INTERNATIONAL JOURNAL OF CRITICAL INFRASTRUCTURE PROTECTION 16 (2017) 26–35

Available online at www.sciencedirect.com

ScienceDirect

www.elsevier.com/locate/ijcip

A methodology for determining the image base of ARM-based industrial control system firmware

Ruijin Zhu[a,b], Baofeng Zhang[a], Junjie Mao[a], Quanxin Zhang[b,c], Yu-an Tan[b,c,*]

[a]China Information Technology Security Evaluation Center, Beijing 100085, China
[b]School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China
[c]Beijing Engineering Research Center for Massive Language Information Processing and Cloud Computing Applications, Beijing 100081, China

ARTICLE INFO

Article history:
Received 1 January 2016
Received in revised form
30 July 2016
Accepted 28 November 2016
Available online 3 January 2017

Keywords:
Industrial Control Systems
ARM Architecture
Firmware
Image Base
Reverse Engineering

ABSTRACT

A common way to evaluate the security of an industrial control system is to reverse engineer its firmware; this is typically performed when the source code of the device is not available and the firmware is not trusted. However, many industrial control systems are based on the ARM architecture for which the firmware format is always unknown. Therefore, it is difficult to obtain the image base of firmware directly, which significantly complicates reverse engineering efforts. This paper describes a methodology for automatically determining the image base of firmware of ARM-based industrial control systems. Two algorithms, FIND-String and FIND-LDR, are presented that obtain the offsets of strings in firmware and the string addresses loaded by LDR instructions, respectively. Additionally, the DBMSSL algorithm is presented that uses the outputs of the FIND-String and FIND-LDR algorithms to determine the image base of firmware. Experiments are performed with 10 samples of industrial control system firmware collected from the Internet. The experimental results demonstrate that the proposed methodology is effective at determining the image bases of the majority of the firmware samples.

1. Introduction

Industrial control systems are widely used in critical infrastructure assets such as water treatment plants, oil and gas pipelines, refineries and electric power grids. Traditionally, industrial control systems have been designed for operation in closed, trusted networks with little emphasis on security and limited protection mechanisms [5]. However, increased interconnectivity, especially connections to corporate networks and the Internet expose industrial control systems and the critical infrastructures they monitor and control to serious threats.

One example is Stuxnet, which, in 2010, targeted uranium hexafluoride centrifuges at Natanz in Iran [10]. In 2011, SCADA systems at water utilities in Illinois were hacked, which disrupted the water supply [8]. In 2014, the U.S. ICS-CERT [9] released a security bulletin about the Havex malware. Like Stuxnet, Havex was designed to attack industrial control systems; it supposedly has the ability to disable hydropower dams, overload nuclear power plants and even shut down power grids.

Statistics indicate that 92.6% of the vulnerabilities discovered in current industrial control systems are in software/firmware

*Corresponding author at: School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China.
E-mail address: tan2008@bit.edu.cn (Y.-a. Tan).

http://dx.doi.org/10.1016/j.ijcip.2016.12.002
1874-5482/

whereas only 7.4% are associated with hardware [11]. Any firmware used in industrial control systems should be assumed to be insecure because it may contain vulnerabilities and security flaws. Therefore, it is imperative to conduct security analyses and vulnerability discovery efforts for industrial control systems [12,13,18,20].

The security of firmware can be analyzed by reverse engineering [3,13,14,19,21]. When disassembling firmware, a tool such as IDA Pro needs to know the processor type and image base. In general, the processor type can be discerned by consulting the product manual or tearing down the device. If the firmware format is known, then the image base can be discerned. Unfortunately, most ARM firmware, which is widely used in modern industrial control systems, are binary files with unknown formats, so it is difficult to obtain the image bases directly. Armed with the correct image base, a disassembler can construct accurate cross references in instances where the address references use absolute addresses instead of offsets in a binary file [16]. The cross references, which include jump location references, function references, string references, etc., can be very helpful when attempting to navigate messy disassembled code. Hence, identifying the image base is important for reverse engineering efforts.

Several solutions have been proposed to obtain the image base of firmware with an unknown format. Skochinsky [17] has proposed a general technique for determining the image base of embedded system firmware; the technique leverages several hints such as self-relocating code and initialization code. Basnight et al. [2,4] have presented two methods for inferring an image base. The first method uses immediate values in firmware instruction and update files to infer a reasonable image base. The second method uses a hardware debugger to connect to and halt a programmable logic controller and obtain a memory dump. The image base is then found by manually analyzing common ARM instruction patterns in the memory dump.

Da Costa et al. [7] have noted that, when the case values in a `switch` statement of a C program are sequential and dense, the memory addresses of the cases are usually stored in a jump table; this fact can be used to infer the memory addresses of pieces of nearby code and eventually obtain the image base. Santamarta [15] describes another way to use a jump table. Since a jump table contains the absolute addresses of cases, the distances between the cases can be calculated. If there is a certain distance that is different from the others, the corresponding relation between the absolute address and offset of the case can be obtained, based on which, the base address can be determined.

These solutions for determining an image base need human interaction, namely the determination relies on the intuition and experience of the reverse engineer. Analysis of the literature reveals that only the methods described in [22,23] can automatically calculate the image base of firmware with an unknown format. However, the efficiency of these methods needs to be improved.

At present, most industrial control system firmware resides in embedded systems. According to Costin et al. [6], approximately 63% of embedded devices are based on the ARM architecture. Hence, this research focuses on industrial systems based on the ARM architecture and proposes a methodology for determining their firmware image bases. Firmware usually contains strings and the strings that are referenced in adjacent code are stored centrally. Therefore, to begin with, the FIND-String algorithm is presented for obtaining the string offsets used to calculate the numbers of bytes occupied by the strings. Since a compiler typically loads a string address into a register using the LDR instruction, the characteristics of the LDR encoding format are leveraged to specify the FIND-LDR algorithm that obtains the addresses of the strings loaded by LDR instructions. Next, the number of bytes occupied by the strings are calculated. Finally, using the numbers of bytes occupied by strings provided by the FIND-String and FIND-LDR algorithms, it is possible to discern the relationships between string offsets and memory addresses, which yield the image base of firmware.

This research has two main contributions. First, it work leverages the encoding of LDR instructions to effectively identify LDR instructions and calculate the address loaded by each LDR instruction. Second, a methodology is presented for determining the image base of industrial control system firmware with an unknown format. The methodology uses string offsets and string addresses loaded by LDR instructions to determine the image base. Experiments demonstrate that the methodology is very effective at determining the image base of firmware that uses LDR instructions to load string addresses.

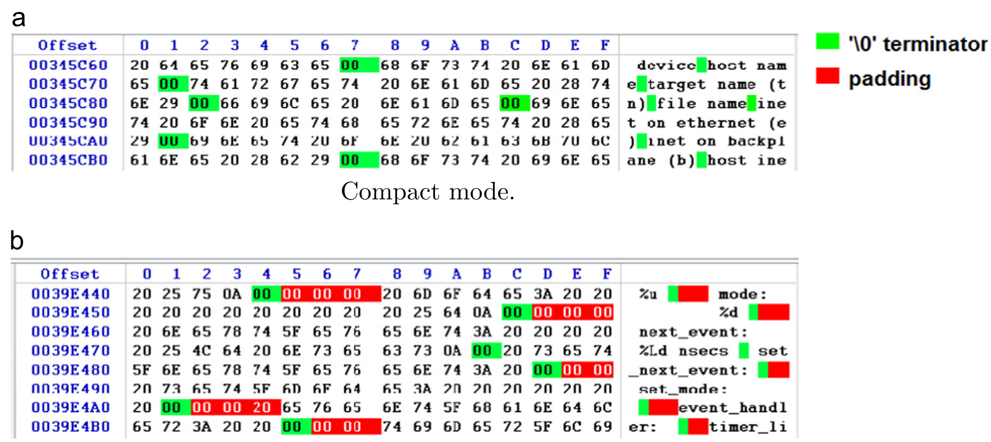
2. Strings and LDR instructions in firmware

This section discusses the storage features and loading process of strings in firmware. The FIND-String algorithm is presented for recognizing strings and outputting their offsets. Additionally, the FIND-LDR algorithm is presented for identifying LDR instructions in firmware and outputting the addresses loaded by LDR instructions.

2.1. *Identifying strings in firmware*

A binary file typically contains a number of strings, including prompt messages, error messages and version information. Each string contains some printable characters and escape characters. Printable characters include letters, numbers and punctuation; the ASCII range of these characters is `0x20` to `0x7E`. Escape characters include line breaks (`0x0a`), tabs (`0x09`) and others; the ASCII range of these characters is `0x09` to `0x0D`. Thus, the ASCII range of strings is [`0x09`, `0x0D`] ∪[`0x20`, `0x7E`]. Since the C language is most commonly used for developing industrial control system software, only C-style strings are discussed in this paper. In the C language, a string is usually stored in a character array whose last element is the string terminator "`\0'' with ASCII code 0x00`.

Figs. 1(a) and 1(b) show strings stored in the compact mode and aligned mode, respectively. In both storage modes, the strings are stored by the compiler. For performance reasons, some compilers store strings in the aligned mode. If the available storage position for a string is not a multiple of four bytes, the compiler adds some padding characters (`0x00` bytes) to create a storage position that is an exact



Compact mode.

4-byte aligned mode.

Fig. 1 – Two string storage modes displayed in WinHex.

multiple of four bytes as shown in Fig. 1(b). Note that, no matter whether strings are stored in the compact mode or in the aligned mode, they have the same feature—except for the first string, all the other strings are bracketed by `0x00` bytes.

The FIND-String algorithm leverages this feature to recognize strings in firmware and output their offsets. In binary files, the machine codes of some instructions correspond to the ASCII values of printable characters, which may cause some dummy identification. In order to improve the accuracy, a parameter *wnd* corresponding to the minimum length of a string is set; only a string whose length is greater than *wnd* is considered to be a valid string.

Algorithm 1. FIND-String algorithm.

```
Require: binaryFile, wnd
Ensure: offset
function FindString (binaryFile, wnd)
    bin[fileSize] ← binaryFile
    pos ← 1
    while 0 < pos < fileSize do
        if bin[pos − 1] == 0x00 & & IsPrint(pos) == TRUE
 then
            offset ← pos
            length ← 1
            while IsPrint(pos + length) == TRUE && bin[pos
 + length + 1] != 0x00 do
                length++
            if length > wnd && IsPrint(bin[pos + length])
 == TRUE then
                Output: offset
            pos ← pos + length
        else
            pos++

function IsPrint (c)
    if ((c >= 0x09 && c <= 0x0D) ∥ (c >= 0x20 &&
 c <= 0x7E)) then
        return TRUE
    else
        return FALSE
```

The FIND-String algorithm is specified as Algorithm 1. The algorithm scans a firmware file and identifies the starting and ending positions between two adjacent `0x00` bytes. The distance between the two positions is computed and the content between them is examined. If the distance is greater than *wnd* and the content only includes printable characters or escape characters, then the content between of the two positions is a string and the distance from the starting position to the beginning of the file is the string offset. In the case of a centrally stored string, it is difficult to determine the starting position of the first string because there are no `0x00` bytes before it, so the algorithm cannot proceed.

The FIND-String algorithm computes the following offsets for the strings in Fig. 1(b):

- Offset = `0x0039E448`
- Offset = `0x0039E460`
- Offset = `0x0039E47C`
- Offset = `0x0039E490`
- Offset = `0x0039E4A4`

These string offsets are used in Section 3 to determine the image base of industrial control system firmware. The next section describes how the addresses of strings loaded by LDR instructions are identified.

2.2. *Identifying the addresses of strings loaded by LDR instructions*

Extensive experimentation revealed that, in the case of ARM firmware, most string addresses are loaded into a register by the LDR instruction.

The loading process is clarified using an example. Fig. 2 shows a disassembly listing of the `uImage` firmware from an ABB NETA-21 Remote Monitoring Tool with the image base set to `0xC0008000`, which is the correct image base of the firmware. As shown in the figure, the memory address of the string `aModeD` is `0xC03A6448`. To load this address into the register, the compiler first stores the value `0xC03A6448` in the code segment `0xC0088198` and then loads `0xC03A6448` into

```
ROM:C0088050 44 10 95 E5         LDR   R1, [R5,#0x44]
ROM:C0088054 3C 01 9F E5         LDR   R0, =aModeD
ROM:C0088058 81 B0 0B EB         BL    sub_C0374264
ROM:C008805C 48 20 95 E5         LDR   R2, [R5,#0x48]
ROM:C0088060 4C C0 95 E5         LDR   R12, [R5,#0x4C]
ROM:C0088064 64 10 9F E5         LDR   R1, =0x3B9ACA00
ROM:C0088068 C2 3F A0 E1         MOV   R3, R2,ASR#31
ROM:C008806C 28 01 9F E5         LDR   R0, =aNext_eventLdNs
ROM:C0088070 9C 21 E3 E0         SMLAL R2, R3, R12, R1
ROM:C0088074 7A B0 0B EB         BL    sub_C0374264
ROM:C0088078 20 01 9F E5         LDR   R0, =aSet_next_event
ROM:C008807C 78 B0 0B EB         BL    sub_C0374264
ROM:C0088080 2C 10 95 E5         LDR   R1, [R5,#0x2C]
ROM:C0088084 0B 00 A0 E1         MOV   R0, R11
ROM:C0088088 1C FD FF EB         BL    sub_C0087500
ROM:C008808C 4C 00 9F E5         LDR   R0, =(a3PossibleIoFai+0x18)
ROM:C0088090 73 B0 0B EB         BL    sub_C0374264
ROM:C0088094 08 01 9F E5         LDR   R0, =aSet_mode
ROM:C0088098 71 B0 0B EB         BL    sub_C0374264
ROM:C008809C 30 10 95 E5         LDR   R1, [R5,#0x30]
ROM:C00880A0 0B 00 A0 E1         MOV   R0, R11
ROM:C00880A4 15 FD FF EB         BL    sub_C0087500
ROM:C00880A8 30 00 9F E5         LDR   R0, =(a3PossibleIoFai+0x18)
ROM:C00880AC 6C B0 0B EB         BL    sub_C0374264
ROM:C00880B0 F0 00 9F E5         LDR   R0, =aEvent_handler
ROM:C00880B4 6A B0 0B EB         BL    sub_C0374264

...

ROM:C0088194
ROM:C0088198 off_C0088198     DCD aModeD
ROM:C008819C off_C008819C     DCD aNext_eventLdNs
ROM:C00881A0 off_C00881A0     DCD aSet_next_event
ROM:C00881A4 off_C00881A4     DCD aSet_mode
ROM:C00881A8 off_C00881A8     DCD aEvent_handler
ROM:C00881A8

...

ROM:C03A6445                  DCB 0, 0, 0
ROM:C03A6448 aModeD           DCB " mode:           %d",0xA,0
ROM:C03A6448
ROM:C03A645D                  DCB 0, 0, 0
ROM:C03A6460 aNext_eventLdNs  DCB " next_event:     %Ld nsecs",0xA,0
ROM:C03A6460
ROM:C03A6460
ROM:C03A647C aSet_next_event  DCB " set_next_event: ",0
ROM:C03A647C
ROM:C03A648E
ROM:C03A6490 aSet_mode        DCB " set_mode:       ",0
ROM:C03A6490
ROM:C03A64A2
ROM:C03A64A4 aEvent_handler   DCB " event_handler:  ",0
ROM:C03A64A4
```

Fig. 2 – Disassembly listing of the `uImage` firmware from an ABB NETA-21 Monitoring Tool.

the register using the LDR instruction. Similarly, the addresses of the other strings in Fig. 2 (i.e., `aNext_eventLdNs`, `aSet_next_event`, `aSet_mode` and `aEvent_handler`) are loaded by LDR instructions. Fig. 1(b) shows these strings in storage.

Next, the details of how the LDR instruction loads string addresses are presented. Consider the string `aModeD`. The memory address and machine code of the LDR instruction used to load the address of string `aModeD` are `0xC0088054` and `3C 01 9F E5`, respectively. Since this firmware is stored in the little-endian format, the actual machine code is `E5 9F 01 3C`. The LDR instruction has multiple syntax formats [1]; however, the format used to load immediate values into the register in the ARM state is `LDR<Rd>,[PC,#immed_12]`. Fig. 3 (a) shows the corresponding encoding format.

Analysis of the encoding format of the LDR instruction and machine code yields:

```
Rd = (0000)_2
   = R0
```

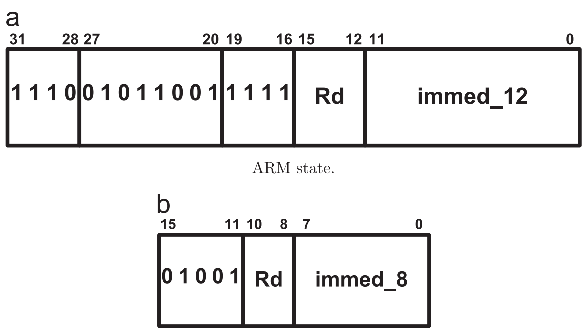
and

```
immed_12 = (0001 0011 1100)_2
         = 0x13C
```

The final 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 final address used by the LDR instruction to load string `aModeD` is given by:



ARM state.

Thumb state.

Fig. 3 – LDR encoding formats.

```
address = (PC & 0xFFFFFFFC) + (immed_12)
        = ((Current+8) & 0xFFFFFFFC) + (immed_12)
        = ((0xC0088054 + 8) & 0xFFFFFFFC) + 0x13C
        = (0xC008805C & 0xFFFFFFFC) + 0x13C
        = 0xC008805C + 0x13C
        = 0xC0088198
```

As shown in Fig. 2, the four bytes at the beginning of the memory address `0xC0088198` are `48 64 3A C0`. Since the firmware is stored in the little-endian format, the address is `0xC03A6448`, which is, in fact, the address used by the LDR instruction to load the string. As shown in Fig. 2, the actual content of the string `aModeD` is stored at address `0xC03A6448`.

As shown in Fig. 3(a), the starting bytes of the LDR instruction in the ARM state are `1110 0101 1001 1111`, which correspond to `0xE5 0x9F`. The syntax and loading process of the LDR instruction in the Thumb state is similar to that in the ARM state; Fig. 3(b) shows the corresponding encoding format.

Algorithm 2. FIND-ARM-LDR algorithm.

Require: *binaryFile*
Ensure: *Rd*
function FIND_ARM_LDR(*binaryFile*)
  bin [*fileSize*] ← *binaryFile*
  offset ← 0
  while 0 ≤ *offset* < *fileSize* **do**
    if *bin*[*offset* + 2] == `0x9F` && *bin*[*offset* + 3] == `0xE5` **then**
      PC ← *offset* + 8
      immed_12 ← *bit*[11, …, 0]
      address ← *PC* & `0xFFFFFFFC` + (*immed_12*)
      Rd ← *Memory*[*address*, 4]
      Output : *Rd*
    offset ← offset + 4

Algorithm 3. FIND-Thumb-LDR algorithm.

Require: *binaryFile*
Ensure: *Rd*
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` **then**
      PC ← *offset* + 4
      immed_8 ← *bit*[7, …, 0]
      address ← (*PC* & `0xFFFFFFFC`) + (*immed_8* * 4)
      Rd ← *Memory*[*address*, 4]
      Output : *Rd*
    offset ← *offset* + 2

Based on the encoding features and the analysis of the LDR instruction presented above, the FIND-LDR algorithm formally describes the identification of the LDR instruction in the ARM and Thumb states and calculates the address used by the LDR instruction. Because the LDR syntax format and loading method are different for the ARM and Thumb states, two algorithms are specified, Algorithm 2 named FIND-ARM-LDR and Algorithm 3 named FIND-Thumb-LDR, respectively.

Note that some outputs of the two FIND-LDR algorithms may be not string addresses. On the one hand, the addresses loaded by LDR instructions are not all string addresses; they may correspond to function addresses, structure addresses, etc. On the other hand, a portion of a binary file may exactly match an LDR instruction encoding, but this could correspond to data rather than an LDR instruction. Since the LDR instruction is shorter in the Thumb state, the FIND-Thumb-LDR algorithm yields more incorrect results than the FIND-ARM-LDR algorithm. However, the invalid results constitute only a small fraction of the overall results and do not affect the final determination of an image base.

3. Determining the image base

This section specifies an algorithm that determines the firmware image base by matching the string storage lengths. Specifically, the new algorithm named DBMSSL (Determining the Image Base by Matching String Storage Lengths) uses the string offsets computed by the FIND-String algorithm and the addresses used by the LDR instructions, which are provided by the FIND-LDR algorithm.

Definition 1: The number of bytes occupied by a firmware string, including the length of the string content L_s, length of the string terminator L_t and length of padding bytes L_p, is defined as the string storage length *SSL*, i.e., $SSL = L_s + L_t + L_p$. If the string is stored in the compact mode, then L_p is 0; otherwise, if it is stored in the aligned mode, then L_p generally is a value in {1, 2, 3}.

The FIND-String algorithm provides an offset set $O = \{o_1, o_2, \ldots, o_n\}$ and the corresponding string set $S = \{s_1, s_2, \ldots, s_n\}$, where n is the number of strings and $o_i < o_{i+1}$. By subtracting elements in O in turn, the string storage length set $D = \{d_1, d_2, \ldots, d_{n-1}\}$, i.e., $d_i = o_{i+1} - o_i$ is obtained.

Upon sorting the results provided by the FIND-LDR algorithm and removing duplicate elements, it is possible to obtain the address set $A = \{a_1, a_2, \ldots, a_m\}$ used by the LDR instructions, where m is the number of addresses. Subtracting

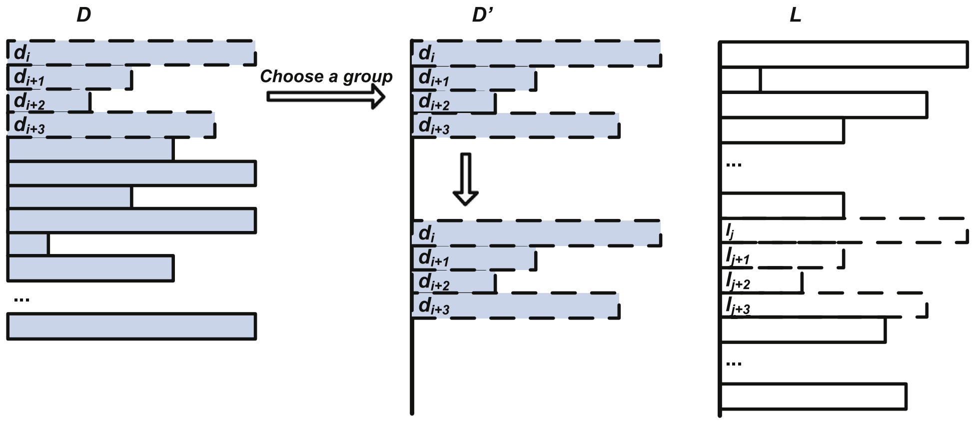


Fig. 4 – Searching for the position of group D' in L ($g=4$).

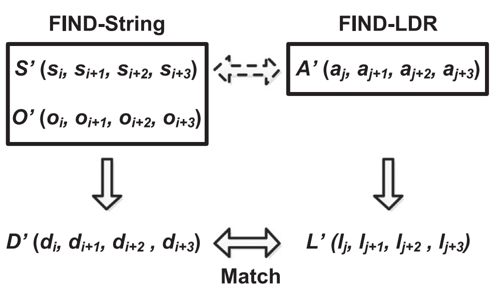


Fig. 5 – Correspondence between strings and their addresses.

the elements in A in turn yields the string storage length set $L=\{l_1, l_2, \ldots, l_{m-1}\}$.

In general, a compiler centrally stores the strings that are referenced in adjacent code; these strings are referred to as a string block. Suppose there are g consecutive strings in set S constituting a group $S'=\{s_i, s_{i+1}, \ldots, s_{i+g-1}\}$ ($g \ll n$ and $g \ll m$), which is a subset of a string block. Then, the corresponding offset set $O'=\{o_i, o_{i+1}, \ldots, o_{i+g-1}\}$ and the corresponding string storage length set $D'=\{d_i, d_{i+1}, \ldots, d_{i+g-1}\}$, where S', O' and D' are subsets of S, O and D, respectively.

Suppose the addresses of the strings in S' are loaded into registers by LDR instructions. Since the compiler usually stores strings that are referenced in adjacent code in a string block, the set D' is also a subset of set L. Next, starting from the first element of L, it is necessary to search for the position of D' in L. Fig. 4 illustrates this process for $g=4$.

At some point, if the elements in the subset $L'=\{l_j, l_{j+1}, \ldots, l_{j+g-1}\}$ are respectively equal to the elements in the subset D', i.e., $D'=L'$, then the search process is complete. Next, using the correspondence illustrated in Fig. 5, the strings in S' that are loaded into the memory addresses $A'=\{a_j, a_{j+1}, \ldots, a_{j+g-1}\}$ can be inferred, namely that a string s_i with offset o_i is loaded into memory address a_j.

As shown in Fig. 6, the address of a string loaded in memory is the image base plus offset (*address* = *image base* + *offset*), so the candidate image base is $base = a_j - o_i$. This process is repeated from the first g elements in D to the end in order to obtain multiple candidate image bases.

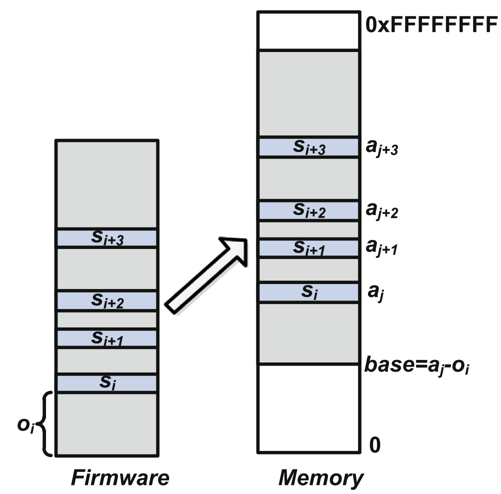


Fig. 6 – Mapping firmware into memory ($g=4$).

The strings shown in Fig. 2 are used to illustrate the execution of the DBMSSL algorithm (with $g=4$).

First, the FIND-String algorithm is applied to the `uImage` file to obtain the set of string offsets $O'=\{0x0039E448, 0x0039E460, 0x0039E47C, 0x0039E490, 0x0039E4A4\}$, from which the string storage length set $D'=\{0x18, 0x1C, 0x14, 0x14\}$ can be determined.

Next, the FIND-LDR algorithm is used to obtain the addresses loaded by LDR instructions. Upon sorting and removing the duplicate elements, set A={…, `0xC03A6448`, `0xC03A6460`, `0xC03A647C`, `0xC03A6490`, `0xC03A64A4`, …} is obtained. Following this, the set L={…, `0x18`, `0x1C`, `0x14`, `0x14`, …} is determined.

Next, the method detailed in Fig. 4 is used to find the set L' in L such that $L'=D'$. Using the position correspondence between D' and L, the strings with offset O' that are loaded into addresses $A'=\{0xC03A6448, 0xC03A6460, 0xC03A647C, 0xC03A6490, 0xC03A64A4\}$ can be inferred, as shown in Fig. 7. Finally, the candidate image base `0xC0008000` (=`0xC03A6448` – `0x0039E448`) is obtained.

Algorithm 4. DBMSSL algorithm.

Require: $O=\{o_1, o_2, \ldots, o_n\}$, $A=\{a_1, a_2, \ldots, a_m\}$, g

Ensure: *base*

```
function DBMSSL(O, A, g)
    for i ← 1, n − 1 do
        d_i ← o_{i+1} − o_i
    for j ← 1, m − 1 do
        l_j ← a_{j+1} − a_j
    for i ← 1, n − g do
        for j ← 1, m − g do
            flag ← EQUAL
            for k ← 1, g do
                if d_{i+k} != l_{j+k} then
                    flag ← NOT_EQUAL
                    break
            if flag == EQUAL then
                base ← a_j − o_i
                Output : base
                i ← i + g − 1
```

The steps described above constitute the DBMSSL algorithm, which is formalized as Algorithm 4. The time complexity of the DBMSSL algorithm is $O(nmg)$ where n is the number of string offsets in set O, m is the number of string addresses in set A and g is the number of strings in a group.

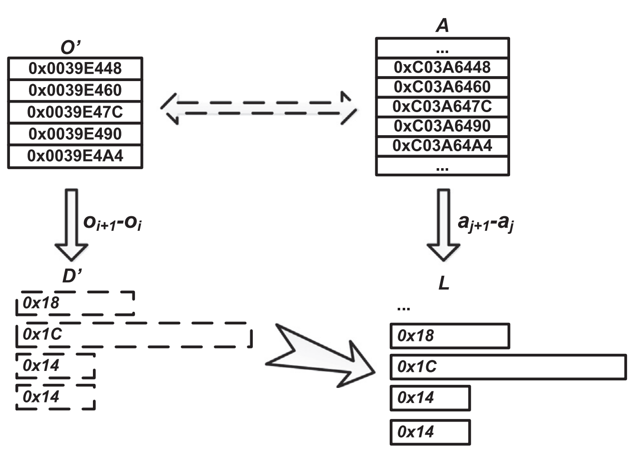


Fig. 7 – Determining the image base of `uImage` using the DBMSSL algorithm.

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 is considered to be the correct image base. Otherwise, the outputs do not contain the correct image base because the DBMSSL algorithm cannot be applied successfully to the binary file.

4. Experimental results and analysis

In order to test the methodology, 10 industrial control system firmware samples used in various devices (programmable logic controllers, switch, gateway, etc.) from well-known vendors were collected. The algorithms described above were written in the C language and were compiled with Visual C++ 6.0. The experiments were performed on a personal computer with a Pentium Dual-Core 3.0 GHz processor and 4 GB memory running Microsoft Windows 7 SP1 and IDA Pro 6.8.150423.

4.1. *Strings and addresses loaded by LDR instructions*

The first experiment applied the FIND-String and FIND-LDR algorithms to identify the strings in the 10 firmware samples and the addresses loaded by LDR instructions. Previous experimentation revealed that most of the strings have lengths greater than five; therefore, the parameter setting $wnd = 5$ was used in the experiments.

Table 1 shows the experimental results. Note that the column “Strings” lists the numbers of strings identified by the FIND-String algorithm and column “LDR Addresses” lists the numbers of addresses identified by the FIND-LDR algorithm after duplicate elements are removed.

4.2. *Determining the image base*

Setting the value of the parameter g is an important step in determining the image base. To test the impact of the parameter g on the determination of image base, the experiment used the `uImage` firmware from the ABB NETA-21 Remote Monitoring Tool with g values of 4, 5 and 6. Table 2 and Fig. 8 show the experimental results.

Fig. 8 shows that the trends of the three curves corresponding to $g = 4$, 5 and 6 are similar and the number of

Table 1 – Experimental results for $g = 4$.

Device	File Name	Strings	LDR Addresses	Candidate Image Bases	Maximum Occurrences	Image Base	Validated
ABB NETA-21	`uImage`	15,262	17,297	2238	1010	`0xC0008000`	Yes
Advantech 4570-CE	`57791ec9.bin`	11,234	29,410	1185	146	`0x7F000000`	Yes
Advantech 2748FI Switch	`3551`	4650	8496	519	365	`0x00400000`	Yes
Emerson ES-03001	`es-03001-1.ffd`	249	3120	3	0	N/A	N/A
Phoenix 400 PND-4TX-IB	`2985563_321.fw`	8694	11,114	294	195	`0x20800F28`	Yes
Phoenix OT 4 M Terminal	`v1.23.nb0`	464	896	58	54	`0xF0040000`	Yes
Rockwell DriveLogix 5730	`pn-82672.bin`	3951	4206	21	9	`0x00D00000`	Yes
Schneider 140CRA31200	`cra31200.bin`	22,984	15,504	4031	791	`0x00001000`	Yes
Schneider 140CRA31200	`140cra31200.bin`	23,464	15,880	4068	631	`0x02001000`	Yes
Schneider M241 PLC	`vxBoot.bin`	8185	5056	958	202	`0x00801FC0`	Yes

Table 2 – Experimental data for various g values.

g	Candidate image bases	Maximum occurrences	Percentage
4	2238	1010	45
5	1527	891	58
6	1059	739	70

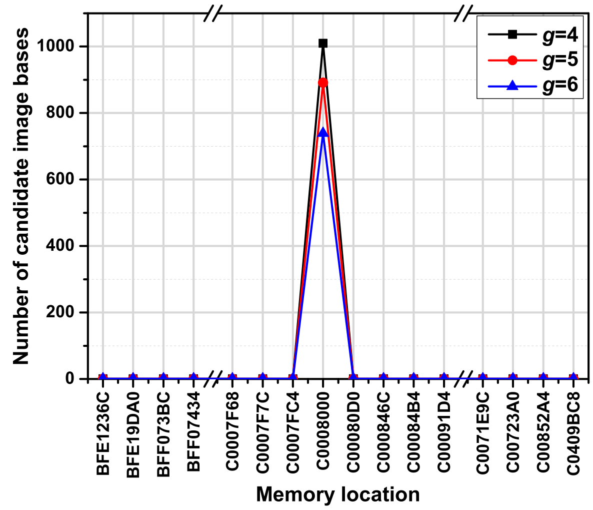


Fig. 8 – Experimental results for `uImage` with various g values.

occurrences of the candidate image base reaches its maximum at the same memory location `0xC0008000` – this corresponds to the correct image base.

The image base determined by the DBMSSL algorithm does not change for different values of g. As shown in Table 2, for g=4, the number of candidate image bases provided by the DBMSSL algorithm is 2238, for which the same candidate image base appears a maximum of 1010 times, corresponding to 45% of the DBMSSL algorithm outputs. When the value of g increases, the numbers of candidate image bases and the maximum occurrences decrease, but the correct image base percentages increase. This is because, when g increases, some smaller string blocks are filtered out, improving the correct image base percentage. In the remainder of the experiments described in this paper, g was set to 4.

Table 1 shows the image base determination results. Note that the column "Candidate Image Bases" lists the numbers of candidate image bases provided by the DBMSSL algorithm, the column "Maximum Occurrences" lists the numbers of correct image bases identified by the DBMSSL algorithm and column "Image Base" lists the correct image bases of the corresponding firmware samples. The N/A assignment means that the methodology is not applicable to the corresponding firmware; the reasons for this assignment are discussed in Section 4.3.

Firmware sample `uImage` from ABB NETA-21 is used as an exemplar for experimental analysis. According to Table 1, 15,262 strings were identified by the FIND-String algorithm and 17,297 addresses loaded by the LDR instruction were identified by the FIND-LDR algorithm.

Fig. 9(a) shows the image base determination results provided by the DBMSSL algorithm. In this figure, the maximum point of the curve is at (`0xC0008000`, 1010). The corresponding candidate image base `0xC0008000` appears 1010 times in the algorithm output, many more times than the other candidate image bases. The practical significance is that there are 1010 groups of size 4 that match the string storage length when the candidate image base is `0xC0008000`. Hence, it can be inferred that the memory location `0xC0008000` corresponds to the image base of the `uImage` firmware.

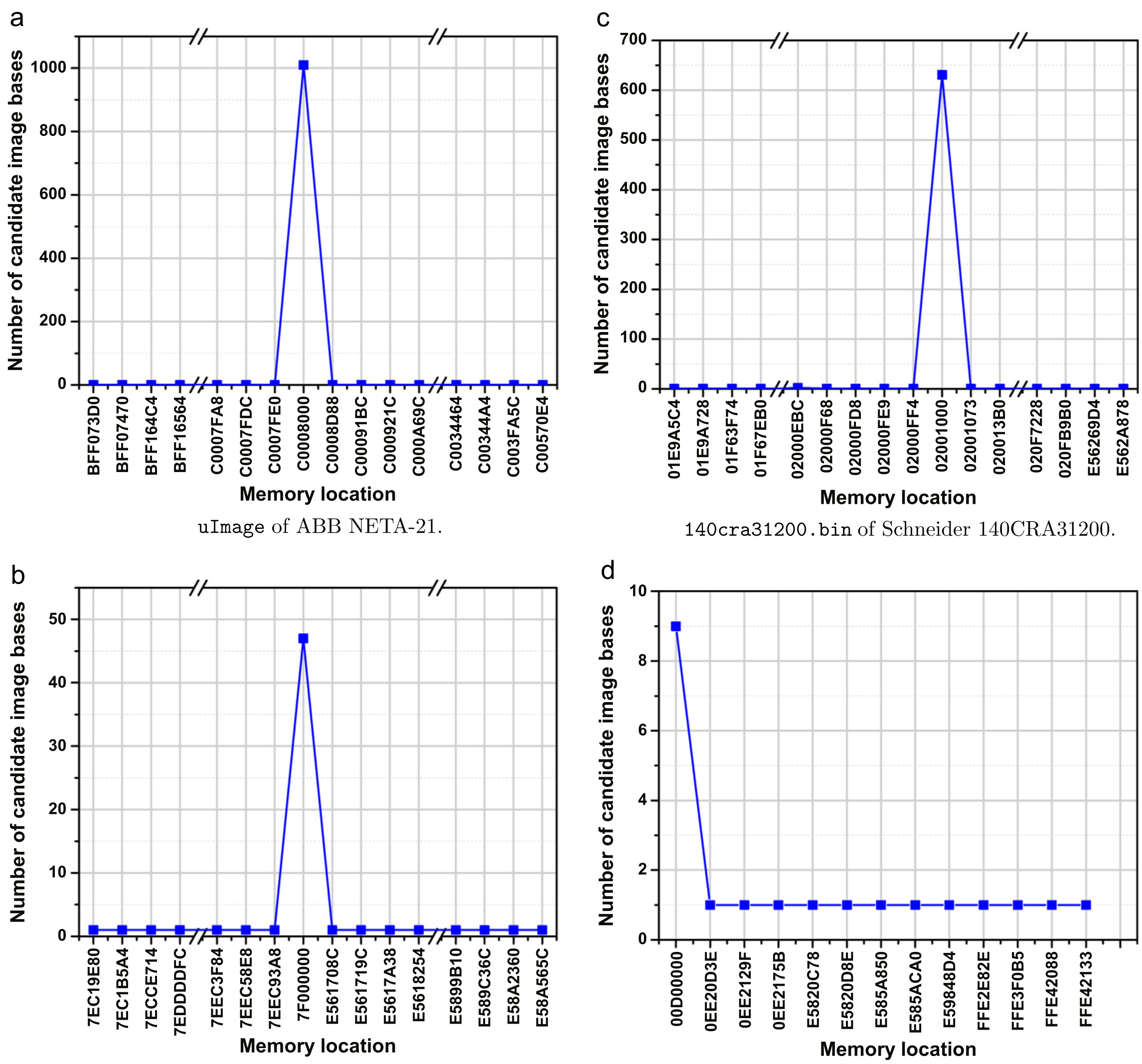
Figs. 9(b), 9(c) and 9(d) show the experimental results obtained for the firmware samples `57791ec9.bin` from Advantech 4570-CE, `140cra31200.bin` from Schneider 140CRA31200 and `pn_82672.bin` from Rockwell DriveLogix 5730, respectively. The corresponding image bases are at `0x7F000000`, `0x02001000` and `0x00D00000`, respectively.

It is possible to manually verify whether or not `0xC0008000` is the correct image base of the `uImage` firmware. This is accomplished by loading `uImage` into IDA Pro, setting the processor type to ARM little-endian and the image base to `0xC0008000`. Next, it is determined that the cross reference to the absolute memory address in the disassembly listing is correct and that the LDR instruction can display the correct string name when loading the string address. This indicates that the candidate image base at `0xC0008000` is, in fact, the correct image base. This methodology was used to verify the correctness of the image bases of the firmware samples in Table 1. The validation results are shown in the "Validated" column of the table.

4.3. *Reasons for image base determination failures*

For some firmware samples, the image base cannot be determined successfully by the DBMSSL algorithm, although the algorithm recognizes some strings. There are three possible reasons for a failure:

- Some firmware samples contain few or no strings, and they have no string blocks. The methodology described in this paper requires offsets and addresses of strings in the same string block to calculate the image base location. Thus, the methodology is not applicable to these firmware samples.
- Some firmware files are encrypted or compressed. These files must be decrypted or decompressed before applying the proposed methodology to determine the location of the image base.
- Some firmware samples use the ADR instruction to load a string address to a register, such as firmware `es-03001-1.ffd` from Emerson ES-03001 in Table 1. The ADR instruction adopts a relative addressing mode based on the PC register and does not require the absolute address of a string. When the FIND-LDR algorithm is applied to such firmware, the string addresses are not recognized. Since the DBMSSL algorithm needs the absolute addresses of strings to calculate the candidate image base, the proposed methodology is not applicable to these firmware samples.



`uImage` of ABB NETA-21.

`140cra31200.bin` of Schneider 140CRA31200.

`57791ec9.bin` of Advantech 4570-CE.

pn_`82672.bin` of Rockwell DriveLogix 5730.

Fig. 9 – Image base determination results for $g = 4$.

5. Conclusions

Reverse engineering firmware is a popular and robust method for identifying security weaknesses and vulnerabilities in industrial control systems. This paper has presented a novel methodology for determining the image bases of firmware samples with unknown formats. The methodology uses the FIND-String algorithm to obtain the offsets of strings in a firmware sample. Following this, the FIND-LDR algorithm is used to obtain the addresses loaded by LDR instructions. Finally, the DBMSSL algorithm uses the results of the FIND-String and FIND-LDR algorithms to determine the image base. The experimental results and manual verification demonstrate that the proposed methodology effectively determines the image base of firmware that uses LDR instructions to load string addresses.

The algorithms described in this paper are based on little-endian firmware, which is relatively common in industrial control systems. Since the only difference between little-endian and big-endian formats is the byte order, the proposed algorithms are also applicable to big-endian firmware. Note that Unicode strings are not considered in this paper because most existing firmware samples contain sufficient numbers of ANSI strings to determine the correct image bases.

Future research will focus on automatically determining the image base of other types of firmware, such as firmware containing a limited number of strings or firmware whose strings are loaded by ADR instructions. While this research will be challenging, it will reduce the difficulty involved in reverse engineering and conducting detailed security analyses of industrial control system firmware.

Acknowledgement

This research was supported by National Natural Science Foundation of China Grant No. 61370063 and U1636213.

REFERENCES

[1] ARM, ARM Architecture Reference Manual, ARMv7-A and ARMv7-R Edition, ARM DDI 0406C.b (ID07251), Cambridge, United Kingdom, 2012.

[2] Z. Basnight, Firmware Counterfeiting and Modification Attacks on Programmable Logic Controllers, M.S. Thesis, Department of Electrical and Computer Engineering, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, 2013.

[3] Z. Basnight, J. Butts, J. Lopez and T. Dube, Analysis of programmable logic controller firmware for threat assessment and forensic investigation, *Proceedings of the Eighth International Conference on Information Warfare and Security*, pp. 9–15, 2013.

[4] Z. Basnight, J. Butts, J. Lopez and T. Dube, Firmware modification attacks on programmable logic controllers, *International Journal of Critical Infrastructure Protection*, vol. 6(2), pp. 76–84, 2013.

[5] W. Bolton, *Programmable Logic Controllers*, Newnes, Oxford, United Kingdom, 2015.

[6] A. Costin, J. Zaddach, A. Francillon and D. Balzarotti, A large-scale analysis of the security of embedded firmware, *Proceedings of the Twenty-Third USENIX Security Symposium*, pp. 95–110, 2014.

[7] I. Da Costa, N. Mehta, E. Metrock and J. Giffin, Security analysis of an IP phone: Cisco 7960G, *Proceedings of the Principles, Systems and Applications of IP Telecommunications Conference*, pp. 236–255, 2008.

[8] D. Danchev, SCADA systems at the water utilities in Illinois, Houston, hacked, *ZDNet*, November 21, 2011.

[9] Industrial Control Systems – Cyber Emergency Response Team (ICS-CERT), ICS Focused Malware (Update A), Alert (ICS-ALERT-14-176-02A), Idaho Falls, Idaho, July 1, 2014.

[10] R. Langner, Stuxnet: Dissecting a cyberwarfare weapon, *IEEE Security and Privacy*, vol. 9(3), pp. 49–51, 2011.

[11] H. Li, Y. Yu, C. Hu, J. Cao and Y. Hou, Security Report on Industrial Control Systems (in Chinese), Technical Report, NSFOCUS, Beijing, China (⟨www.nsfocus.com.cn/content/details_62_881.html⟩), 2013.

[12] L. McMinn and J. Butts, A firmware verification tool for programmable logic controllers, in *Critical Infrastructure Protection VI*, J. Butts and S. Shenoi (Eds.), Springer, Heidelberg, Germany, pp. 59–69, 2012.

[13] S. Milinkovic and L. Lazic, Industrial PLC security issues, *Proceedings of the Twentieth Telecommunications Forum*, pp. 1536–1539, 2012.

[14] S. Milinkovic and L. Lazic, Some facts about industrial software security, *Proceedings of the Eleventh International Conference on Systems, Automatic Control and Measurements*, pp. 232–235, 2012.

[15] R. Santamarta, Reversing Industrial Firmware for Fun and Backdoors I, Reversemode (⟨www.reversemode.com/index.php?option=com_content&task=view&id=80&Itemid=1⟩), December 12, 2011.

[16] C. Schuett, J. Butts and S. Dunlap, An evaluation of modification attacks on programmable logic controllers, *International Journal of Critical Infrastructure Protection*, vol. 7(1), pp. 61–68, 2014.

[17] I. Skochinsky, Intro to embedded reverse engineering for PC reversers, presented at the *REcon Conference*, 2010.

[18] A. Varghese and A. Bose, Threat modeling of industrial controllers: A firmware security perspective, *Proceedings of the International Conference on Anti-Counterfeiting, Security and Identification*, 2014.

[19] F. Xie, Y. Peng, W. Zhao, Y. Gao and X. Han, Evaluating industrial control device security: Standards, technologies and challenges, *Proceedings of the Thirteenth IFIP TC-8 International Conference on Computer Information Systems and Industrial Management*, pp. 624–635, 2014.

[20] W. Yang and Q. Zhao, Cyber security issues of critical components for industrial control systems, *Proceedings of the IEEE Chinese Guidance, Navigation and Control Conference*, pp. 2698–2703, 2014.

[21] 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.

[22] 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.

[23] R. Zhu, Y. Tan, Q. Zhang, F. Wu, J. Zheng and Y. Xue, Determining image base of firmware files for ARM devices, *IEICE Transactions on Information and Systems*, vol. E99.D(2), pp. 351–359, 2016.