R[0xd] , R[0x2]);
        R[0xd] += 0x4;

        R[0x4] = R[0xd];
        R[0xc] -= 1;

    }


    R[0xc] = 0x10;
    R[0xc] <<= 0x10;
    R[0xb] = 0x20;

    R[0xd] -= 0x20;
    R[0x4] = 0x0;

    for (i_b = 32; i_b > 0; i_b--) {
        R[0x0] = read_mem(R[0xd]);
        R[0x0] &= 0xff;
        R[0x1] = read_mem(R[0xc]);
        R[0x1] &= 0xff;


        R[0x0] -= R[0x1];

        if (R[0x0] !=0)
            R[0x4] = 0x1;

        R[0xd] += 0x1;
        R[0xc] += 0x1;
    }

        return(R[0x4]);

}
/*********************************/
int LoadKey()
{
    int i;
    uint32 addr;
    uint32 val;
```

```
	addr = 0x100000 + 0x20;

	for (i=0; i<8; i++) {
		val = 0x12345678;
		set_mem(addr, val);
		addr +=4;
	}

}
/**********************************/
int encrypt(uint32 ikeys[8], uint32 output[8])
{
	int i;
	uint32 addr;
	uint32 val;

	int i_c, i_e, i_b;
	uint32  R[16];


	addr = 0x100000 + 0x20;

	for (i=0; i<8; i++) {
		set_mem(addr, ikeys[i]);
		addr +=4;
	}




	R[0x4] = ((0x10<<0x10) + 0x20);
	R[0xd] = ((0x10<<0x10) + 0x20);

	R[0xc] = 0x4;


	for (i_c = 4; i_c>0; i_c--) {

		R[0x0] = read_mem(R[0x4]);
		R[0x0] <<= 0x10;
		R[0x0] >>= 0x10;
		R[0x0] = ((R[0x0] & 0xFF ) <<8) | (R[0x0] >> 8);

		R[0x4] += 0x2;
		R[0x1] = read_mem(R[0x4]);
		R[0x1] <<= 0x10;
		R[0x1] >>= 0x10;
		R[0x1] = ((R[0x1] & 0xFF ) <<8) | (R[0x1] >> 8);

		R[0x4] += 0x2;
		R[0x2] = read_mem(R[0x4]);
		R[0x2] <<= 0x10;
		R[0x2] >>= 0x10;
		R[0x2] = ((R[0x2] & 0xFF ) <<8) | (R[0x2] >> 8);

		R[0x4] += 0x2;
		R[0x3] = read_mem(R[0x4]);
		R[0x3] <<= 0x10;
		R[0x3] >>= 0x10;
		R[0x3] = ((R[0x3] & 0xFF ) <<8) | (R[0x3] >> 8);


		R[0xe] = 0x20;
		R[0x7] = 0x7;

		for (i_e = 32; i_e > 0; i_e--) {

			/*printf("i=%d\n",i_e);
			printf("R0=0x%x\n",R[0]);
			printf("R1=0x%x\n",R[1]);
			printf("R2=0x%x\n",R[2]);
```

```
			printf("R3=0x%x\n\n",R[3]);*/


			R[0xe] -= 1;

			//printf("Salt:%x\n",R[7]);
			R[0x9] = Hash(R[0x1], &(R[0x7]));



			R[0x8] = ((R[0xe])>>0x3) & 0x1;

			//SetKey(R[0xe]);


			if (R[0x8] == 0) {
				R[0x8] = R[0x3];
				R[0x3] = (R[0xe] + 1) ^ R[0x0];
				if (1) {
					R[0x3] ^= R[0x1];
					R[0x0] = R[0x9];
					R[0x1] = R[0x2];
					R[0x2] = R[0x8];
				}
			} else {

				R[0x8] = R[0x0];
				R[0x0] = R[0x9];

				R[0x1] = (R[0xe]+1) ^ R[0x0] ^ R[0x2];
				if (1) {
					R[0x2] = R[0x3];
					R[0x3] = R[0x8];
				}
			}


		}

		/*printf("R0=0x%x\n",R[0]);
		printf("R1=0x%x\n",R[1]);
		printf("R2=0x%x\n",R[2]);
		printf("R3=0x%x\n\n\n",R[3]);
		printf("==\n");*/


		R[0x0] = ((R[0x0] & 0xFF ) <<8) | (R[0x0] >> 8);
		R[0x1] = ((R[0x1] & 0xFF ) <<8) | (R[0x1] >> 8);
		R[0x1] <<= 0x10;
		R[0x0] += R[0x1];
		set_mem(R[0xd],  R[0x0]) ;
		R[0xd] += 0x4;

		R[0x2] = ((R[0x2] & 0xFF ) <<8) | (R[0x2] >> 8);
		R[0x3] = ((R[0x3] & 0xFF ) <<8) | (R[0x3] >> 8);
		R[0x3] <<= 0x10;
		R[0x2] += R[0x3];
		set_mem(R[0xd] , R[0x2]);
		R[0xd] += 0x4;

		R[0x4] = R[0xd];
		R[0xc] -= 1;

	}


	addr = (R[0xd] - 0x20);

	for (i=0; i<8; i++) {
		output[i] = read_mem(addr);
		//printf("Read out addr: 0x%x\n",addr);
```

```
			//printf("Read out val: 0x%x\n",output[i]);
			addr +=4;
		}

}
/**********************************/
/**********************************/
uint16 luts[65536][11];
uint16 InvLuts[65536][11];
/**********************************/
int InitLuts()
{
		int salt;
		int i;
		int R;

		for (i=0; i<11; i++) {
			for (R=0; R<65536; R++) {
				salt = i;
				luts[R][salt] = Hash(R, &salt);
			}
		}

/*		for (i=0; i<11; i++) {
			for (R=0; R<65536; R++) {
				salt = i;
				InvLuts[ R ] [salt] =0;
			}
		}

		for (i=0; i<11; i++) {
			for (R=0; R<65536; R++) {
				salt = i;
				InvLuts[ luts[R][salt]] [salt] ++;
			}
		}

		for (R=0; R<65536; R++) {
			printf("cnt=%d\n",InvLuts[R][0]);
		}
*/
}
/**********************************/
uint32 HashInv(uint32 R, uint32 *salt)
{
		int i;
		uint32 res=0;

		R &= 0xFFFF;


		//printf("HInv_R:%x\n",R);
		for (i=0; i<65536; i++) {
			if (luts[i][*salt] == R) {
				//printf("HInv found:%d\n",i);
				res = i;
				break;
			}
		}
		//printf("HInv i:%d\n",i);
		if (i==65536)
			printf("HInv not found:%x\n",R);

		//printf("HInv_REs:%x\n",res);

		*salt +=4;
		if (*salt >=10)
			*salt -=10;
		return(res);
}
/**********************************/
```

```
int decrypt1(uint32 output[2], uint32 ikeys[2])
{
        int i;
        uint32  R[16];
        uint32 salt;
        uint32 tmp;

        R[0] = output[0] & 0xFFFF;
        R[1] = (output[0] >>16)& 0xFFFF;
        R[2] = output[1] & 0xFFFF;
        R[3] = (output[1] >>16)& 0xFFFF;

                /*printf("iR0=0x%x\n",R[0]);
                printf("iR1=0x%x\n",R[1]);
                printf("iR2=0x%x\n",R[2]);
                printf("iR3=0x%x\n\n",R[3]);*/

        R[0]= SWAP_BYTES(R[0]);
        R[1]= SWAP_BYTES(R[1]);
        R[2]= SWAP_BYTES(R[2]);
        R[3]= SWAP_BYTES(R[3]);

        salt=3; // TODO ...
        for (i = 1; i <= 32; i++) {

                /*printf("i=%d\n",i);
                printf("R0=0x%x\n",R[0]);
                printf("R1=0x%x\n",R[1]);
                printf("R2=0x%x\n",R[2]);
                printf("R3=0x%x\n\n",R[3]);*/

                R[0x8] = ((i-1)>>0x3) & 0x1;

                if (R[0x8] == 0) {
                        tmp = R[3];
                        R[3] = R[2];
                        R[2] = R[1];
                        R[1] = HashInv(R[0], &salt);
                        R[0] = tmp ^ (i) ^ R[1];
                } else {
                        tmp = R[3];
                        R[3] = R[2];
                        R[2] = R[0] ^ (i) ^ R[1];
                        R[1] = HashInv(R[0], &salt);
                        R[0] = tmp;
                }

        }
        /*printf("R0=0x%x\n",R[0]);
        printf("R1=0x%x\n",R[1]);
        printf("R2=0x%x\n",R[2]);
        printf("R3=0x%x\n\n\n",R[3]);
        printf("==\n");*/

        R[0]= SWAP_BYTES(R[0]);
        R[1]= SWAP_BYTES(R[1]);
        R[2]= SWAP_BYTES(R[2]);
        R[3]= SWAP_BYTES(R[3]);

        ikeys[0] = R[0] + (R[1] <<16) ;
        ikeys[1] = R[2] + (R[3] <<16) ;

}
/**********************************/
int decrypt(uint32 output[8], uint32 ikeys[8])
{
        int i;

        for (i=0; i<4; i++) {
                decrypt1(output+ 2*i, ikeys+2*i);
        }
```

```
}
/**********************************/
int load_mem(unsigned char *memory)
{
	FILE *fch;
	int msize;
	int cnt[256];
	int i,j,k;

	//fch = fopen("LUT1.bin","rb");
	fch = fopen("LUT_hacked.bin","rb");
	if (fch == NULL)
		exit(1);

	msize = fread(memory, sizeof(unsigned char), MEM_SIZE, fch);

	printf("Load %d bytes in memory\n", msize);

	fclose(fch);

	/*for (k=0; k<11; k++)
		for (i=0; i<256; i++)
			for (j=0; j<256; j++)
				memory[k*65536+i*256+j] = (i+j)&0xFF;*/


	for (i=0; i<256; i++)
		cnt[i]=0;

	for (i=0; i<256; i++)
		cnt[memory[i*256]]++;

	for (i=0; i<256; i++)
		printf("cnt0=%d\n",cnt[i]);


}
/**********************************/
int main(int argc, char *argv[])
{
	uint32 res;
	int i;

	uint32 ikeys[8];
	uint32 output[8];
	uint32 dkeys[8];


	load_mem(memory+0x1000);

	LoadKey();
	res = pcrypt();
	printf("res=%d\n",res);

	ikeys[0] = 0x12345678;
	ikeys[1] = 0xA2345678;
	ikeys[2] = 0xB2345678;
	ikeys[3] = 0xC2345678;
	ikeys[4] = 0xD2345678;
	ikeys[5] = 0xE2345678;
	ikeys[6] = 0xF2345678;
	ikeys[7] = 0x92345678;

	ikeys[0] = 0x67452301;
	ikeys[1] = 0x45230189;
	ikeys[2] = 0x23018967;
	ikeys[3] = 0x01896745;
	ikeys[4] = 0x89674523;
	ikeys[5] = 0x67452301;
	ikeys[6] = 0x45230189;
```

```
        ikeys[7] = 0x34128967;


        for (i=0; i<8; i++) {
                printf("ikeys%d = %x\n",i,ikeys[i]);
        }
        encrypt( ikeys,  output);
        for (i=0; i<8; i++) {
                printf("output%d = %x\n",i,output[i]);
        }

        printf("============================\n");

        output[0] = 0x612e7270;
        output[1] = 0x6766722e;
        output[2] = 0x666e632e;
        output[3] = 0x2e76662e;
        output[4] = 0x76706e73;
        output[5] = 0x66407279;
        output[6] = 0x70766766;
        output[7] = 0x7465622e;

        InitLuts();
        decrypt( output, dkeys);
        for (i=0; i<8; i++) {
                printf("dkeys%d= %x\n",i,dkeys[i]);
        }

}
```