Hindawi
Wireless Communications and Mobile Computing
Volume 2020, Article ID 8899193, 12 pages
https://doi.org/10.1155/2020/8899193

Research Article

Determining the Image Base of Smart Device Firmware for Security Analysis

Ruijin Zhu,[1] **Baofeng Zhang**,[1,2] **Yu-an Tan**,[3] **Jinmiao Wang**,[4,5] **and Yueliang Wan**[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 100081, 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 28 June 2020; Revised 7 September 2020; Accepted 24 September 2020; Published 28 December 2020

Academic Editor: Ding Wang

The authorization mechanism of smart devices is mainly implemented by firmware, yet many smart devices have security issues about their firmware. Limited research has focused on securing the firmware of smart devices, although increasingly more smart devices are used to deal with the very sensitive applications, activities, and data of users. Thus, research on smart device firmware security is of growing importance. Disassembly is a common method for evaluating the security of authorization mechanisms. When disassembling firmware, the processor type of the running environment and the image base of the firmware should first be determined. In general, the processor type can be obtained by tearing down the device or consulting the product manual. However, it is not easy to determine the image base of firmware. Since the processors of many smart devices are ARM architectures, in this paper, we focus on firmware under the ARM architecture and propose an automated method for determining the image base. By studying the storage law of the jump table in the firmware of ARM-based smart devices, we propose an algorithm, named determining the image base by searching jump tables (DBJT), to determine the image base. The experimental results indicate that the proposed method can successfully determine the image base of firmware, which stores the absolute address in the jump table.

1. Introduction

Wireless technologies for smart devices are developing rapidly and are widely used. Smart devices have been deployed in several scenarios, such as smart phones, wearable devices, and vehicles. A recent marketing research report forecasted that the amount of smart devices will grow to approximately 10 billion in number worldwide by 2025 [1].

There have been a number of authorization security incidents caused by defects in firmware in recent years. For example, researchers found that several D-Link routers contain authentication backdoors by disassembling the firmware. If the attacker's browser user agent string is xmlset_roodkcableoj28840ybtide, then he/she can access the web interface of the device, bypassing the authentication procedure and viewing/changing the device settings [2]. A similar incident occurred on the Tenda router, in which an authentication backdoor was found by disassembling the firmware. The backdoor allows for the execution of commands remotely by sending them to specific strings and commands [3].

Unlike traditional embedded devices, smart devices are more vulnerable to attack. Some incidents [4–8] indicate that the security situation of smart devices is becoming increasingly serious, which has a profound impact on a country's economic and social development. Therefore, the security evaluation analysis and vulnerability assessment of smart devices are the primary considerations at present.

However, limited papers have been found that focus on securing the firmware of smart devices, although the firmware running on these smart devices is vulnerable to attack. Firmware provides the necessary instructions on how a smart device determines its functionality and communicates with

```
ROM:00000950 sub_950
ROM:00000950
ROM:00000950 var_18  = -0x18
ROM:00000950
ROM:00000950         STMFD    SP!, {R4,R5,LR}
ROM:00000954         SUB      SP, SP, #0xC
ROM:00000958         LDR      R5, =0xC00310A8
ROM:0000095C         LDR      R4, [R5]
ROM:00000960         CMP      R4, #0
ROM:00000964         BNE      loc_99C
ROM:00000968         LDR      R1, =0xC00310AC
ROM:0000096C         MOV      R2, #0x1000
ROM:00000970         LDR      R0, =0xC00300A8
ROM:00000974         BL       sub_111180
ROM:00000978         LDR      R12, =0xC0018E3C
ROM:0000097C         MOV      R3, R4
ROM:00000980         MOV      R2, R4
ROM:00000984         LDR      R0, =0xC023CF6E
ROM:00000988         LDR      R1, =0xC00300A8
ROM:0000098C         STR      R12, [SP,#0x18+var_18]
ROM:00000990         BL       sub_53354
ROM:00000994         MOV      R3, #1
ROM:00000998         STR      R3, [R5]
```

(a) The image base is set to 0

```
ROM:C0018950 sub_C0018950
ROM:C0018950
ROM:C0018950 var_18  = -0x18
ROM:C0018950
ROM:C0018950         STMFD    SP!, {R4,R5,LR}
ROM:C0018954         SUB      SP, SP, #0xC
ROM:C0018958         LDR      R5, =dword_C00310A8
ROM:C001895C         LDR      R4, [R5]
ROM:C0018960         CMP      R4, #0
ROM:C0018964         BNE      loc_C001899C
ROM:C0018968         LDR      R1, =unk_C00310AC
ROM:C001896C         MOV      R2, #0x1000
ROM:C0018970         LDR      R0, =unk_C00300A8
ROM:C0018974         BL       sub_C0129180
ROM:C0018978         LDR      R12, =sub_C0018E3C
ROM:C001897C         MOV      R3, R4
ROM:C0018980         MOV      R2, R4
ROM:C0018984         LDR      R0, =aEarlyOptions
ROM:C0018988         LDR      R1, =unk_C00300A8
ROM:C001898C         STR      R12, [SP,#0x18+var_18]
ROM:C0018990         BL       sub_C006B354
ROM:C0018994         MOV      R3, #1
ROM:C0018998         STR      R3, [R5]
```

data cross-references

code cross-references

(b) The image base is set to 0xC0018000

Figure 1: Comparison of incorrect and correct image base disassembly results.

other devices. The firmware can be obtained by downloading it from the website of the vendor or extracting it from the flash storage of the device hardware. Any firmware used in smart devices should be assumed insecure, which may have security vulnerabilities.

To evaluate and improve the security of firmware, a necessary method is disassembling [9, 10]. In this case, a disassembler, such as IDA Pro, needs to know the processor type and image base of the firmware [11]. In general, the processor type can be discerned by consulting the product manual or physical examination of the hardware [12, 13]. However, the image base cannot be obtained directly. Without the image base, the disassembler is unable to create cross-references based on absolute addresses [14]. When these cross-references are lacking, it is difficult to navigate efficiently in disassembly listing. Facing the obscure disassembly

code, people often lose their direction when they look for the assembly code in which they are most interested. Conversely, knowledge of the correct image base is critical in understanding the firmware as a whole [12].

Heterogeneous hardware architectures are used in firmware images; however, many smart devices are based on the ARM architecture [15–17]. Therefore, this work mainly focuses on ARM-based firmware. As shown in Figure 1, Figure 1(a) shows the disassembly code with the wrong image base and Figure 1(b) shows the disassembly code with the correct image base. IDA Pro cannot establish a cross-reference when the wrong image base is set, and the absolute addresses are marked in red. When the correct image base is set, IDA Pro establishes cross-references to these absolute addresses, which are important for reverse engineers to understand the intention of the assembly code.

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

Skochinsky [18] proposed a general principle for determining the image base of a file with an unknown format. He suggested that some kinds of hints, such as self-relocating code and initialization code, can be used.

Basnight et al. [12, 19] presented two methods for inferring the image base. The first method uses immediate values in instruction to infer a reasonable image base. The second method uses a hardware debugger to halt a programmable logic controller and obtain a memory dump. Then, the image base can be found by manually analyzing common instruction patterns in the memory dump.

Dacosta et al. [20] noted that when the case values in a switch-case statement of a C program are sequential and dense, the memory addresses of the case are usually stored in a jump table; this fact can be used to infer the memory address of the nearby code and eventually obtain the image base. Dacosta's approach manually analyzed the instruction of jump to default statement block (in this case, the BHI instruction) first, obtained the offset of the default statement block, and then analyzed the memory address of the default statement block to calculate the image base.

All of the above methods are not automated and heavily rely on reverse engineers' experience and intuition. We have proposed [21–23] three methods for automatically determining the image base. These automated methods are applicable to different types of ARM firmware, which cannot determine the image base of all types of firmware.

In this paper, we proposed a method for determining the image base of firmware that uses a jump table to store absolute addresses. The source code of firmware usually contains switch-case statements, and the compiler may generate jump tables for such code. By searching the sequence of instructions, the jump table can be located. Then, according to the absolute addresses in the jump table and the offset of the case statement block, we can obtain the image base. The experimental result indicates that the proposed method can effectively determine the image base of firmware that uses the jump table to store the absolute addresses.

```
switch(n)
  {
  case 0:
    printf("n =0\n");
    break;
  case 1:
    printf("n =1\n");
    break;
  case 2:
    printf("n =2\n");
    break;
  case 3:
    printf("n =3\n");
    break;
  case 4:
    printf("n =4\n");
    break;
  default:
    printf("default.\n");
}
```

Listing 1: Example of switch-case statements.

2. Jump Table in Firmware

The switch-case statement often appears in the source code of firmware and may generate a jump table after being compiled. After the code in Listing 1 is compiled into a binary file, IDA Pro can be used to disassemble the binary file, and the disassembly results are shown in Figure 2.

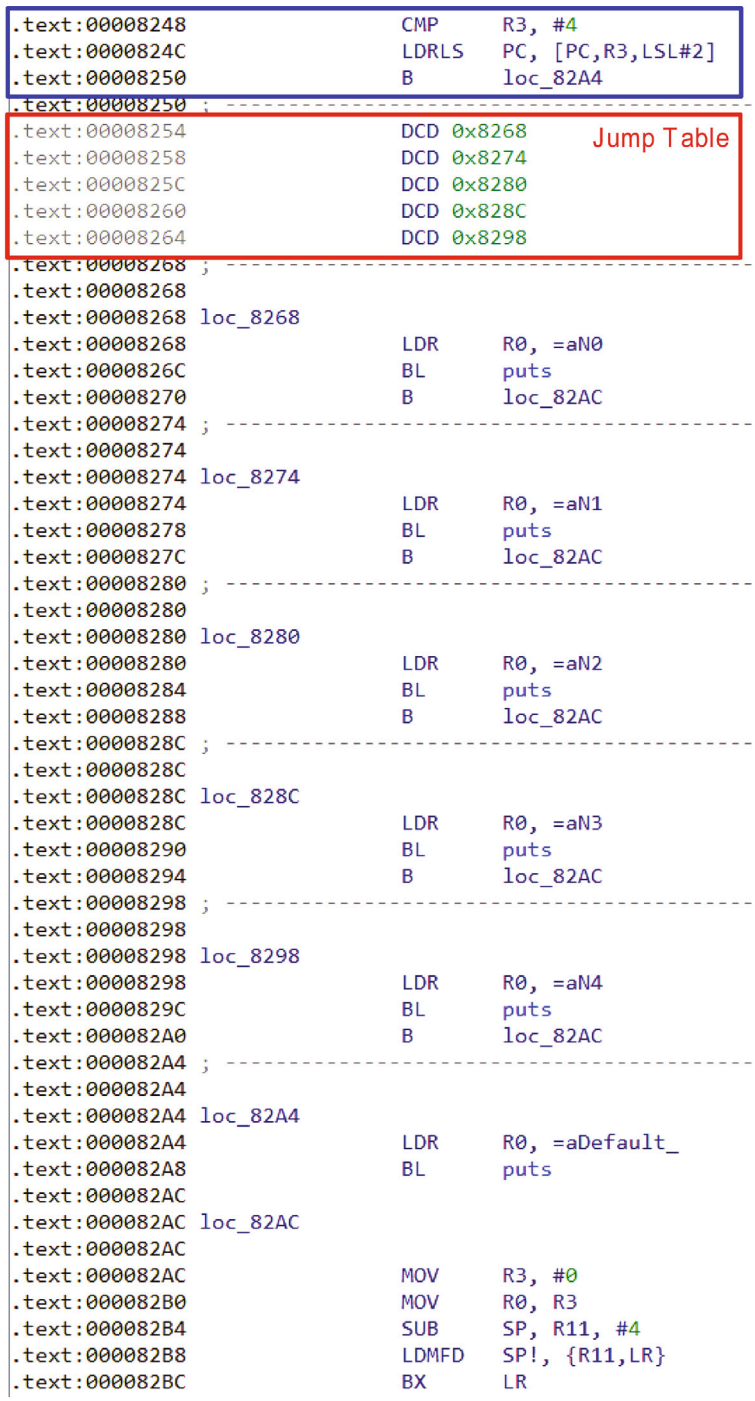
It can be seen that when there is a switch-case statement in the code, the compiler may generate a jump table. The content in the jump table is the addresses of the case statement block; for example, 0x8268 in the jump table is the address of the first case statement block.

Next, we analyze the calculation process of the jump table in two cases.

(1) Suppose that variable n in the code of Listing 1 is less than or equal to 4 (e.g., 3), then register R3 in the instruction at memory 0x8248 in Figure 2 is 3. After executing the instruction "CMP R3, #4" at offset 0x00008248, the LDRLS instruction is executed. According to the ARM manual [24], the memory address accessed by LDRLS is

$$\begin{aligned} \text{address} &= \text{PC} + (\text{R3} * 4) \\ &= (\text{Current} + 8) + (\text{R3} * 4) \\ &= (\text{0x824C} + 8) + (\text{0x3} * 4) \\ &= \text{0x8260}. \end{aligned} \tag{1}$$

As shown in Figure 2, the word at address 0x8260 is 0x828C. This means that the PC register will be assigned a value of 0x828C, and the program will jump to 0x828C to continue execution




```
.text:00008248                  CMP     R3, #4
.text:0000824C                  LDRLS   PC, [PC,R3,LSL#2]
.text:00008250                  B       loc_82A4
.text:00008250 ; ---------------------------------------------
.text:00008254                  DCD 0x8268
.text:00008258                  DCD 0x8274
.text:0000825C                  DCD 0x8280
.text:00008260                  DCD 0x828C
.text:00008264                  DCD 0x8298
.text:00008268 ; ---------------------------------------------
.text:00008268
.text:00008268 loc_8268
.text:00008268                  LDR     R0, =aN0
.text:0000826C                  BL      puts
.text:00008270                  B       loc_82AC
.text:00008274 ; ---------------------------------------------
.text:00008274
.text:00008274 loc_8274
.text:00008274                  LDR     R0, =aN1
.text:00008278                  BL      puts
.text:0000827C                  B       loc_82AC
.text:00008280 ; ---------------------------------------------
.text:00008280
.text:00008280 loc_8280
.text:00008280                  LDR     R0, =aN2
.text:00008284                  BL      puts
.text:00008288                  B       loc_82AC
.text:0000828C ; ---------------------------------------------
.text:0000828C
.text:0000828C loc_828C
.text:0000828C                  LDR     R0, =aN3
.text:00008290                  BL      puts
.text:00008294                  B       loc_82AC
.text:00008298 ; ---------------------------------------------
.text:00008298
.text:00008298 loc_8298
.text:00008298                  LDR     R0, =aN4
.text:0000829C                  BL      puts
.text:000082A0                  B       loc_82AC
.text:000082A4 ; ---------------------------------------------
.text:000082A4
.text:000082A4 loc_82A4
.text:000082A4                  LDR     R0, =aDefault_
.text:000082A8                  BL      puts
.text:000082AC
.text:000082AC loc_82AC
.text:000082AC
.text:000082AC                  MOV     R3, #0
.text:000082B0                  MOV     R0, R3
.text:000082B4                  SUB     SP, R11, #4
.text:000082B8                  LDMFD   SP!, {R11,LR}
.text:000082BC                  BX      LR
```

Figure 2: Disassembly code.

(2) When the value of variable n is greater than 4, i.e., the value of R3 is greater than 4, the instruction "B loc_82A4" at offset 0x00008250 will be executed. The program will jump to location loc_82A4 to continue execution

According to the above analysis, we can understand the calculation process of the jump table. Take the firmware of ABB NETA-21 as a case, as shown in Figure 3. The CMP instruction at offset 0x000AB124 is followed by the LDRLS instruction, the B instruction, and a jump table. The jump table begins at offset 0x000AB130 with four addresses, as shown by the red background in Figure 3, which are 0xC00B326C, 0xC00B3160, 0xC00B3150, and 0xC00B3140. In general, the minimum memory address in the jump table points to the first case statement block, and the first case statement block is usually next to the jump table. The minimum memory address in the jump table is 0xC00B3140, and the first case statement block starts at offset 0x000AB140. That is, the case statement block with offset 0x000AB140 is

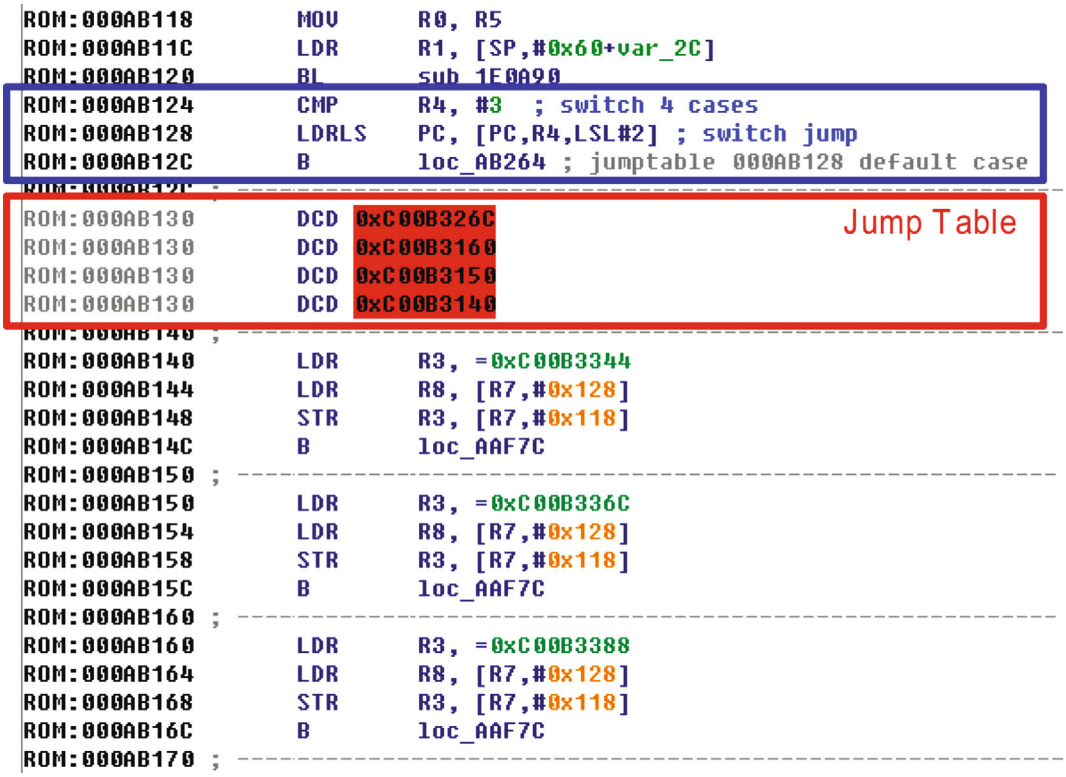


Figure 3: Jump table in ABB NETA-21 firmware uImage (the image base is set to 0).

mapped to the memory address 0xC00B3140, and then, the image base can be calculated.

3. DBJT Algorithm

According to the above analysis, when compiling the switch-case statement, the compiler usually generates the CMP instruction, LDRLS instruction, B instruction, and jump table in turn. The program jumps according to the addresses in the jump table. The model is shown in Figure 4.

In general, the minimum memory address in the jump table points to the first case statement block. A jump table can be used to deduce the memory address of the first case statement block; thus, the difference between the memory address and offset of the first case statement block can be used to obtain the candidate image base.

Figure 5 shows that the firmware that contains a case block with offset *offset_case*1 is mapped to memory. The image base of firmware is denoted as the *base*, and the minimum memory address in the jump table is denoted as *min_addr*. According to the analysis in Section 2, the first case block with offset *offset_case1* is mapped to memory location *min_addr*, i.e., $\text{min\_addr} = base + offset\_case1$, and then, we can obtain the image base as $base = \text{min\_addr} - offset\_case1$.

Based on the model of the switch-case statement, we can scan from the starting position of the firmware to locate the switch-case statement. If in a location, the current instruction is CMP, the second instruction is LDRLS, and the third instruction is B, then we consider it to be a switch-case statement, and the B instruction is followed by the jump table. Then, read in all the content of the jump table, obtain the minimum element of the jump table, and subtract the offset of the first case block from the minimum element to obtain a candidate image base. With one jump table, we can obtain a candidate image base. All candidate image bases can be calculated from all jump tables of the firmware. Then, we count the frequency of each candidate image base. If the frequency of a particular candidate image base is much larger than those of others, then we consider this candidate to be the actual image base. Based on the above analysis, we propose the determining the image base by searching jump tables (DBJT) algorithm to determine the image base. The pseudocode of the algorithm is shown in Listing 2.

The time complexity of the DBJT algorithm is *O(fileSize)*, where *fileSize* is the size of the firmware file. The algorithm first locates the jump table according to three consecutive instructions (CMP instruction, LDRLS instruction, and B instruction) and then sorts all the addresses in the jump table to obtain the minimum memory address. A candidate image base is obtained by the difference between the offset of the case statement block and the minimum memory address, and the candidate image base is added to multiset **M**. Finally, count the number of occurrences of each candidate image base in the multiset **M**, and then, sort them in descending order by occurrences. If a candidate image base appears much more frequently than other elements, then it is considered the correct image base. Otherwise, the outputs do not contain the correct image base because the DBJT algorithm cannot be applied successfully to this firmware.

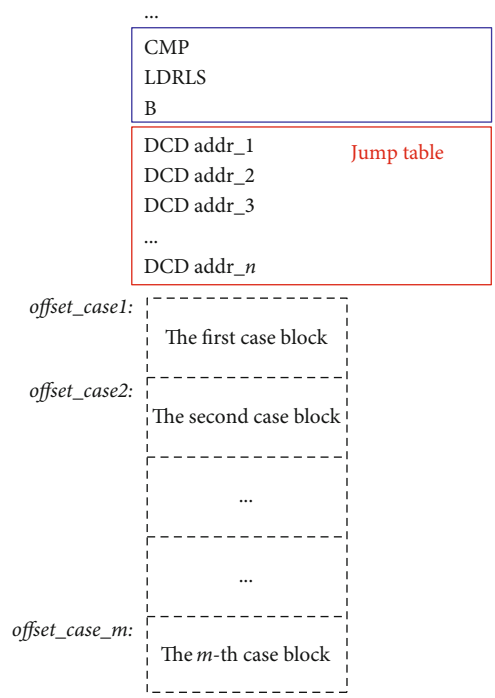


Figure 4: The assembly model of the switch-case statement.

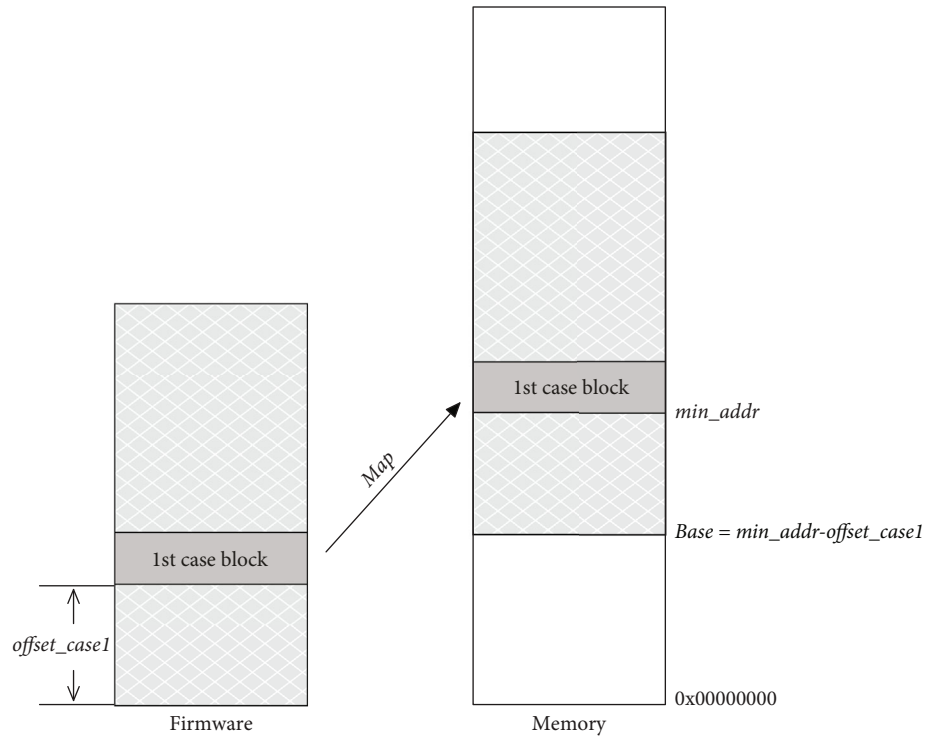


Figure 5: Map firmware into memory.

4. Experimental Results and Analysis

To test the proposed algorithm, we collected 10 firmware from well-known vendors' official websites. The DBJT algorithm was implemented in the C language and was compiled with Visual C++6.0. The experiments were performed on a personal computer with an Intel i7-2600 3.4 GHz processor and 18 GB memory, running Microsoft Windows 7 SP1.

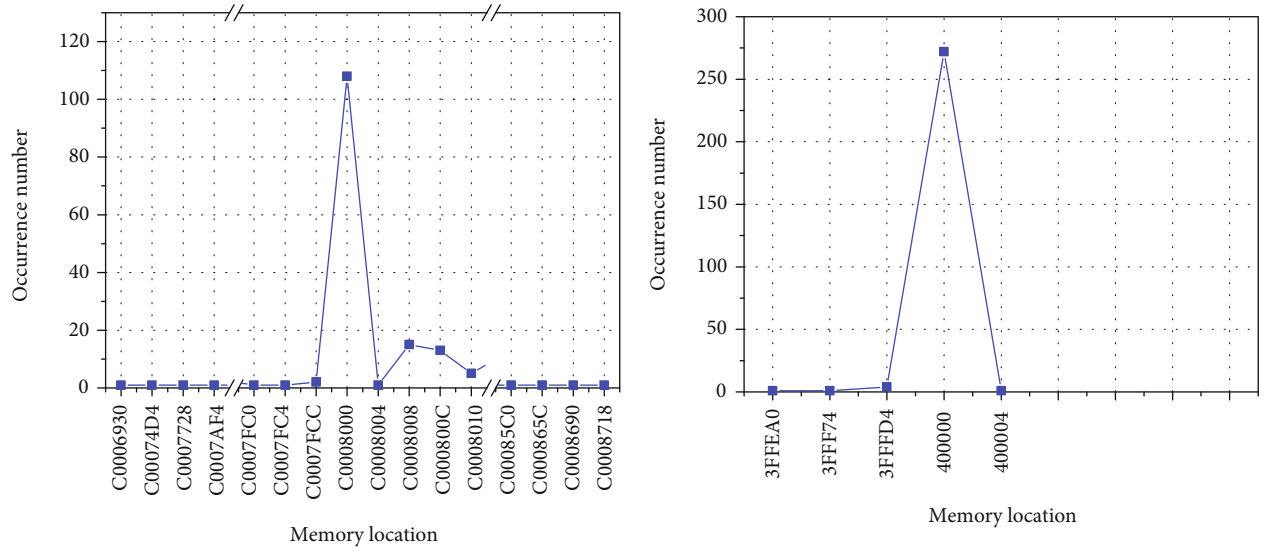
4.1. Experimental Results. In the experiment, the DBJT algorithm proposed in this paper is used to identify the jump table in the firmware and calculate the image base. The

```
Input: firmwareFile
Output: A sorted result of the elements and their occurrence in multiset M
function DBJT (firmwareFile)
  fileSize ⟵Obtain the size of firmwareFile
  offset ⟵0
  while(0 ≤ offset < fileSize) do
    CMP_FLAG ⟵ FALSE
    LDRLS_FLAG ⟵ FALSE
    B_FLAG ⟵ FALSE
    if Current instruction is CMP instruction, then
        CMP_FLAG ⟵ TRUE
    else
        offset ⟵ offset +4
        continue
    end if
    if The second instruction is LDRLS instruction, then
        LDRLS_FLAG ⟵ TRUE
    else
        offset ← offset +4
        continue
    end if
    if The third instruction is B instruction, then
        B_FLAG ⟵ TRUE
    else
        offset ⟵ offset +4
        continue
    end if
    if CMP_FLAG ==TRUE && LDRLS_FLAG == TRUE && B_FLAG == TRUE then
        jt[n] ⟵ Read the jump table
        min_addr ⟵ Obtain the minimum element of the array jt[n]
        offset_case1 ⟵ Obtain offset of the first case block
        base ⟵ min_addr - offset_case1
        if base % 4 ==0 then
            M ⟵ base
        end if
        offset ⟵ offset_case1
    end if
    offset ⟵ offset +4
  end while
  Count the number of occurrences of each element in the multiset M
  Sort the elements and their occurrence in descending order by number of occurrences
  Output: Sorted elements and their occurrences
end function
```

Listing 2:Determining the image base by searching jump tables (DBJT).

Table 1: Experimental results of the DBJT algorithm.

Device	Firmware	Jump table	Correct	Base	Time (ms)	Validated
ABB NETA-21	uImage	261	108	0xC0008000	250	Yes
Advantech 4570-CE	57791ec9.bin	222	38	0x7F000000	172	Yes
Advantech 2748FI Switch	3551.bin	279	272	0x00400000	93	Yes
Emerson ES-03001	es-03001-1.ffd	0	0	N/A	31	N/A
Phoenix 400 PND-4TX-IB	2985563_321.fw	448	437	0x20800F28	546	Yes
Phoenix OT 4 M Terminal	v1.23.nb0	0	0	N/A	15	N/A
Rockwell DriveLogix 5730	pn-82672.bin	0	0	N/A	47	N/A
Schneider 140CRA31200	cra31200.bin	318	153	0x00001000	156	Yes
Schneider 140CRA31200	140cra31200.bin	217	111	0x02001000	109	Yes
Schneider M241 PLC	vxBoot.bin	43	20	0x00801FC0	93	Yes



(a) uImage of ABB NETA-21

(b) 3551.bin of Advantech EKI-2748FI

Figure 6: Image base determination results.

```
ROM:C00A06C4      CMP    R3, #0xA
ROM:C00A06C8      LDRLS  PC, [PC,R3,LSL#2]
ROM:C00A06CC      B      loc_C00A0778
ROM:C00A06CC ; ---------------------------
ROM:C00A06D0      DCD loc_C00A0770
ROM:C00A06D0      DCD loc_C00A0764
ROM:C00A06D0      DCD loc_C00A0758
ROM:C00A06D0      DCD loc_C00A074C
ROM:C00A06D0      DCD loc_C00A0744
ROM:C00A06D0      DCD loc_C00A0738
ROM:C00A06D0      DCD loc_C00A072C
ROM:C00A06D0      DCD loc_C00A0720
ROM:C00A06D0      DCD loc_C00A0714
ROM:C00A06D0      DCD loc_C00A0708
ROM:C00A06D0      DCD loc_C00A06FC
ROM:C00A06FC ; ---------------------------
ROM:C00A06FC
ROM:C00A06FC loc_C00A06FC
ROM:C00A06FC      LDRB R3, [R12,#0xA]
ROM:C00A0700      MOV  R3, R3,LSL#24
ROM:C00A0704      ADD  R0, R0, R3
ROM:C00A0708
ROM:C00A0708 loc_C00A0708
ROM:C00A0708      LDRB R3, [R12,#9]
ROM:C00A070C      MOV  R3, R3,LSL#16
ROM:C00A0710      ADD  R0, R0, R3
ROM:C00A0714
ROM:C00A0714 loc_C00A0714
ROM:C00A0714      LDRB R3, [R12,#8]
```

(a) The image base is set to 0xC0008000

```
ROM:000986C4      CMP    R3, #0xA
ROM:000986C8      LDRLS  PC, [PC,R3,LSL#2]
ROM:000986CC      B      loc_98778
ROM:000986CC ; ---------------------------
ROM:000986D0      DCD 0xC00A0770
ROM:000986D0      DCD 0xC00A0764
ROM:000986D0      DCD 0xC00A0758
ROM:000986D0      DCD 0xC00A074C
ROM:000986D0      DCD 0xC00A0744
ROM:000986D0      DCD 0xC00A0738
ROM:000986D0      DCD 0xC00A072C
ROM:000986D0      DCD 0xC00A0720
ROM:000986D0      DCD 0xC00A0714
ROM:000986D0      DCD 0xC00A0708
ROM:000986D0      DCD 0xC00A06FC
ROM:000986FC ; ---------------------------
ROM:000986FC      LDRB   R3, [R12,#0xA]
ROM:00098700      MOV    R3, R3,LSL#24
ROM:00098704      ADD    R0, R0, R3
ROM:00098708      LDRB   R3, [R12,#9]
ROM:0009870C      MOV    R3, R3,LSL#16
ROM:00098710      ADD    R0, R0, R3
ROM:00098714      LDRB   R3, [R12,#8]
```

(b) The image base is set to 0

Figure 7: The disassembly result of the correct and incorrect image base.

experimental results are shown in Table 1. The column "Jump table" lists the number of jump tables identified by the DBJT algorithm in each firmware file. The column "Correct" lists the frequency of the correct image base identified by the DBJT algorithm, and the column "Base" lists the correct image bases of the corresponding firmware. The column "Time" lists the execution time of the proposed algorithm. The symbol N/A means that the method is not applicable to the corresponding firmware; the reasons for this are discussed in Section 4.2. The manual validation results are shown in the "Validated" column of Table 1.

We take the firmware uImage of ABB NETA-21 as an example to analyze the experimental results. As shown in Table 1, 261 jump tables are identified by the DBJT algorithm, 108 of which point to the same candidate image base 0xC0008000. Figure 6(a) shows the candidate image base and the corresponding occurrence frequency. It can be seen that the candidate image base 0xC0008000 appears 108 times, which is much higher than the frequency of other candidate image bases. The practical significance is that the candidate image base calculated by 108 jump tables is 0xC0018000. Therefore, we consider 0xC0018000 to be the correct image base of the firmware.

To verify whether the experimental results are correct, we load the firmware file uImage using IDA Pro and set the processor type to "ARM little-endian" and the image base to 0xC0008000. Then, we can see that the cross-references for absolute addresses in the disassembly code are correct, as shown in Figure 7(a). This indicates that the memory address 0xC0008000 is the correct image base. In comparison, the same file loaded by IDA Pro without setting the correct image base is shown in Figure 7(b).

As shown in Table 1, the execution time of the proposed algorithm for uImage is 250 ms. Compared to the time of reverse engineering, the time to determine the image base is insignificant.

Figure 6(b) shows the experimental results obtained for the firmware sample 3551.bin from the Advantech EKI-2748FI-managed Ethernet switch, the image base of which is 0x00400000, which is manually verified as the correct image base.

In Figure 6, we can see that there are some other points near the image base. These points are caused by errors in the algorithm. If the default statement block is in the first position in the switch-case statement, then the minimum memory address in the jump table no longer points to the first case statement block, and the default statement block is next to the jump table. This style of the C code is shown in Listing 3, and its corresponding assembly code is shown in Figure 8. Although such style of the C code is legitimate, most programmers never write in such style. This type of switch-case statement will lead to the inaccuracy of the DBJT algorithm, which will differ from the correct image base by a few bytes.

4.2. Possible Reasons for Determination Failure. In Table 1, the number of recognized jump tables in some firmware is 0, and the image base is not determined successfully, indicating that the DBJT algorithm is not suitable for this firmware. The possible reasons for this are as follows.

```
switch(n)
{
default:
    printf("default.\n");
    break;
case 0:
    printf("n =0\n");
    break;
case 1:
    printf("n =1\n");
    break;
case 2:
    printf("n =2\n");
    break;
case 3:
    printf("n =3\n");
    break;
case 4:
    printf("n =4\n");
    break;
    }
```

Listing 3: Example of switch-case statements.

(1) The compiler generates a jump table only when the value of the case in the switch-case is sequential and dense. Otherwise, the compiler generates no jump table. For example, the case value in Listing 4 is not sequential, and there is no jump table generated, as shown in Figure 9

(2) In some firmware, the jump table contains no absolute addresses, and the DBJT algorithm cannot be used to determine the image base, such as firmware es-03001-1.ffd of Emerson ES-03001, firmware v1.23.nb0 of Phoenix OT 4 M Terminal, and pn-82672.bin of Rockwell DriveLogix 5730. Figure 10 shows the assembly code of firmware es-03001-1.ffd

In Figure 10, the BHI instruction at address 0x00004E00 is the "Branch if Higher" instruction. Combined with the previous instruction, "CMP R1, #6," if R1 is greater than 6, then it will jump to the label def_4E0C. If R1 is less than or equal to 6 (e.g., 2), then the ADR instruction will be executed. The ADR instruction at address 0x00004E04 assigns register R2 to 0x00004E10. LDRB instruction loads a byte from memory. Then, R2 + R1 = 0x00004E10 + 0x2 = 0x00004E12. The 0x01 at address 0x00004E12 is loaded into register R2. The ADD instruction at address 0x00004E0C will modify the value of the PC register. The calculation process of the PC register is as follows:

$$\begin{aligned} PC &= PC + R2 * 4 \\ &= (Current + 8) + (R2 * 4) \\ &= (0x4E0C + 8) + (0x01 * 4) \\ &= 0x4E18. \end{aligned} \tag{2}$$

```
.text:00008248                  CMP     R3, #4
.text:0000824C                  LDRLS   PC, [PC,R3,LSL#2]
.text:00008250                  B       loc_8268
.text:00008250 ; ---------------------------------------
.text:00008254                  DCD 0x8274
.text:00008258                  DCD 0x8280
.text:0000825C                  DCD 0x828C
.text:00008260                  DCD 0x8298
.text:00008264                  DCD 0x82A4
.text:00008268 ; ---------------------------------------
.text:00008268
.text:00008268 loc_8268
.text:00008268                  LDR     R0, =aDefault_
.text:0000826C                  BL      puts
.text:00008270                  B       loc_82AC
.text:00008274 ; ---------------------------------------
.text:00008274
.text:00008274 loc_8274
.text:00008274                  LDR     R0, =aN0
.text:00008278                  BL      puts
.text:0000827C                  B       loc_82AC
.text:00008280 ; ---------------------------------------
.text:00008280
.text:00008280 loc_8280
.text:00008280                  LDR     R0, =aN1
.text:00008284                  BL      puts
.text:00008288                  B       loc_82AC
.text:0000828C ; ---------------------------------------
.text:0000828C
.text:0000828C loc_828C
.text:0000828C                  LDR     R0, =aN2
.text:00008290                  BL      puts
.text:00008294                  B       loc_82AC
.text:00008298 ; ---------------------------------------
.text:00008298
.text:00008298 loc_8298
.text:00008298                  LDR     R0, =aN3
.text:0000829C                  BL      puts
.text:000082A0                  B       loc_82AC
.text:000082A4 ; ---------------------------------------
```

Figure 8: Disassembly code.

```
switch(n)
{
case 1:
      printf("n =1\n");
      break;
case 100:
      printf("n =100\n");
      break;
default:
      printf("default.\n");
}
```

Listing 4: Example of switch-case statements.

That is, the PC register will be assigned the value 0x4E18. From the above calculation, it can be seen that there is no absolute address stored in the jump table, so the algorithm proposed in this paper cannot be used for this firmware.

5. Conclusions

The disassembly of firmware is a necessary step in the security assessment of authentication mechanisms. However, for the firmware of most smart devices, the image base cannot be obtained directly, which is a major obstacle to disassembly. In this paper, we research the storage law of the jump table in the ARM firmware of smart devices and

```
.text:00008240                 STR     R3, [R11,#var_8]
.text:00008244                 LDR     R3, [R11,#var_8]
.text:00008248                 STR     R3, [R11,#var_18]
.text:0000824C                 LDR     R3, [R11,#var_18]
.text:00008250                 CMP     R3, #1
.text:00008254                 BEQ     loc_8268
.text:00008258                 LDR     R3, [R11,#var_18]
.text:0000825C                 CMP     R3, #0x64
.text:00008260                 BEQ     loc_8274
.text:00008264                 B       loc_8280
.text:00008268 ; ---------------------------------------
.text:00008268
.text:00008268 loc_8268
.text:00008268                 LDR     R0, =aN1
.text:0000826C                 BL      puts
.text:00008270                 B       loc_8288
.text:00008274 ; ---------------------------------------
.text:00008274
.text:00008274 loc_8274
.text:00008274                 LDR     R0, =aN100
.text:00008278                 BL      puts
.text:0000827C                 B       loc_8288
.text:00008280 ; ---------------------------------------
.text:00008280
.text:00008280 loc_8280
.text:00008280                 LDR     R0, =aDefault_
.text:00008284                 BL      puts
.text:00008288
.text:00008288 loc_8288
.text:00008288
.text:00008288                 MOV     R3, #0
```

Figure 9: Disassembly code.

```
ROM:00004DFC                CMP             R1, #6
ROM:00004E00                BHI             def_4E0C
ROM:00004E04                ADR             R2, jpt_4E0C
ROM:00004E08                LDRB            R2, [R2,R1]
ROM:00004E0C                ADD             PC, PC, R2,LSL#2
ROM:00004E0C ; ---------------------------------------------------------
ROM:00004E10 jpt_4E0C       DCB 0x22 ; "                    Jump Table
ROM:00004E11                DCB 0x22 ; "
ROM:00004E12                DCB    1
ROM:00004E13                DCB    6
ROM:00004E14                DCB  0xF
ROM:00004E15                DCB 0x1B
ROM:00004E16                DCB 0x20
ROM:00004E17                DCB    0
ROM:00004E18 ; ---------------------------------------------------------
ROM:00004E18
ROM:00004E18 loc_4E18
ROM:00004E18                LDR             R1, =0x25AB02
ROM:00004E1C                LDRB            R0, [R1]
ROM:00004E20                CMP             R0, #2
ROM:00004E24                BNE             def_4E0C
ROM:00004E28                B               loc_4E74
ROM:00004E2C ; ---------------------------------------------------------
ROM:00004E2C
ROM:00004E2C loc_4E2C
ROM:00004E2C                LDRB            R0, [R4,#0x247]
ROM:00004E30                CMP             R0, #1
ROM:00004E34                BEQ             loc_4E58
ROM:00004E38                LDR             R0, [R4,#0x278]
ROM:00004E3C                BL              sub_6CEF4
```

Figure 10: Jump table in Emerson ES-03001 firmware es-03001-1.ffd (the image base is set to 0).

propose a method for determining the firmware image base by using a jump table. The experimental results show that the proposed method is effective for the firmware that stores the absolute addresses in the jump table. For future work, it is still a challenge to automatically determine the image base of other types of firmware, such as firmware that contains no jump table. We will continue to research new methods for other kinds of firmware in smart devices. We believe that these automated approaches can effectively reduce the difficulty of reverse analysis.

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] "IoT connections outlook|Mobility report - Ericsson," https://www.ericsson.com/en/mobility-report/reports/november-2019/iot-connections-outlook.

[2] "Reverse engineering a D-Link backdoor," http://www.devttys0.com/2013/10/reverse-engineering-a-d-link-backdoor/.

[3] "From China, with love," http://www.devttys0.com/2013/10/from-china-with-love/.

[4] W. Wang, X. Wang, D. Feng, J. Liu, Z. Han, and X. Zhang, "Exploring permission-induced risk in Android applications for malicious application detection," *s*, vol. 9, no. 11, pp. 1869–1882, 2014.

[5] W. Li, W. Meng, Z. Tan, and Y. Xiang, "Design of multi-view based email classification for IoT systems via semi-supervised learning," *Journal of Network and Computer Applications*, vol. 128, pp. 56–63, 2019.

[6] W. Wang, Y. Shang, Y. He, Y. Li, and J. Liu, "BotMark: automated botnet detection with hybrid analysis of flow-based and graph-based traffic behaviors," *Information Sciences*, vol. 511, pp. 284–296, 2020.

[7] W. Meng, W. Li, and L. Kwok, "EFM: enhancing the performance of signature-based network intrusion detection systems using enhanced filter mechanism," *Computers & Security*, vol. 43, pp. 189–204, 2014.

[8] Z. Guan, X. Liu, L. Wu et al., "Cross-lingual multi-keyword rank search with semantic extension over encrypted data," *Information Sciences*, vol. 514, pp. 523–540, 2020.

[9] L. Zhang, S. Hao, J. Zheng, Y. Tan, Q. Zhang, and Y. Li, "Descrambling data on solid-state disks by reverse-engineering the firmware," *Digital Investigation*, vol. 12, pp. 77–87, 2015.

[10] Z. Liu, Y. Huang, J. Li, X. Cheng, and C. Shen, "DivORAM: towards a practical oblivious RAM with variable block size," *Information Sciences*, vol. 447, pp. 1–11, 2018.

[11] P. Shirani, L. Collard, B. L. Agba et al., "BINARM: scalable and efficient detection of vulnerabilities in firmware images of intelligent electronic devices," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2018.

[12] Z. Basnight, J. Butts, J. Lopez, and T. Dube, "Firmware modification attacks on programmable logic controllers," *International Journal of Critical Infrastructure Protection*, vol. 6, no. 2, pp. 76–84, 2013.

[13] J. C. Mulder, M. D. Schwartz, M. J. Berg, J. R. Van Houten, J. M. Urrea, and A. N. Pease, "Reverse engineering industrial control system field devices," in *International Conference on Critical Infrastructure Protection*, Albuquerque, NM, USA, 2012.

[14] C. D. Schuett, *Programmable logic controller modification attacks for use in detection analysis*, DTIC Document, 2014.

[15] B. Chen, X. Dong, G. Bai, S. Jauhar, and Y. Cheng, "Secure and efficient software-based attestation for industrial control devices with arm processors," in *Proceedings of the 33rd Annual Computer Security Applications Conference*, New York, NY, USA, 2017.

[16] Y. J. Kwon, H. K. Kim, K. M. Koumadi, Y. H. Lim, and J. In Lim, "Automated vulnerability analysis technique for smart grid infrastructure," in *IEEE Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT)*, Washington, DC, USA, 2017.

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

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

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

[20] 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*, Springer-Verlag, 2008.

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

[23] 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.

[24] ARM Limited, *ARM Architecture Reference Manual*, ARM Limited, 2014.