Determining Image Base of Firmware Files for ARM Devices

Ruijin ZHU†(a), Yu-an TAN†, Quanxin ZHANG†, Fei WU††, Jun ZHENG†(b), Nonmembers, and Yuan XUE†, Student Member

SUMMARY Disassembly, as a principal reverse-engineering tool, is the process of recovering the equivalent assembly instructions of a program’s machine code from its binary representation. However, when disassembling a firmware file, the disassembly process cannot be performed well if the image base is unknown. In this paper, we propose an innovative method to determine the image base of a firmware file with ARM/Thumb instruction set. First, based on the characteristics of the function entry table (FET) for an ARM processor, an algorithm called FIND-FET is proposed to identify the function entry tables. Second, by using the most common instructions of function prologue and FETs, the FIND-BASE algorithm is proposed to determine the candidate image base by counting the matched functions and then choose the one with maximal matched FETs as the final result. The algorithms are applied on some firmwares collected from the Internet, and results indicate that they can effectively find out the image base for the majority of example firmware files.

key words: reverse engineering, image base, determining, disassembly, firmware

1. Introduction

A software product is usually released as a compiled binary file which makes it difficult to obtain its design ideas and principles for normal users. However, reverse engineering can generate the corresponding source code, system structure and design principles from the binary file. Reverse engineering can not only avoid duplicating efforts and improve the efficiency and quality of software but also translate legacy system into evolution system to efficiently reuse them. Hence, reverse engineering has huge economic value. It has been wildly used in malware analysis, binary reuse, binary translation, vulnerabilities detection and other fields.

Since the machine code of binary file is difficult to understand, it needs to disassemble [1], [2] the binary file firstly, which is the basic and important component of reverse engineering. At present, the disassembly algorithm mainly includes linear sweep and recursive traversal [2], [3]. A critical problem of disassembling is to differentiate code bytes and data bytes whose distribution information is implied by the format of the file. Hence, if the format of the binary file is known, the disassembly algorithms will work properly. When dealing with a file whose format is unknown, the disassembly algorithm cannot distinguish code bytes from data bytes directly. In this case, only the image base of the file and function entry address can be used to get the distribution information of the code. Using the correct base address ensures that the disassembler provides a more complete analysis of the binary file. Indeed, when the correct base address is used, disassembler is able to build accurate cross references in instances where the address reference uses absolute addresses rather than offsets in the binary file [4]. The cross references, which includes jump location references, function references, string references and so on, can be a great help when trying to navigate messy disassembly code. On the other hand, working with an incorrect base address may lead to inaccurate interpretations of segments referenced by immediate addresses. Therefore, knowledge of the true base address is critical in understanding the binary file as a whole [5].

As to the embedded firmware, Skochinsky et al. [6] proposed a general principle for determining the image base of a file with unknown format. They suggested some kinds of hints, such as self-relocating code, initialization code, jump tables, string tables, etc., can be used, but it is not an automatic method and heavily relies on the engineer’s experience. Zachry Basnight et al. [5] presented an overview of the reverse engineering process and proposed a method which can manually reason out image base by immediate values in the instruction.

All the above methods require the intuition and experience of a reverse engineer, and the success and effectiveness often rely on the human factor. To our best knowledge, at present there is still no automatic method can get the image base of a file with unknown format. Since most of the unknown format files in reverse engineering come from firmware in embedded systems of which 63% were identified as targeting ARM devices [7], our algorithm focuses on the firmware under ARM architecture. In this paper, we proposed a method that can effectively determine the image base of unknown format files, other than ELF executable files with the loading base information. The method consists of two algorithms. First, based on the characteristics of function entry table (FET), we propose an algorithm called FIND-FET to identify the FETs in an unknown format file. Second, utilizing the FETs identified by FIND-FET, we determine the image base of the file with FIND-BASE algo-
We present an algorithm to identify FETs in binary file for the first time. By using sliding window and FET gap, our algorithm can effectively identify the FET. Disassembly tools need to know the function entry address for working. Hence, it’s the critical step to get the binary function entry addresses for unknown format files’ disassembly process.

We present an algorithm to determine image base of unknown format binary file that contains FETs for the first time. The algorithm relies on the FETs and most common instructions of binary function prologue to determine the image base. The experiment results prove that the proposed algorithm is effective for determining the image base of binary files containing function entry tables.

The remainder of this paper is organized as follows. In Sect. 2, we introduce the generation and characteristics of function entry table, and present an algorithm to identify them. In Sect. 3 we propose an algorithm to determine the image base of the binary file. The experiments and the analysis of the result are given in Sect. 4. This paper is concluded in Sect. 5.

2. Identifying of Function Entry Table

2.1 Function Entry Table

Table 1 shows a part of kernel source code from Samsung mobile phones [8]. Code in such form can be found frequently when we analyze source code of large-scale applications. Typical characteristics of this kind of code are that there is an initialized global array of structures of which function pointer variable is a member. The value of such structure array is written to a created binary file during the linking process. In this file, each function pointer is assigned the function entry address. All of the function entry addresses in an array of structures form a table in binary file, which is referred to Function Entry Table (FET) in this paper.

FET can be classified into two categories. The first one is discrete type where the function pointers are not adjacent. If structure contains more than one member and one of the members is a function entry pointer, array of this structure will be compiled into discrete FET in binary file. The second one is continuous type where function pointers stores continuously. If structure contains only one member and this member is a function pointer, array of this structure is simplified into function pointer array. The function pointer array can be compiled to generate a FET belonging to continuous distribution type.

Table 2 and Table 3 respectively show an example for above two types of FET. The binary files compiled† from them are shown in Fig. 1 and Fig. 2 respectively. FET is usually a continuous area, as annotated in solid line in the two figures. In Fig. 1, the 8 bytes (64 bits) annotated in dotted line represent a structure where “28 82 00 01 00 00 00 00” is the first element of the array structure. The bytes “28 82 00 00” is the value of fun_array1[0].id and the bytes “01 00 00 00” is the value of fun_array1[0].id.

Table 1 Samsung I9500 kernel code snippet.

Table 2 The example of structure corresponding to discrete FET.

Table 3 The example of structure corresponding to continuous FET.

Fig. 1 Compiled binary file from Table 2.

†In the current compiling environment (arm-linux-gcc 4.3.2), int type variables and pointer variables consume 4 bytes (32 bits) respectively.
Since this binary file stored in little-endian byte order, \texttt{fun \_array1[0].func} is 0x00008228 and \texttt{fun \_array1[0].id} is 1. By analogy, \texttt{fun \_array1[1].func} is 0x00008270 and \texttt{fun \_array1[1].id} is 2, and so on. Similarly, in Fig. 2 the value of \texttt{fun \_array2[0].func} is 0x00008228, the value of \texttt{fun \_array2[1].func} is 0x00008270 and so on.

### 2.2 FIND-FET Algorithm

In Sect. 2.1 we illustrated the storage forms of FET. Now, we present an algorithm called FIND-FET to identify the FETs in a binary file.

The feature of FET is that the entry address of function distributes regularly, as shown in Fig. 1 and Fig. 2. It is because that each function pointer is stored as a member of the structure element, and each structure element has the fixed size. For a 32-bit device, a function pointer occupies 4 bytes which is a 32-bit integer. The distance between two pointers is very short in general.

We have the following Definitions:

**Definition 1.** The number of words (4 bytes) between two adjacent function pointers in the same FET is defined as FET gap.

**Definition 2.** Sliding window is corresponding to a contiguous memory unit. The number of memory units that it occupies is fixed.

**Definition 3.** The size of sliding window (\(w_{nd}\)) is the number of function entry addresses in the window.

The gap of FET in Fig. 1 is 1. Next, we will introduce the FET identifying algorithm in the case that gap is 1. Other cases are similar. Figure 3 is a FIND-FET algorithm model where the gap is 1 and the size of sliding window (\(w_{nd}\)) is 3. The first 4 bytes in the sliding window is the first function entry address. In the case that the gap of FET is 1, we specify the first function entry address as the starting point and calculate the location of subsequent function entry address in turn. We define the following rules:

**Rule 1:** distance between each two of the function entry addresses in window not more than 64KB.

**Rule 2:** function entry addresses within the sliding window cannot be the same address.

**Rule 3:** function entry addresses in the window meet the ARM/Thumb function entry address rules, namely entry address for ARM function is a multiple of 4, and Thumb function entry address is odd.

**Fig. 2** Compiled binary file from Table 3.

**Fig. 3** FET model of FIND-FET algorithm.

For **Rule 1**, the memory locations pointed by addresses in the same FET usually distribute relatively concentrated. In our experiments, we found that the distance between these function entry addresses in a same FET is usually less than 64KB. Sometimes binary files have some padding bytes which may be 0x00, 0xFF, 0xCC etc. In order to exclude these padding bytes, we set **Rule 2**. For **Rule 3**, if function entry address in a sliding window appears neither the ARM function entry address nor the Thumb function entry address, then we consider the sliding window area is not a FET.

If all the function entry addresses in the window conform to the above three matching rules, we consider that the sliding window area is a possible function entry table. Then the sliding window moves forward, until the addresses in the sliding window do not match the rules. The FIND-FET algorithm is detailed as Algorithm 1.

In this paper, we only detailed one source of FET, i.e. the array of structures that contains function pointer. There are also some other sources of FET which include the interrupt vector table (IVT), the function entry addresses stored in the literal pool[9], and so on. Although the source of FETs is different, they all have a common feature, namely the content of FETs is function entry addresses. Therefore, the FIND-FET can identify FETs from all sources mentioned above. On the other hand, there are some content in firmware whose structure is similar to FET, but they are not function entry address. For example, when compiler processes the switch-case statement in C language, if the addresses of case are dense and sequential, these addresses are usually stored in a jump table[10], [11]. The structure of the jump table is similar to the one of FET and the FIND-FET algorithm identifies them as FETs, which produces some dummy FETs.

Hence, according to the 3 rules defined in the algorithm FIND-FET, there may still be some real FETs are filtered.
Algorithm 1 FIND-FET algorithm

1: function FindFET(start, fileSize, wnd)
2: pos ← start
3: end ← start + fileSize
4: while start ≤ pos < end do
5: for gap ← 0, gap_max do
6: if match(Rule1) && match(Rule2) && match(Rule3) then
7: head ← pos
8: tableSize ← wnd
9: MoveWindow()
10: while match(Rule1) && match(Rule2) && match(Rule3) do
11: tableSize + +
12: MoveWindow()
13: end while
14: pos + = tableSize
15: Output : head, gap, tableSize
16: break
17: end if
18: end for
19: pos + +
20: end while
21: end function

out, and the results may also contain some dummy FETs.
However, the algorithm can find out most of the genuine
FETs, which is enough to determine the correct image base
using the algorithm proposed in Sect. 3.

3. The Determining of Image Base

3.1 Common Instruction of Prologue

Disassembling the binary file we can get some binary func-
tions which roughly correspond to functions in a high-level
language [12], [13]. When compiling a function, the com-
piler usually adds some instructions at the beginning to cre-
ate and initialize stack frame and save registers. These in-
cstructions are called prologue. Similarly, it also adds some
instructions at the end of the function to clear stack frame
and restore registers. These instructions are called epilogue.
The binary function which is compiled from function in
high-level language includes prologue, body and epilogue.

We will introduce the common instruction of prologue
under ARM architecture. In general, ARM processor has
two working states, i.e. ARM state and Thumb state. In
ARM state, instructions are always word-aligned (32 bits),
so the entry address of ARM function is an even number.
In Thumb state, instructions can be halfword-aligned (16
bits) or word-aligned. ARM processor can switch between
the two states. When executing the BX instruction, if the
least significant bit (LSB) of the target address is 1, the pro-
cessor switches to Thumb state. Otherwise, it switches to
ARM state. For example, when the LSB of register Rm is
1, the execution of instruction “BX Rm” makes the process-
or switch to the Thumb state. This instruction is equiv-
lent to the assignment PC = Rm&0xFFFFFE. In fact, the
actual entry address of the Thumb function in memory
is Rm&0xFFFFFE, while the value of target address is
Rm&0xFFFFFE + 1. Therefore, all entry addresses of
the Thumb function are odd. In the next section we will use
this rule to improve the efficiency of the proposed algorithm.

In our experience, the first instruction in the common
function prologue is responsible for saving the original value
of registers, and is respective PUSH and STMFD in Thumb
and ARM state. The syntax of PUSH instruction is PUSH
<registers>. The PUSH instruction in the Thumb function
prologue is usually in the form of PUSH (Rn-Rm, LR). The
machine code of this instruction is XX B5 where the values
of XX vary with the “Rn-Rm” registers while the value of
B5 is constant [14]. The STMFD instruction in the ARM
function prologue is usually in the form of STMFD SP!,
[Rn-Rm]. Its machine code is XX XX 2D E9 where the
values of XX vary with the “Rn-Rm” registers in STMFD
instruction while 2D E9 is constant [14].

3.2 FIND-BASE Algorithm

To determine the image base of unknown format binary file
that contains FETs, we propose an algorithm called FIND-
BASE in this section. The main idea of the algorithm is
detailed follows. First, according to the lowest and highest
function entry addresses of FET and size of binary file, we
calculate the range of image base. Then we enumerate
all image base and lookup for each function prologue
according to the FET. If the function entry address is odd,
judge whether the corresponding bytes match the Thumb
function prologue; otherwise, if the function entry address
is even, judge whether the corresponding bytes match the
ARM function prologue. Second, we record the number
of matched function entry addresses (i.e. matched function
prologues) in every memory location. The memory loca-
tion where the percent of matched function entry addresses
exceeds a threshold is considered the candidate image base.

The FIND-BASE algorithm is illustrated in Fig. 4. First,
for function entry addresses in FET, remove duplica-
tions and sort in ascending order, and store the result into
array entry. The element entry[0] is the lowest function en-
try address of the FET, entry[n−1] is the highest function
entry address, and fileSize is the size of binary file. If the bi-
nary file is loaded into memory, it must include entry[n−1].
Assume entry[n−1] is the highest location that the binary
file is loaded in memory, we can calculate that the location
(entry[n−1]−fileSize) is the starting of the binary file
loaded in memory, this is the lowest location of the image
base, as shown in Fig. 4 (a). Since entry[0] is the lowest
address in FET, the location that the file loaded in memory
cannot higher than entry[0]. Hence, the location entry[0] is
considered the highest location of image base, as shown in
Fig. 4 (b).

Then we get the location range of image base is from
(entry[n−1]−fileSize) to entry[0], as shown in Fig. 4 (c)
where the array entry holds the sorted and de-duplicated
function entry addresses of FET in Fig. 1. Second, we pro-
cess each location in the range as follows: for each function
pointed by the function entry address in entry, we estimate
whether the bytes at the beginning match the machine code
of the prologue. If they match, the address is considered a matched function entry address. Then we record the number of matched function entry addresses in each location. More specifically, we define a variable \( x \) to record the possible image base location. The \( x \) varies from \( \text{entry}[n-1] - \text{fileSize} \) to \( \text{entry}[0] \). By changing the value of \( x \), we can traverse the range of image base. The offset of the j-th function \( \text{entry}[j] \) in binary file is \( \text{entry}[j] - x \). If \( \text{entry}[j] \) is odd, the corresponding function may be a Thumb function; then we judge whether or not the byte in function prologue matches the machine code of PUSH (i.e. 0xB5). Otherwise, if \( \text{entry}[j] \) is even, the corresponding function may be an ARM function. Then we judge whether or not the bytes in function prologue match the machine code of STMFD (i.e. 0x2D E9). Finally, the match rate \( M \) of function entry table is defined in Eq. (1).

\[
M = \frac{\text{The number of matched function entry addresses}}{\text{The number of all function entry addresses}} \tag{1}
\]

At a particular location in memory, if the match rate is greater than the given threshold of match rate \( T \), output this location as the candidate image base. The FIND-BASE algorithm is detailed as Algorithm 2.

**Algorithm 2 FIND-BASE algorithm**

1: function FindBase(binary file, function entry table, \( T \))
2: \( \text{bin[fileSize]} \leftarrow \text{binary file} \)
3: \( \text{entry}[n] \leftarrow \text{function entry addresses after deduplicated and sorted} \)
4: for \( x \leftarrow (\text{entry}[n-1] - \text{fileSize}) \), \( \text{entry}[0] \) do
5:   for \( j \leftarrow 0, n-1 \) do
6:     if \( \text{entry}[j] \% 2 == 1 \) then
7:       if \( \text{bin[entry}[j] - x] == 0xB5 \) then
8:         ThumbCount ++
9:       end if
10:      else
11:        if \( \text{bin[entry}[j] - x + 2] == 0x2D \)
12:          \&\& \( \text{bin[entry}[j] - x + 3] == 0xE9 \) then
13:          ArmCount ++
14:        end if
15:      end if
16:   end if
17: end for
18: if \( (\text{ThumbCount} + \text{ArmCount})/n > T \) then
19:   Output : \( x \)
20: end function

The time complexity of the FIND-BASE algorithm is \( O(\text{fileSize} - \delta) \ast n \) where \( \text{fileSize} \) is the size of the binary file, \( \delta \) is the difference between highest address and lowest
Table 4 The experiment results of FET and image base ($T=40\%$).

| Model                        | File          | The number of FETs | Maximal matched FETs | Base         |
|------------------------------|---------------|--------------------|----------------------|--------------|
| Sony NEX-7 Camera           | av-cam.bin    | 1639               | 281                  | C3421000     |
| Sony NEX-7 Camera           | vmlinux.bin   | 532                | 34                   | C0018000     |
| Canon EOS 6D DSLR           | 6d000116.tr   | 0                  | 0                    | N/A          |
| Samsung NX210 DSLR          | nx210.bin     | 18614              | 692                  | C0004000     |
| Cowon AW1 Car DVR           | update.bin    | 3415               | 113                  | C0007FSC     |
| Sony HDR-AS30 DV            | av-cam.bin    | 1706               | 313                  | C3431000     |
| Sony HDR-AS30 DV            | vmlinux.bin   | 795                | 60                   | C0018000     |
| Pebble Smart Watch          | tintin_fw.bin | 80                 | 7                    | 08010000     |
| Sony SmartWatch 2           | asw.bin       | 145                | 10                   | 08040000     |
| Samsung Gear Fit            | wingtip.ex.bin| 647                | 31                   | 60800000     |
| Samsung Gear Fit            | wingtip_in.bin| 515                | 98                   | 08040000     |
| Cowon J3 MP4                | cowon_j3_fw.bin| 2143              | 118                  | 20000000     |
| iAudio E2 MP3               | e2_equ_fwcsb  | 830                | 28                   | 00059000     |
| iAudio 10 MP3               | iAudio10_fw.bin| 1856              | 85                   | 20000000     |
| Sony SBH52 Headset          | sbh52_firmware.bin | 82          | 12                   | 08040000     |
| ADATA SP600 SSD             | sys_oie.bin   | 62                 | 0                    | N/A          |
| Crucial M500 SSD            | fwb1.img      | 13                 | 0                    | N/A          |
| SanDisk SDSSDHPP 256G       | firmware.bin  | 248                | 9                    | 7FFFF400     |
| Western Digital MyBook      | 20080111.bin  | 71                 | 26                   | 43000000     |
| Seagate ST350320AS          | sd1a2d.lod    | 63                 | 12                   | 0026A000     |
| Seagate DiamondMax 22       | mx1a4d.lod    | 57                 | 12                   | 0026A000     |

Fig. 5 The experiment results of av-cam.bin under different threshold of match rate ($T$).

address in the FET.

The output of FIND-BASE algorithm is the candidate image base for one FET. Then we perform the algorithm on all FETs of a binary file and do some statistics on the outputs. If the number of FETs corresponding to a particular candidate image base is much larger than others, we consider this candidate is actual image base.

4. Experiment Results and Analysis

Since there is no common test set can be used in our experiments, we collected multiple embedded firmwares, such as digital cameras, smart watches, solid-state drives (SSD) etc., from the Internet and created a test set to evaluate the validity of our algorithms. We implement our algorithms in C language. The experiments are performed in the PC with Pentium Dual-Core 3.0 GHz processor, 2 GB of memory, Microsoft Windows 7 SP1 and Visual C++ 6.0.

4.1 The Identifying of Function Entry Table

In this experiment, we firstly choose 21 unknown format binary files from the test set. We find that over 60% FET has more than 5 function entry addresses according to our experiment results. Therefore, the parameter $wnd$ is usually set to be 5 in experiments. The parameter $gap$ usually belongs to \{0, 1, 2, 3\}. The parameter $gap_{max}$ is the maximum value of the FET $gap$ in algorithm, we usually set $gap_{max} = 3$ in experiments. Then we run the FIND-FET algorithm on each binary files and identify FETs, the total number of FETs is shown in column “The number of FETs” of Table 4.

4.2 The Determination of Image Base and Case Analysis

As detailed in Sect. 3.2, the actual image base is the location where the number of matched FETs is much larger than other candidate image bases. In this experiment, an important step is to set the threshold of match rate ($T$). The setting of $T$ is affected by the type of instructions in function prologue. These instructions are influenced by the compiler tool chain, compiler options, etc. Taking av-cam.bin of Sony NEX-7 as an example, we discuss the impact of parameter $T$ on determining the image base. Figure 5 is a determination result of image base when parameter $T$ is assigned 30%, 40%, 50% and 60% respectively. When the value of $T$ changes, the trend of the four curves is similar and the number of matched FETs is most in the same memory location 0xC3421000 which is the correct image base. In other words, the image base determined by FIND-BASE algorithm does not change with the parameter $T$. When $T$ is lower (eg. 30%), the number of FETs corresponding to the correct image base is larger, and the number of FETs corresponding to wrong candidate image base also increases. When $T$ is higher (eg. 60%), the number of FETs corre-
The number of FETs corresponding to wrong candidate image base also decreases. This is because with the increase of $T$, some FETs of which match rate is lower than $T$ are filtered out, which causes the number of FETs corresponding to the correct image base decreases.

In the following experiments, the parameter $T$ is set to 40%. The number of FETs corresponding to actual image base is shown in column “Maximal matched FETs” and the actual image base is shown in column “Base” in Table 4. N/A means that the proposed algorithm is not available for this binary file. The reasons of N/A will be detailed in next section.

We analyzed the experiment results through a binary file named av-cam.bin in Sony NEX-7 Camera firmware. We firstly perform FIND-FET algorithm on av-cam.bin and find 1639 FETs. Then we respectively process each FET using FIND-BASE algorithm, the statistical results are illustrated in Fig. 6 (a). We can find the maximum number of matching location is $X = \text{C}3421000$, $Y = 281$. It means that there are 281 FETs whose candidate image base is $0x\text{C}3421000$. If $0x\text{C}3421000$ is the correct image base, we can affirm that these 281 FETs are actual. In other memory locations the number of matching FETs is far less than the one in $0x\text{C}3421000$. Hence, it can be inferred that the memory location $0x\text{C}3421000$ is the image base of av-cam.bin. Figure 6 (b), 6 (c) and 6 (d) are the experimental results of other firmwares, i.e. wingtip_in.bin of Samsung Gear Fit, e2_eu_fw.sb of iAudio E2 MP3 and 20080111.bin of Western Digital MyBook.

In order to verify whether or not the $0x\text{C}3421000$ is the actual image base, we load the file av-cam.bin with IDA Pro and set the processor type to be ARM little-endian and the image base to be $0x\text{C}3421000$. Then we randomly choose a function entry address in the 281 FETs, jump to it and disassemble with IDA Pro, we find a reasonable disassembly listing as well as nicely matched string references, and each binary function having matched prologue and epilogue. It indicates the image base determined by our algorithm is correct.

With the same method we verified other files in Table 4 and found all image bases are correct except for the file update.bin of Cowon AW1 Car DVR and firmware.bin of Sandisk SSD. This is because both the firmware files contain header. The firmware header usually includes some padding bytes and metadata such as firmware version, firmware compile time, magic number, checksum and so on. Some information in the header can be used to verify firmware in-
4.3 Possible Reasons for Determination Failure

Many different potential reasons may lead our algorithm to be unavailable in some binary files, but they can generally be slotted into the following areas.

(1) Some firmware binary files are packaged together with other types of files (such as configuration files, resource files, etc.). To use the proposed method we must extract the binary file. However since the custom file structure of some vendors is unknown, we cannot correctly extract the binary file, such as fwbl.img of Crucial M500.

(2) Some binary files in firmware are encrypted, such as 6d000116.fir in Canon EOS 6D Camera firmware. It will be failure to directly perform our method on such kind of files.

(3) Some binary files do not contain FET. For example, sys_oie.bin of ADATA SP600 SSD is an unencrypted and uncompressed binary file. Due to the lack of actual FET (all of the FETs corresponding to sys_oie.bin in Table 4 are dummy), the proposed algorithm cannot determine the image base.

4.4 0xB5 Trap and Solution

During the experiments, we found an abnormal phenomenon: in some accidental circumstances, the match rate of a FET is particularly high (>90%), but the image base determined according to the FET is proved wrong. We call the phenomenon as 0xB5 trap. Next, we analyze the generation reasons of the trap with a practical scenario.

The length of 58th FET of binary file av-cam.bin is 16. The number of memory locations of which the match rate (M) is 100% in the experiment is up to 18. The candidate image bases output by the FIND-BASE algorithm distribute relatively centralized in the memory and the match rate of some memory locations is higher than the threshold (T). For example, in the candidate image base of 0xF071692C, ThumbCount is 16, ArmCount is 0, n is 16 and match rate (M) of FET is 100%. Assume the location 0xF071692C is the correct image base, we can find that the disassembling result is disorganized. There is neither prologue and epilogue nor binary function structure, which indicate this location is not correct image base. In this case, the memory location pointed by the function entry addresses in 58th FET is shown in Fig. 8. Though there are a lot of 0xB5 in the solid line annotation region and function entry address exactly corresponds to the region, these 0xB5 in the region are not PUSH instruction machine code, which is called the 0xB5 trap. Hence, we do not determine image base by a single high match rate FET but count the results of all FETs in a binary file to avoid the 0xB5 trap.

5. Conclusion

In this paper, we propose an algorithm to determine the image base of binary file with unknown format based on the rules of FET and binary function prologue. First, we propose the FIND-FET algorithm to identify the FETs in a binary file. Second, we determine the candidate image base with our FIND-BASE algorithm. The actual image base is the candidate base that corresponds to the largest number of
FETs. Experiment results show that the proposed method is effective for determining image base of unknown format binary files in ARM firmware. At present, the image base determining of unknown format binary files that has no FET is still unsolved. For future work, it would be interesting to determine the image base in such kinds of binary files.

Acknowledgments

This research was supported by the National High Technology Research and Development Program of China (Grant No.2013AA01A212), National Natural Science Foundation of China (No.61370063, No.61272511).

References

[1] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, “Static disassembly of obfuscated binaries,” pp.255–270, USENIX Association, 2004. DOI: 10.1.1.108.8062.
[2] B. Schwarz, S. Debray, and G. Andrews, “Disassembly of executable code revisited,” pp.45–54, 2002. DOI: 10.1109/WCRE.2002.1173063.
[3] C. Linn and S. Debray, “Obfuscation of executable code to improve resistance to static disassembly,” pp.290–299, ACM, 2003. DOI: 10.1145/948109.9481149.
[4] C. Schuett, J. Butts, and S. Dunlap, “An evaluation of modification attacks on programmable logic controllers,” International Journal of Critical Infrastructure Protection, vol.7, no.1, pp.61–68, 2014. DOI: 10.1016/j.ijcip.2014.01.004.
[5] Z. Basnight, J. Butts, J. Lopez Jr., and T. Dube, “Firmware modification attacks on programmable logic controllers,” International Journal of Critical Infrastructure Protection, vol.6, no.2, pp.76–84, 2013. DOI: 10.1016/j.ijcip.2013.04.004.
[6] I. Skochinsky, “Intro to embedded reverse engineering for pc reversers,” REcon, 2010.
[7] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, “A large-scale analysis of the security of embedded firmware,” pp.95–110, USENIX Association, 2014.
[8] SAMSUNG, “Samsung opensource release center.” http://opensource.samsung.com/, accessed Feb. 12, 2014.
[9] ARM, “Arm compiler armasm user guide version 5.04.” 2013.
[10] C. Cifuentes and M. Van Emmerik, “Recovery of jump table case statements from binary code,” Science of Computer Programming, vol.40, no.2-3, pp.171–188, 2001. DOI: 10.1016/S0167-6423(01)00014-4.
[11] I. Dacosta, N. Mehta, E. Metrock, and J. Giffin, “Security analysis of an ip phone: Cisco 7960g,” in Principles, Systems and Applications of IP Telecommunications. Services and Security for Next Generation Networks, ed. H. Schulzrinne, R. State, and S. Niccolini, pp.236–255, Springer-Verlag, 2008. DOI: 10.1007/978-3-540-89054-6_12.
[12] C.R. Ramakrishnan, J. Rehof, G. Balakrishnan, and T. Reps, “Analyzing stripped device-driver executables,” in Tools and Algorithms for the Construction and Analysis of Systems, ed. C.R. Ramakrishnan and J. Rehof, Tools and Algorithms for the Construction and Analysis of Systems, pp.124–140, Springer Berlin Heidelberg, Budapest, Hungary, 2008. DOI: 10.1007/978-3-540-78800-3_10.
[13] J. Lee, T. Averinos, and D. Brumley, “Tie: Principled reverse engineering of types in binary programs,” 2011. DOI: 10.1.1.465.5046.
[14] ARM Limited, ARM Architecture Reference Manual, 2014.

Ruijin Zhu born in 1985, Ph.D. candidate in Beijing Institute of Technology, Beijing, China. His research interests revolve around security of embedded system, reverse engineering and digital forensics. 5 South Zhongguancun Street, Haidian District, Beijing Postcode: 100081, China.

Yu-an Tan has received Ph.D., professor and Ph.D. supervisor in Beijing Institute of Technology, senior member of China Computer Federation. His main research interests include network storage, storage security and embedded system.

Quanxin Zhang is on the faculty of the School of Computer Science and Technology at Beijing Institute of Technology. His main research interests focus on ad hoc network and mobile computing.

Fei Wu has received Ph.D., associate professor in Huazhong University of Science and Technology. Her main research areas are SOC, storage algorithm.

Jun Zheng has received Ph.D., vice professor in Beijing Institute of Technology, Beijing, China. Her research interests include information security and cloud computing.

Yuan Xue born in 1985, Ph.D. candidate in Beijing Institute of Technology. His main research interests include information security, vulnerability discovery and malicious code detection.