Hindawi
Wireless Communications and Mobile Computing
Volume 2021, Article ID 4664882, 10 pages
https://doi.org/10.1155/2021/4664882

Research Article

Determining the Image Base of ARM Firmware by Matching Function Addresses

Ruijin Zhu,[1] **Baofeng Zhang**,[1,2] **Yu-an Tan**,[3] **Yueliang Wan**,[4,5] **and Jinmiao Wang**[4,5]

[1]*China Information Technology Security Evaluation Center, Beijing 100085, China*
[2]*Tsinghua University, Beijing 100084, China*
[3]*School of Computer Science and Technology, Beijing Institute of Technology, Beijing 10081, China*
[4]*Run Technologies Co., Ltd. Beijing, Beijing 100192, China*
[5]*Beijing Engineering Research Center for Cyberspace Data Analysis and Applications, Beijing 100083, China*

Correspondence should be addressed to Jinmiao Wang; jinmiao_wang@163.com

Received 14 July 2021; Revised 1 September 2021; Accepted 25 October 2021; Published 17 November 2021

Academic Editor: Ding Wang

Copyright © 2021 Ruijin Zhu et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Firmware is software embedded in a device and acts as the most fundamental work of a system. Disassembly is a necessary step to understand the operational mechanism or detect the vulnerabilities of the firmware. When disassembling a firmware, it should first obtain the processor type of running environment and the image base of firmware. In general, the processor type can be obtained by tearing down the device or consulting the product manual. However, at present, there is still no automated tool that can be used to obtain the image base of all types of firmware. In this paper, we focus on firmware in ARM and propose an automated method to determine the image base address. Firstly, by studying the storage rule and loading mode of the function address, we can obtain the function offset and the function address loaded by LDR instruction, respectively. Then, with this information, we propose an algorithm, named Determining image Base by Matching Function Addresses (DBMFA), to determine the image base. The experimental results indicate that the proposed method can successfully determine the image base of firmware which uses LDR instruction to load function address.

1. Introduction

From mobile phones, smart bands, smart watches to routers, switches, solid-state disks, wireless sensor, etc., embedded systems have spread all over society [1, 2]. Recently, there have been many incidents related to security of firmware in embedded systems. For instance, NSA developed malware that infects hard disk firmware; Stuxnet targets SCADA systems and is believed to be responsible for causing substantial damage to Iran's nuclear program [3, 4]; Heartbleed vulnerability exists in the firmware of embedded systems, and many vendors release new versions of firmware to mitigate Heartbleed vulnerability [5]; at Recon BRX 2018, two researchers from Northeastern University reversed the firmware of Xiaomi's IoT devices and found vulnerabilities in the Xiaomi ecosystem [6]. The security issues of firmware in embedded systems are getting more and more attention [7–9], including firmware emulation [10–12], firmware testing [13], and uncovering vulnerabilities [14]. In order to improve the security of the embedded system, it is necessary to perform reverse engineering on the firmware [15].

When disassembling firmware, the disassembly tools such as IDA Pro need to know the processor type and image base. Correct image base allows the disassembler to establish accurate cross-references, which are important for firmware analysts to understand the firmware. At the same time, correct image base helps to understand the memory layout of the firmware as a whole, and the wrong image base leads to the instruction that references the immediate values to addressing failed [16].

To determine the image base of firmware, many researchers have put a great deal of effort and several manual solutions have been proposed.

Skochinsky [17] proposed a general principle for determining the image base of a file with an unknown format.

They suggested that some kinds of hints, such as self-relocating code, initialization code, and string tables, can be used.

Basnight et al. [16, 18] presented two methods for inferring an image base. The first method uses immediate values in instructions to infer a reasonable image base. And the second method uses a hardware debugger to halt a programmable logic controller to obtain a memory dump. Then, the image base can be found by manually analyzing common instruction patterns in the memory dump.

Dacosta et al. [19] noted that, when the case values in a switch-case statement of a C program are sequential and dense, the memory addresses of case are usually stored in a jump table; this fact can be used to infer the memory address of nearby code and eventually obtain the image base.

All the above methods require the intuition and experience of reverse engineers, and the success and effectiveness highly depend on the human factor. To address this problem, Zhu et al. proposed several methods to automatically determine the image base of ARM-based firmware [20–23]. However, these methods cannot determine the image base of all types of firmware, and some of them are time-consuming.

1.1. Contributions. According to statistics, about 63% of embedded devices are based on ARM architecture [24]. Hence, we focus on the firmware under ARM architecture in this paper. By studying the binary function in firmware and the loading method of its address, we propose a method for determining the image base of the firmware that loads the function address using *LDR* instruction. The method is divided into three steps. The first step is to identify all the binary functions and output their offsets. The second step is to identify all the function addresses in the firmware that might be loaded by *LDR* instructions. The third step is to determine the image base by using the binary function offset and the function address loaded by *LDR* instruction. The experimental results indicate that the proposed method is effective for firmware which uses *LDR* instruction to load function address. The method proposed in this paper can improve the efficiency of reverse engineering.

1.2. Roadmap. The rest of this paper is organized as follows. Section 2 introduces the binary function and the method of loading the function address and introduces the FIND-LDR algorithm to identify *LDR* instruction in the firmware and calculate its loaded address. Section 3 introduces the principle of determining the image base and gives the DBMFA (Determining image Base by Matching Function Addresses) algorithm for determining image base. Section 4 analyzes the experimental results with the real firmware as the test set. Finally, this paper is concluded in Section 5.

2. Binary Functions in Firmware and Their Addresses

2.1. Binary Functions in Firmware. Disassembling the binary file, we can obtain some binary functions which roughly correspond to functions in a high-level language [25]. When

```
int add (int a, int b)
{
    printf ("Hello ARM! \n");
    int c = a + b;
    return c;
}
```

Figure 1: Function of C source code.

```
. text : 00008968 ;   ======== S U B R O U T I N E =======
. text : 00008968
. text : 00008968
. text : 00008968   sub_8968
. text : 00008968       STMFD   SP!, {R3–R5, LR}                    Prologue
. text : 0000896C       MOV     R4, R0
. text : 00008970       LDR     R0, = (aHelloArm = 0×8980)
. text : 00008974       MOV     R5, R1
. text : 00008978       ADD     R0, PC, R0 ; "Hello ARM!"           Body
. text : 0000897C       BL      puts
. text : 00008980       ADD     R0, R4, R5
. text : 00008984       LDMFD   SP!, {R3–R5, PC}                    Epilogue
. text : 00008984 ; End of function sub_8968
. text : 00008984
```

Figure 2: Binary function of ARM.

```
int (*p) (int, int);
int add (int a, int b)
{
    int c = a + b;
    return c;
}
int main (int argc, char* argv [])
{
    p = add;
    printf ("sum = %d\n", p (3, 5));
    return 0;
}
```

Figure 3: Source code.

compiling a function, the compiler usually adds some instructions at the beginning to create and initialize stack frame and save registers. These instructions 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 [26–28].

We compile the C source code shown in Figure 1 into a binary file and then disassemble the binary file using IDA Pro; the results are shown in Figure 2. The binary function corresponding to the *add* function is divided into 3 parts, the prologue of which is the *STMFD* instruction, and the epilogue of the function is the *LDMFD* instruction.

Base on the above analysis, we write an IDA Pro script to obtain all binary functions in a firmware and output the offset of prologue of binary functions [26]. The script is shown in the appendix.

2.2. Binary Function Address Loaded by LDR Instruction. In ARM-based firmware, the compiler typically uses the *LDR* instruction to load the function address into the register. The C code snippet that defines a function pointer is shown

```
. text : 000082A4    STMFD    SP!, {R4, R11, LR}
. text : 000082A8    ADD      R11, SP, #8
. text : 000082AC    SUB      SP, SP, #0×C
. text : 000082B0    STR      R0, [R11, #var_10]
. text : 000082B4    STR      R1, [R11, #var_14]
. text : 000082B8    LDR      R3, = p
. text : 000082BC    LDR      R2, = 0×826C
. text : 000082C0    STR      R2, [R3]
. text : 000082C4    LDR      R4, = aSumD          ; "sum = %d\n"
. text : 000082C8    LDR      R3, = p
. text : 000082CC    LDR      R3, [R3]
. text : 000082D0    MOV      R0, #3
. text : 000082D4    MOV      R1, #5
. text : 000082D8    MOV      LR, PC
. text : 000082DC    BX       R3
. text : 000082E0    MOV      R3, R0
. text : 000082E4    MOV      R0, R4
. text : 000082E8    MOV      R1, R3
. text : 000082EC    BL       printf
. text : 000082F0    MOV      R3, #0
. text : 000082F4    MOV      R0, R3
. text : 000082F8    SUB      SP, R11, #8
. text : 000082FC    LDMFD    SP!, {R4, R11, LR}
. text : 00008300    BX       LR
```

Figure 4: Disassembly code.

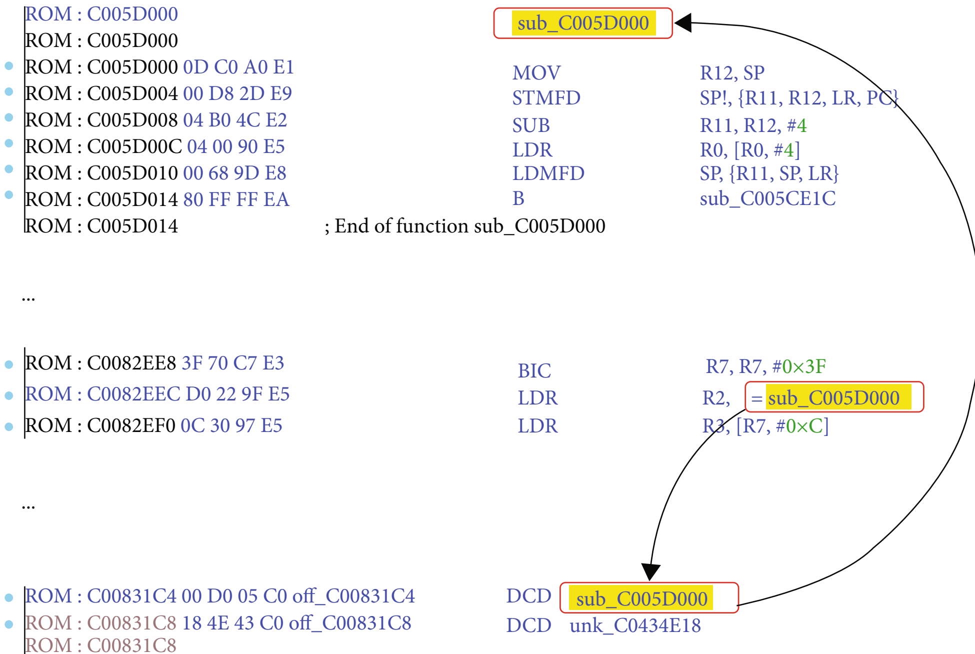
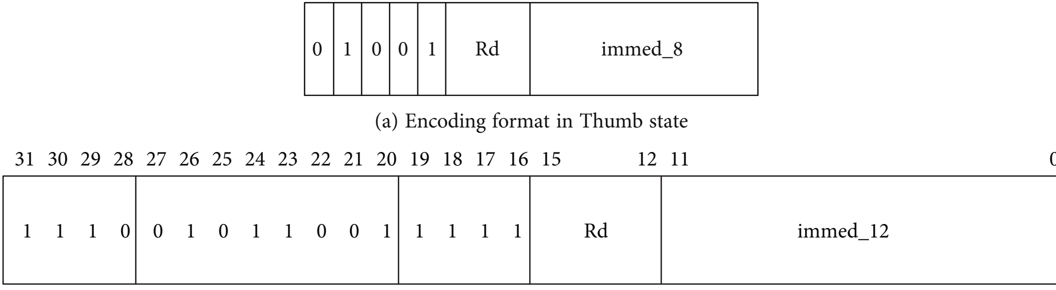


Figure 5: uImage of ABB NETA-21 (base address set to 0xC0008000).

in Figure 3. And the corresponding disassembly code is shown in Figure 4, where the assignment operation uses *LDR* instruction to load the function pointer, as shown in the red box. The compiler and disassembler in this article are arm-linux-gcc 4.3.2 and IDA Pro 6.8, respectively.

Next is an example of *LDR* instruction loading function address in real firmware. The disassembly code of ABB NETA-21 firmware *uImage* is shown in Figure 5.

The loading process of function address is detailed as follows. Take the *LDR* instruction at the memory address



(a) Encoding format in Thumb state

(b) Encoding format in ARM state

Figure 6: Encoding format of LDR instruction.

```
Input: binaryFile
Output: The addressees loaded by LDR instruction in ARM state.
function Find_ARM_LDR(binaryFile)
    bin[fileSize] ⟵ binaryFile
    offset ⟵0
      while(0 ≤ offset < fileSize-3) do
          if (bin[offset +2] ==0x9F && bin[offset+3] ==0xE5)
              PC ⟵ offset +8
              immed_12 ⟵ bit[11,...,0]
              address ⟵ PC&0xFFFFFFFC + (immed_12)
              Rd ⟵ Memory[address, 4]
              Output: Rd
          end if
          offset ⟵ offset +4
      end while
end function
```

Algorithm 1: FIND-ARM-LDR algorithm.

```
Input: binaryFile
Output: The addressees loaded by LDR instruction in Thumb state.
function Find_Thumb_LDR(binaryFile)
    bin[fileSize] ⟵ binaryFile
    offset ⟵0
      while(0 ≤ offset < fileSize) do
          opcode ⟵ bin[offset+1]
          opcode ⟵ opcode & (11111000)_2
          if( opcode == (01001000)_2)
            PC ⟵ offset +4
            immed_8 ⟵ bit[7,...,0]
            address ⟵ (PC & 0xFFFFFFFC) + (immed_8*4)
            Rd ⟵ Memory[address, 4]
            Output: Rd
          end if
          offset ⟵ offset +2
      end while
end function
```

Algorithm 2: FIND-Thumb-LDR algorithm.

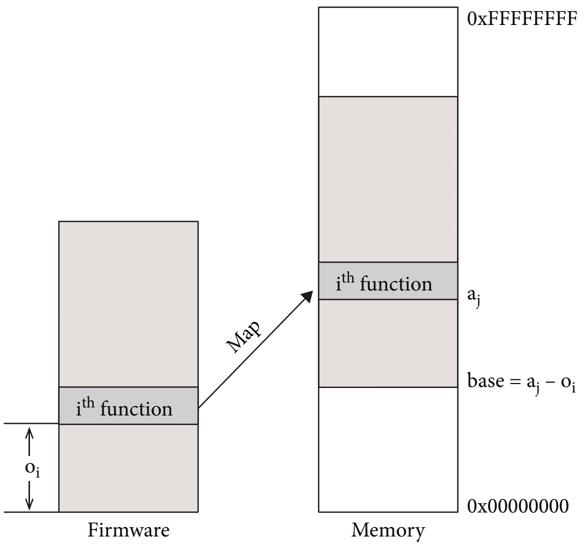


Figure 7: Mapping firmware into memory.

0xC0082EEC in Figure 5 as an example, the machine code for *LDR* instruction is D0 22 9F E5. Since this firmware is stored in the little-endian format, the actual machine code is E5 9F 22 D0. The format of *LDR* instruction used to load immediate values to a register in ARM state is *LDR* <*Rd*>, [*PC*, *#immed_12*], and the corresponding instruction encoding format is shown in Figure 6(b) [29].

Analysis of the encoding format of the *LDR* instruction and machine code yields $Rd = (0010)_2 = R2$, $immed_12 = (0010\,1101\,0000)_2 = 0x2D0$. The access address of the *LDR* instruction in the ARM state is $(PC\&0xFFFFFFFC) + immed_12$. Because an ARM processor uses three-stage pipeline technology, the value of *PC* equals the address of the current instruction plus 8 in the ARM state (i.e., $PC = Current + 8$). Then, the access address of the *LDR* instruction is given by

$$\begin{aligned} \text{address} &= (\text{PC}\&0\text{xFFFFFFFC}) + \text{immed_12} \\ &= ((\text{Current} + 8)\&0\text{xFFFFFFFC}) + \text{immed_12} \\ &= ((0\text{xC0082EEC} + 8)\&0\text{xFFFFFFFC}) + 0\text{x2D0} \\ &= (0\text{xC0082EF4}\&0\text{xFFFFFFFC}) + 0\text{x2D0} \\ &= 0\text{xC0082EF4} + 0\text{x2D0} \\ &= 0\text{xC00831C4}. \end{aligned} \tag{1}$$

Thus, the address accessed by *LDR* instruction is 0xC00831C4, as shown in Figure 5. The four bytes at the beginning of the memory address 0xC00831C4 are 00 D0 05 C0. Since the firmware is stored in the little-endian format, the actual address is 0xC005D000, which is, in fact, the address loaded by the *LDR* instruction. As shown in Figure 5, 0xC005D000 is the entry address of the binary function *sub_C005D000*.

The syntax and loading process of the *LDR* instruction in the Thumb state is similar to which in the ARM state; Figure 6(a) shows the corresponding encoding format.

Based on the above analysis, we introduce the FIND-Thumb-LDR algorithm and FIND-ARM-LDR algorithm [23] to scan all *LDR* instructions in the firmware and output its loaded addresses.

Note that the addresses loaded by *LDR* instructions are not all function addresses; they may correspond to string addresses, structure addresses, etc. However, the nonfunction entry addresses have no effect on the final determination of image base.

3. Determination of Image Base

The script described in Section 2.1 can obtain a set of binary function offsets in firmware, and the FIND-LDR algorithm described in Section 2.2 can obtain a set of memory addresses loaded by the *LDR* instruction in firmware. The address of some binary functions in the firmware is loaded into the register, usually for the assignment of the function pointer variable. If the function addresses are loaded by *LDR* instruction, there is a correspondence between some elements in the offset set of the binary function and some elements in the memory address set loaded by the *LDR* instruction, and the corresponding relationship between two sets can be used to determine the image base.

The set of binary functions obtained by the script in Section 2.1 is recorded as *F*, and the corresponding function offset set $(o_1, o_2, \cdots, o_n)(o_i < o_{i+1}(1 \le i \le n-1))$ is recorded as *O*, where *n* is the number of binary functions in the firmware. The memory addresses obtained by the FIND-LDR algorithm after removing the duplicate element are recorded as $A = (a_1, a_2, \cdots, a_m)$, where *m* is the number of addresses in set *A*.

As shown in Figure 7, if a binary function with offset o_i loaded into memory location a_j, assuming that the image base is *base*, then

$$o_i + \text{base} = a_j. \tag{2}$$

Therefore, the image base of the firmware is base $= a_j - o_i$.

The address of some binary functions in set *F* will be loaded by the *LDR* instruction, and some of the addresses in set *A* are the binary function addresses, so some of the elements in set *O* and some of the elements in set *A* match formula (1). When *base* is a particular memory address, the number of elements that satisfy formula (1) in set *O* and set *A* is the maximum, this memory address is considered to be the image base of the firmware.

In a 32-bit system, the memory range is large ($0 \sim 2^{32}$-1). We can obtain an image base by enumerating memory addresses in the range, but this method is less efficient. Therefore, we designed an algorithm to efficiently calculate the image base. The main idea of the algorithm is detailed as follows. First, we can determine the minimum value of image base is 0, and the maximum value of the image base

```
Input: Set of binary function offsets in firmware O = (o_1, o_2···,o_n)
       Set of addresses loaded by LDR instructions A = (a_1, a_2···,a_m)
       Size of the binary file fileSize
Output: Descending order of elements and the number of occurrences
function DBMFA(O, A, fileSize)
    Initialize each element in matrix M to -1
    max ⟵0xFFFFFFFF - fileSize
    for all a_j ∈ A do
        for all o_i ∈ O do
            if 0 < a_j - o_i && a_j - o_i < max
                M[j, i] ⟵ a_j - o_i
            end if
        end for
    end for
    Count the number of occurrences of each element in matrix M
    Sort the number of occurrences in descending order
    Delete the initial value -1 and the number of occurrences in the sort result
    Output: Descending order of elements and the number of occurrences
end function
```

Algorithm 3: DBMFA algorithm.

in a 32-bit system is 0xFFFFFFFFFF-*fileSize*, where *fileSize* is the size of the binary file. Second, subtract each element o_i in set O by each element a_j in set A. If the difference is in the range of image base, i.e., 0-0xFFFFFFFFFF-*fileSize*, it is saved; otherwise, it is discarded. Then, count the number of occurrences of each difference, sort in descending order by the number of occurrences, and output the results.

Assume *base* is a particular memory address, and set O and set A have k pairs of elements that conform to $o_i + \text{base} = a_j$. When subtracting each element in set O with the jth element in set A, a set is obtained.

$$\left(a_j - o_1, a_j - o_2, \cdots, a_j - o_n\right). \tag{3}$$

So that each element a_j in set A minus each element o_i of set O obtains a matrix M.

$$\begin{bmatrix} a_1 - o_1 & a_1 - o_2 & \cdots & a_1 - o_n \\ a_2 - o_1 & a_2 - o_2 & \cdots & a_2 - o_n \\ \vdots & \vdots & \vdots & \\ a_j - o_1 & a_j - o_2 & \cdots & a_j - o_n \\ \vdots & \vdots & \vdots & \\ a_m - o_1 & a_m - o_2 & \cdots & a_m - o_n \end{bmatrix}. \tag{4}$$

We have assumed that there is k pair of elements to satisfy formula (1), but the specific index of k pair of elements is unknown. Suppose (a_x, o_y) is one of k pairs of elements. Element $M_{[x,y]}$ on the xth row yth column of matrix M stores the value of $a_x - o_y$. Then, we count the number of occurrences of each element in matrix M, sort in descending order according to the number of occurrences of the elements, and output the result. Then, the most frequent occurrence of the element is the candidate image base. The practical significance of the candidate image base is that there is a correspondence between the most elements in set A and set O when the image base is this memory address. That is, the most binary function addresses in set F are loaded into registers. Based on the above analysis, we propose the Determining image Base by Matching Function Addresses (DBMFA) algorithm. For statistical purposes, the algorithm first initializes all elements of matrix M to -1.

The time complexity of the DBMFA algorithm is $O(n * m)$ where n is the number of binary function offsets and m is the number of addresses in set A.

The first memory address in the algorithm output (that is, the address with the most occurrences) is considered a candidate image base. If there exists one and only one candidate image base whose number of occurrences is much greater than those of the other candidate image bases, such candidate image base is considered to be the correct image base. Otherwise, the outputs do not contain the correct image base because the DBMFA algorithm cannot be applied successfully to the binary file.

4. Experimental Results and Analysis

Since there is no common test set that can be used in our experiments, we collected multiple firmware from some embedded devices, such as digital video cameras, smart watches, MP3 players, solid-state drives, and satellite phones, from the Internet, and created a test set to evaluate the validity of our algorithms. The DBMFA algorithms described above were written in python. The experiments were performed on a personal computer with an Intel i7-2600 3.40 GHz processor and 18 GB memory running Microsoft Windows 7 SP1.

Table 1: The experimental results.

Model	File	Function	ARM_LDR	Thumb_LDR	ALL_LDR	Match	Base	Time (s)
ABB NETA-21	uImage	7549	14611	2221	16723	226	0xC0008000	233
Advantech_EKI-2748FI	3551	3176	7350	879	8195	113	0x400000	41
Emerson TopWorx ES-03001	ES-03001-1.ffd	1668	2177	440	2599	84	0x1000FFD4	8
Pebble	tintin_fw.bin	1522	0	3152	3152	N/A	N/A	8
Phoenix 400 PND-4TX-IB	2985563_321.fw	6856	9867	897	10728	187	0x20800F28	116
Samsung gear fit	wingtip_in.bin	11619	0	4334	4334	N/A	N/A	99
Schneider 140CRA31200	CRA31200_Com.bin	7517	13980	1378	15284	377	0x1000	167
Schneider 140CRA31200	140CRA31200_ Master.bin	7185	13883	2176	15950	396	0x02001000	180
Sony AS30 DV	vmlinux.bin	4838	7993	1751	9676	265	0xC0018000	80
Sony SBH52	SBH52_firmware.bin	2949	4	1564	1568	73	0x8040001	8

4.1. Experimental Results. In the experiment, we choose the test set as the experimental object and then performed the method described in Section 2.1 to obtain the binary function offset in the firmware and performed the FIND-LDR algorithm to identify the address loaded by the LDR instruction. Table 1 shows the experimental results. Note that the column "Function" lists the number of binary functions, the column "ARM_LDR" and column "Thumb_LDR" are the number of addresses loaded by LDR instruction identified by the FIND-LDR algorithm in ARM and Thumb state, respectively. The column "ALL_LDR" lists the numbers of addresses identified by the FIND-LDR algorithm after duplicate elements are removed. The column "Match" lists the most frequent occurrence of the element in matrix M, i.e., the number of image base appears in matrix M. The "Base" column is the image base determined by the DBMFA algorithm. The symbol N/A means that the proposed algorithm is not available for this firmware, and the reasons for this are discussed in Section 4.3. "Time" column is the execution time of the DBMFA algorithm.

4.2. Case Studies. Take the firmware *uImage* of ABB NETA-21 as a case. According to the experiment results of Section 4.1, 7549 binary functions were identified in the firmware. The FIND-LDR algorithm identifies 14611 addresses loaded by *LDR* instruction in ARM state and 2221 addresses loaded by *LDR* instruction in Thumb state. The total is 16723 deduplicated addresses.

The results of the DBMFA algorithm are shown in Figure 8(a). As you can see from the figure, the peak point is $X = 0\text{xC0008000}$, $Y = 226$. That is, the memory address 0xC0008000 in matrix M appears 226 times, and its number of occurrences is much greater than other candidate image bases. Hence, 0xC0008000 is the image base. The practical significance is there are 226 pairs of data meeting $o_i + \text{base} = a_j$ at memory address 0xC0008000.

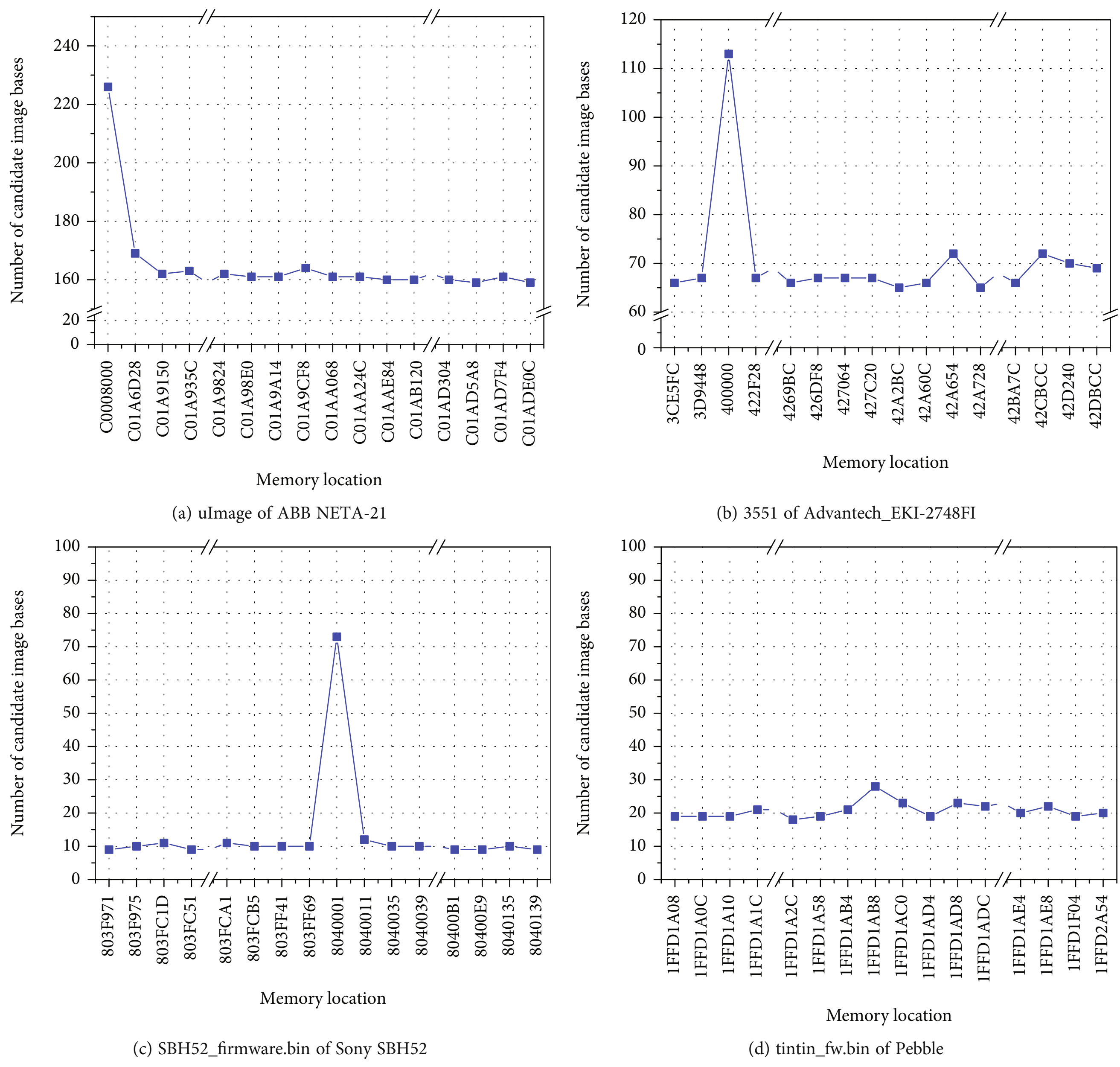
To verify whether the experimental results are correct, we load *uImage* file using IDA Pro and set the processor type to "ARM little-endian" and the image base to 0xC0008000. Then, IDA Pro can identify most binary functions, and some of the addresses loaded by the *LDR* instructions point to binary functions and display as function names. This means that the memory address 0xC0008000 is the correct image base while verifying that the 7549 binary functions in the firmware have 226 function addresses loaded into the register via the *LDR* instruction.

Figure 8(d) shows the experimental results obtained for the firmware samples *tintin_fw.bin* from Pebble smart watch. We can see that there is no sharp point in curve, which indicates that the algorithm proposed in this paper does not apply to this file.

Figure 8(c) is the experimental results of the *SBH52_firmware.bin* from Sony SBH52, which shows that its image base is 0x8040001. As we all know, image base of ARM firmware is 4 bytes aligned, but why this image base is 0x8040001? The reason is that 1564 LDR addresses in Thumb state are identified in the *SBH52_firmware.bin* firmware, while only 4 LDR addresses in ARM state are identified. When executing the *BX Rm* instruction, if the least significant bit (LSB) of the target address is 1, the processor 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 processor switch to the Thumb state. This instruction is equivalent to the assignment $PC = Rm\&0\text{xFFFFFFFE}$. In fact, the actual entry address of the Thumb function in memory is $Rm\&0\text{xFFFFFFFE}$, while the value of target address is $Rm\&0\text{xFFFFFFFE}+1$. Therefore, all entry addresses of the Thumb function are odd.

Some of the 1564 LDR addresses mentioned above are Thumb function addresses, and where the function address is its true value plus 1. This results in one-byte difference between the results and the true image base. The correct image base for the firmware *SBH52_firmware.bin* should be 0x8040000.

4.3. Reasons for Image Base Determination Failures. From Table 1, we can see that for some firmware, the image base



(a) uImage of ABB NETA-21

(b) 3551 of Advantech_EKI-2748FI

(c) SBH52_firmware.bin of Sony SBH52

(d) tintin_fw.bin of Pebble

Figure 8: Image base determination results.

is not determined successfully (even though it recognizes some *LDR* instructions). The possible reasons for this are as follows:

(1) Some firmware files are encrypted or compressed. These files must be decrypted or decompressed before applying the proposed methodology

(2) Due to different coding practices or compilation modes of a compiler, some firmware uses other instructions, such as *ADR* instruction, to load binary function addresses. In this case, the addresses identified by the FIND-LDR algorithm contain no binary function addresses. The algorithm proposed in this paper needs to utilize the binary function address loaded by *LDR* instruction, so it is not valid for such firmware, such as firmware *tintin_fw.bin* file of Pebble smart watch and firmware *wingtip_in.bin* file of Samsung gear fit

5. Conclusions

Disassembling for firmware is a necessary step in the analysis of embedded system security. Most of the firmware for unknown format cannot obtain the image base directly, which blocks the disassembly work. This paper studies the prologue of binary function and load method of binary function in ARM firmware and proposes a method for determining the image base by the binary function offset in firmware and the function address loaded by the *LDR* instruction. The experimental results

indicate that the method proposed in this paper is effective for the firmware of loading function addresses using *LDR* instruction. For other types of firmware, try using other methods to determine the image base, or manual method.

For the future work, it is interesting to explore the encoding of ARM instruction and propose the image base determination method for other types of ARM firmware. We will focus on automatically determining the image base of firmware of other architectures, such as MIPS and PowerPC.

Appendix

```
import csv # import the csv module
    print("start")
    # get the path to save the file
    csvFilePath = AskFile(True," *.csv", "Save to CSV File");
    if csvFilePath == None: # If no file is selected, exit the program
            Message("The user canceled the operation!")
            sys.exit()
    Message("csvFilePath = %s\n"%(csvFilePath)) # for debug
    csvFilePath = unicode(csvFilePath , "utf8") # for Chinese path
    FunCount =0                # variable FunCount is used to record the number of functions
    funcs = Functions() # walk through each function
    for f in funcs:
            name = Name(f) # obtain current function name
            offset = idaapi.get_fileregion_offset(f) # get offset of current function
            FunCount = FunCount + 1
            Message("FunCount = %d, function_name = %s, offset =0x%X\n"%(FunCount, name, offset))
            # save to csv file
            try:
                    with open(csvFilePath, 'ab+') as csvFile:
                            csvFile.write(codecs.BOM_UTF8)
                            spamwriter = csv.writer(csvFile, dialect='excel')
                            str = '{:X}'.format(offset) # format the str
                            spamwriter.writerow([str])
                            csvFile.close()
                     # close the CSV File
            except:
                    # get detail from sys.exc_info() method
                    error_type, error_value, trace_back = sys.exc_info()
                    print 'error_value = ', error_value # print error value
    print("done!")
```

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that there is no conflict of interest regarding the publication of this paper.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (Grant No. 61802439) and Beijing Youth Backbone Personal Project (Grant No. 201800002685XG357).

References

[1] C. Wang, D. Wang, Y. Tu, G. Xu, and H. Wang, "Understanding node capture attacks in user authentication schemes for wireless sensor networks," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, p. 1, 2020.

[2] S. Qiu, D. Wang, G. Xu, and S. Kumari, "Practical and provably secure three-factor authentication protocol based on extended chaotic-maps for mobile lightweight devices," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, p. 1, 2020.

[3] N. Falliere, L. O. Murchu, and E. Chien, "W32. Stuxnet Dossier," *Symantec Security Response*, 2010.

[4] Wikipedia, "Stuxnet-Wikipedia [EB/OL]," https://en.wikipedia.org/wiki/Stuxnet.

[5] C. Wang, D. Wang, and G. Xu, "Efficient privacy-preserving user authentication scheme with forward secrecy for industry 4.0," *Science China Information Sciences*, vol. 65, no. 1, article 112301, 2021.

[6] D. Giese and D. Wegemer, "Reversing IoT: Xiaomi ecosystem," in *RECON BRUSSELS 2018*, Brussels, Belgium, 2018.

[7] N. Redini, A. Machiry, R. Wang et al., "KARONTE: detecting insecure multi-binary interactions in embedded firmware," in *2020 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, 2020.

[8] A. Gazet, F. Périgaud, and J. Czarny, "HPE iLO5 firmware security-go home cryptoprocessor you're drunk![EB/OL]," https://www.blackhat.com/us-21/briefings/schedule/#hpe-ilo-firmware-security—go-home-cryptoprocessor-youre-drunk-23398.

[9] D. Ibdah, N. Lachtar, A. A. Elkhail, A. Bacha, and H. Malik, "Dark firmware: a systematic approach to exploring application security risks in the presence of untrusted firmware," in *USENIX Security Symposium*, pp. 413–426, San Sebastian, 2020.

[10] E. Johnson, M. Bland, Y. Zhu et al., "Jetset: targeted firmware rehosting for embedded systems," in *USENIX Security Symposium*, pp. 321–338, Vancouver, Canada, 2021.

[11] W. Zhou, L. P. Le Guan, P. Liu, and Y. Zhang, "Automatic firmware emulation through invalidity-guided knowledge inference," in *USENIX Security Symposium*, pp. 2007–2024, Vancouver, Canada, 2021.

[12] A. Mera, B. Feng, L. Lu, and E. Kirda, *DICE: automatic emulation of DMA input channels for dynamic firmware analysis*, 42nd IEEE Symposium on Security and Privacy, Los Alamitos, CA, USA, 2021.

[13] B. Feng, A. Mera, and L. Lu, "P2IM: scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *USENIX Security Symposium*, pp. 1237–1254, Boston, MA, United States, 2020.

[14] M. Elsabagh, R. Johnson, A. Stavrou, C. Zuo, Q. Zhao, and Z. Lin, "FIRMSCOPE: automatic uncovering of privilege-escalation vulnerabilities in pre-installed apps in Android firmware," in *USENIX Security Symposium*, pp. 2379–2396, Boston, MA, United States, 2020.

[15] D. Votipka, S. Rabin, K. Micinski, J. S. Foster, and M. L. Mazurek, "An observational investigation of reverse engineers' processes," in *USENIX Security Symposium*, pp. 1875–1892, Boston, MA, United States, 2020.

[16] Z. H. Basnight, *Firmware counterfeiting and modification attacks on programmable logic controllers*, Air Force Institute of Technology, Ohio, 2013.

[17] I. Skochinsky, "Intro to embedded reverse engineering for PC reversers," in *REcon Conference*, Montreal, Canada, 2010.

[18] 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.

[19] 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. IPTComm 2008*, H. Schulzrinne, R. State, and S. Niccolini, Eds., vol. 5310 of Lecture Notes in Computer Science, pp. 236–255, Springer, Berlin, Heidelberg, 2008.

[20] R. Zhu, Y. Tan, Q. Zhang, F. Wu, J. Zheng, and Y. Xue, "Determining image base of firmware files for ARM devices," *Transactions on Information and Systems*, vol. E99.D, no. 2, pp. 351–359, 2016.

[21] R. Zhu, Y. Tan, Q. Zhang, Y. Li, and J. Zheng, "Determining image base of firmware for ARM devices by matching literal pools," *Digital Investigation*, vol. 16, pp. 19–28, 2016.

[22] R. Zhu, B. Zhang, J. Mao, Q. Zhang, and Y. Tan, "A methodology for determining the image base of ARM-based industrial control system firmware," *International Journal of Critical Infrastructure Protection*, vol. 16, pp. 26–35, 2017.

[23] R. Zhu, B. Zhang, J. Mao, Y. Luo, Y. Tan, and Q. Zhang, "Determining image base of ARM firmware based on matching string addresses," *Acta Electronica Sinica*, vol. 45, no. 6, pp. 1475–1482, 2017.

[24] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A large-scale analysis of the security of embedded firmwares," in *Proceedings of the 23rd USENIX conference on security symposium*, San Diego, CA, 2014.

[25] J. Lee, T. Avgerinos, and D. Brumley, "TIE: Principled Reverse Engineering of Types in Binary Programs," in *NDSS*, San Diego, California, 2011.

[26] C. Eagle, *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*, No Starch Press, 2008.

[27] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *Proceedings of the 13th conference on USENIX security symposium-volume 13*, San Diego, CA, 2004.

[28] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *USENIX Security Symposium*, Austin, Texas, 2015.

[29] A. R. M. Limited, *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R Edition*, ARM Limited, 2014.