ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/294112252

Determining image base of firmware for ARM devices by matching literal pools

Article *in* Digital Investigation · March 2016

DOI: 10.1016/j.diin.2016.01.002

CITATIONS
15

READS
566

5 authors, including:

Zhu Ruijin
Beijing Institute of Technology
5 PUBLICATIONS **37** CITATIONS

SEE PROFILE

Tan yu-an
Beijing Institute of Technology
85 PUBLICATIONS **454** CITATIONS

SEE PROFILE

All content following this page was uploaded by Zhu Ruijin on 14 March 2018.

The user has requested enhancement of the downloaded file.

Digital Investigation 16 (2016) 19–28

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

Determining image base of firmware for ARM devices by matching literal pools

Ruijin Zhu [a], Yu-an Tan [a], Quanxin Zhang [b], Yuanzhang Li [a], Jun Zheng [a, b, *]

[a] School of Computer Science and Technology, Beijing Institute of Technology, Beijing, 100081, China
[b] Research Center of Massive Language Information Processing and Cloud Computing Application, Beijing, 100081, China

ARTICLE INFO

Article history:
Received 1 August 2015
Received in revised form 19 January 2016
Accepted 21 January 2016
Available online 10 February 2016

Keywords:
Image base
Literal pool
Reverse engineering
Firmware
ARM

ABSTRACT

In the field of reverse engineering, the correct image base of firmware has very important significance for the reverse engineers to understand the firmware by building accurate cross references. Furthermore, patching firmware needs to insert some instructions that references absolute addresses depending on the correct image base. However, for a large number of embedded system firmwares, the format is nonstandard and the image base is unknown. In this paper, we present a two-step method to determine the image base of firmwares for ARM-based devices. First, based on the storage characteristic of string in the firmware files and the encoding feature of literal pools that contain string addresses, we propose an algorithm called FIND-LP to recognize all possible literal pools in firmware. Second, we propose an algorithm called Determining image Base by Matching Literal Pools (DBMLP) to determine the image base. DBMLP can obtain the relationship between absolute addresses of strings and their corresponding offsets in a firmware file, thereby a candidate list for image base value is obtained. If the number of matched literal pools corresponding to a certain candidate image base is far greater than the others, this candidate is considered as the correct image base of the firmware. The experimental result indicates that the proposed method can effectively determine image base for a lot of firmwares that use the literal pools to store the string addresses.

© 2016 Elsevier Ltd. All rights reserved.

Introduction

Embedded devices have become the usual presence in our life, such as cell phones, digital cameras, printers, smart watches and so on. All these devices run special software, often called firmware, which is usually distributed by vendors as firmware updates or firmware images (Costin et al., 2014). Firmware is the soul of embedded devices, because some embedded devices have no other software besides firmware, and the firmware also determines the function and performance of the device.

In the field of digital forensics, we need to reverse analysis firmwares of the embedded devices in some scenarios. For example, (1) By reverse engineering the firmware, back door has been discovered in some devices, such as D-Link (Heffner, 2013a) and Schneider Electric Quantum Ethernet Module (Santamarta, 2011). (2) If data stored on the devices is encrypted, reverse engineering can be applied to obtain the encryption algorithm and even the encryption key which can recover the clear data (Zhang et al., 2015). (3) By reverse engineering the firmwares released by the competing companies, we can determine whether they plagiarize our company's algorithms or infringe our patents and so on. Hence, reverse engineering is an important technology in the digital forensics.

* Corresponding author. School of Computer Science and Technology, Beijing Institute of Technology, Beijing, 100081, China.

E-mail addresses: ruijinzhu@gmail.com (R. Zhu), tan2008@bit.edu.cn (Y.-a. Tan), zhangqx@bit.edu.cn (Q. Zhang), popular@bit.edu.cn (Y. Li), zhengjun_bit@163.com (J. Zheng).

http://dx.doi.org/10.1016/j.diin.2016.01.002
1742-2876/© 2016 Elsevier Ltd. All rights reserved.

Reverse engineering takes a software system as input and uses some technology (such as disassembling and system analysis) to deduce the software source code, design principles, application structures, algorithms, operation processing and related documentation. Reverse engineering not only can avoid duplicating efforts and improve the efficiency and quality of software, but also can translate legacy system into evolution system to efficiently reuse them. Some tools such as Binwalk (Heffner, 2013b), avatar (Zaddach et al., 2014), FRAK (Cui et al., 2013) and BAT (Hemel and Coughlan, 2009) have been designed to modularize the firmware unpacking, modification and repacking processes. They are particularly useful in reverse engineering of firmware. By utilizing reverse engineering, some previously unknown vulnerabilities or security weaknesses have been discovered in banking application (Yoo et al., 2015) and some firmwares of devices, such as SSD (Zhang et al., 2015), printer (Cui et al., 2013) and satellite phone (Driessen et al., 2012).

In reverse engineering area, when disassembling executable file, disassembler needs to know processor type of its runtime environment and image base[1] of executable file (Basnight et al., 2013a). For a given embedded system firmware, we can easily get the processor type[2] but cannot get the image base of firmware. Correctly setting the image base in disassembler during the initial import enhances the analysis of the firmware. More specifically, setting the correct image base ensures that subsequent cross references are accurate where the cross references use absolute addresses rather than offsets in the firmware (Schuett et al., 2014). Cross references include code cross-references and data cross-references. When lack of these cross references information, we are difficult to navigate efficiently in disassembly listing. Facing the obscure disassembly code, people often lost their direction when they look for parts code that they are most interested in. On the other hand, knowledge of the correct image base is critical in understanding the firmware as a whole. Working with an incorrect image base may lead to inaccurate interpretations of segments referenced by absolute addresses (Basnight et al., 2013b).

Related work

As reverse engineering of firmware develops, people have put a great deal of effort into determining the image base of firmware techniques. Skochinsky (2010) proposed a general principle for determining the image base of file with unknown format. They suggested some kinds of hints, such as self-relocating code, initialization code, etc., can be used, but it is not an automatic method and heavily relies on the engineers' experience. Basnight et al. (2013b) presented an overview of the reverse engineering process and proposed a method which can analyze to learn the image base by absolute addresses in the instructions. Heffner (2011) presented a method to infer image base from decompress code in firmware. Utilizing zeroing loops code in BSS section, Heffner (2015) also inferred the image base of firmware file. Peck et al. (2009) scanned the firmware image to look for zlib compressed section in which they found some symbol names. When looking through the firmware, a very regular ten-byte pattern was found in some offset. They speculated these ten-bytes are addresses of symbol names, thereby inferred image base.

Santamarta (2011) mentioned that it needs some tricks to determine image base of firmware, and introduced two methods. The first method is using the “li instructions trick” for MIPS firmware. This method consists of searching the li instructions that load an absolute address into a register. The trick presumes that a significant number of absolute addresses refer to locations in the firmware itself, and therefore have the same base address. Candidate image base are then tested by rebasing the firmware and determining if the absolute addresses correctly align with target data such as functions or strings. Then we can find the image base with trial and error. The second method is to use absolute addresses in jump table to determine image base. The jump table is comprised of absolute addresses of cases, and then the distances between the cases are calculated. If a certain distance is different from others, the corresponding relation between the absolute address of case and offset can be obtained, by which the base address can be determined.

All the above methods require intuition and experience of reverse engineer; in other words, the success and effectiveness often rely on the human factor. And none of them focused on automatic methods that can determine the image base of unknown format files.

Contributions

Binary files with unknown base in reverse engineering mostly come from embedded firmware of which about 63% are based on ARM devices (Costin et al., 2014). Hence, we focus on the image base of firmware on ARM-based devices and propose a new method to automatically determine the image base in this paper. The main contributions of our work are summarized as follows.

(1) Based on the characteristics of literal pool, we propose an algorithm called FIND-LP to recognize all possible literal pools that contain string addresses in firmware. Besides, we can get the string information including string lengths and offsets in the firmware.

(2) We propose an algorithm called Determining image Base by Matching Literal Pools (DBMLP) to determine image base of firmware for the first time. The algorithm utilizes the literal pools recognized by FIND-LP and the string information to calculate some appropriate memory locations which are candidate image base. If

[1] The image base is the base address of the executable file loaded into the memory.

[2] There are usually several ways to get processor type, such as consulting the product manual (Basnight et al., 2013a), physical examination of the device (Basnight et al., 2013b), disassembling the firmware and guessing the processor type (Basnight et al., 2013a).

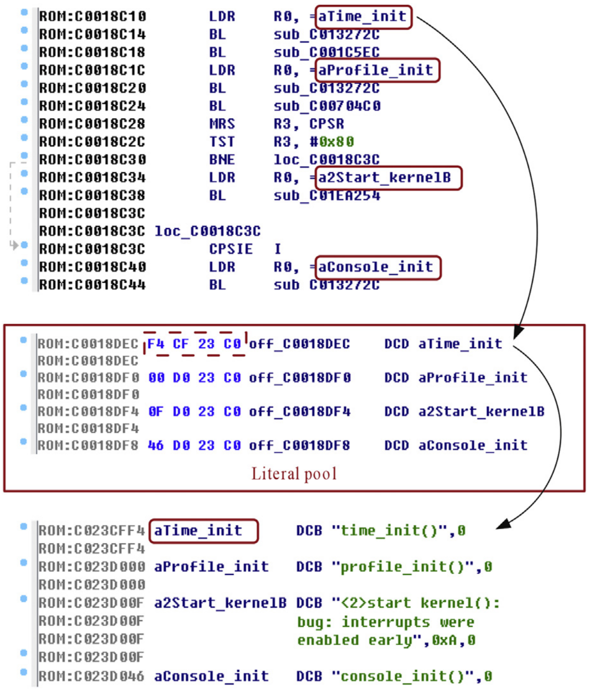


Fig. 1. The literal pool of *vmlinux.bin* (Sony AS30 DV).

the number of literal pools corresponding to a certain candidate image base is far greater than the others, this candidate is considered correct image base.

Roadmap

This paper is organized as follows: In Strings and literal pools in firmware, we introduce the strings and literal pool characteristics in firmware, and present an algorithm to recognize literal pools. In Determining of image base we propose DBMLP algorithm to determine the firmware image base using subvector. The experiments and the analysis of the result are given in Experimental results and analysis. This paper is concluded in Conclusion.

Strings and literal pools in firmware

Strings in firmware

Generally, the strings in firmware consist of some printable characters and escape characters. These strings typically include prompt messages, error messages, version information, copyright information and compiler version, etc. C language is usually used as programming language in ARM embedded system. So we only discuss the C-style strings in this paper. In C language string is typically stored in a character array. The last element of a character array stores string terminator "\0" of which ASCII code is 0x00. According to the above features of strings, we can get the string information including the lengths of strings and the offsets in firmware which will be used in Determining of image base.

In 32-bit operating system, the string address is a 32-bit integer which is beyond the scope of the immediate value of MOV instruction,[3] so the string address cannot be saved into a register with MOV instruction. In fact, it stores the string addresses in literal pool which is an area of constant data in a code section (ARM, 2013), and loads the addresses into registers with the LDR pseudo-instructions. Fig. 1 illustrates the disassembly listing of file *vmlinux.bin* after been disassembled with IDA Pro. The address of string *aTime_init* is 0xC023CFF4 which is stored in a literal pool and is loaded with the LDR pseudo-instruction. The compiler usually stores addresses used in adjacent code within a same area. For example, the address of string *aTime_init* (0xC023CFF4) is stored at memory location 0xC0018DEC, the address of *aProfile_init* (0xC023D000) is stored at 0xC0018DF0, etc. This area that stores addresses is called a literal pool.

Strings shown in Fig. 1 are actually stored in a firmware as shown in Fig. 2(a). There is a string terminator "\0" at the end of each string, followed by the beginning of next string. In 32-bit ARM architecture, the length of a word is 4 bytes. If the length of a string is not an integral multiple of 4, the compiler adds some trailing padding (00 bytes) at the end of the string to make the beginning address of next string being integral multiple of 4. This is called word-aligned, as shown in Fig. 2(b). To improve the performance, some compilers store the strings in word-aligned. In this paper, we discuss both stored modes in order to improve the effectiveness of the algorithm which determines the image base.

Literal pool recognition

The essence of literal pool is an area of code section that is used to store constant data instead of executable code. Literal pool may contain the string address, the function entry address, other types of constant data used in code and so on. As shown in Fig. 1, the addresses of string *aTime_init*, *aProfile_init*, *a2Start_kernelB* and *aConsole_init* are 0xC023CFF4, 0xC023D000, 0xC023D00F and 0xC023D046, respectively. It can be seen that the addresses of these strings are adjacent, because locations where strings are defined in code are close to each other. These addresses are stored in the same literal pool in little-endian order. In order to determine the image base, we need the relationship of absolute addresses and offsets in a binary file. Since literal pool contains absolute addresses of strings, we propose an algorithm FIND-LP to recognize them. Firstly, we define sliding window and the size of sliding window as follows:

[3] MOV instruction only has 12 bits of space to store immediate value, which includes a 4-bit rotate value *roate*4 and an 8-bit *integer*. The formula to calculate immediate value is $immediate = integer\ ROR\ (2^*rotate4)$, so it is not enough for representing an arbitrary 32-bit address value.

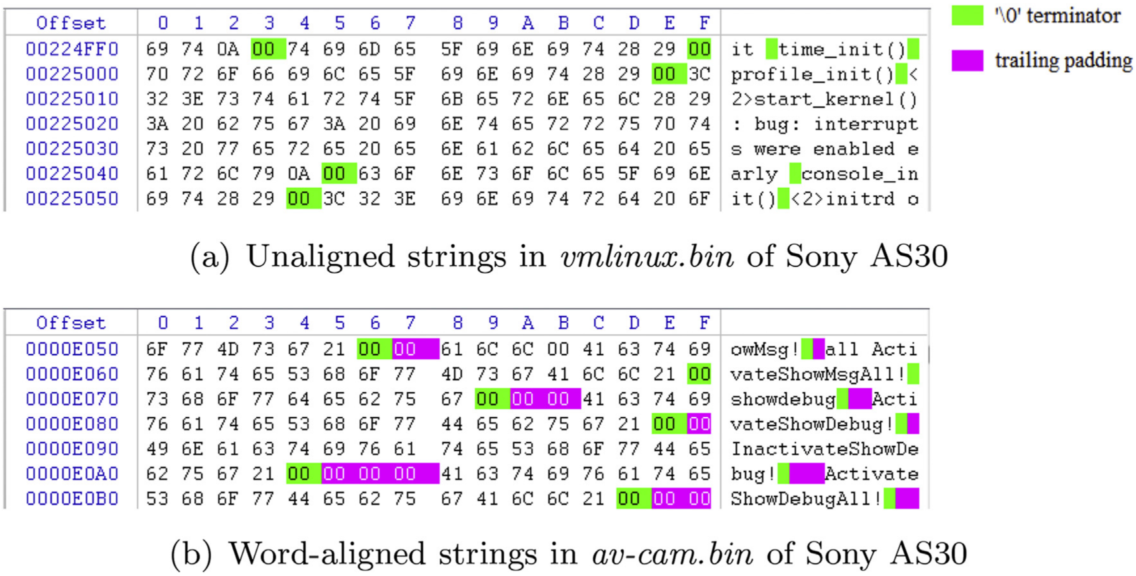


Fig. 2. Two modes of string storage displayed in WinHex.

Definition 1. The sliding window corresponds to a continuous memory sequence, and the number of basic memory units it holds is a constant value.

Definition 2. The size of the sliding window wndsize is defined as the number of string addresses within the window.

Fig. 3 illustrates a recognition process of literal pool. The sliding window has 3 string addresses and each address consumes a word (32 bits). For example, the first 32 bits in the sliding window is the first string address.

According to our experiments, string addresses in the same literal pool usually come from closer code references, and these strings are stored in adjacent position in the firmware, so the distance between the addresses in the same literal pool is mostly less than 64 KB. For example, in Fig. 1 the string *aTime_init*, *aProfile_init*, *a2Start_kernelB and aConsole_init* are referenced in closer code (as shown in the upper portion of the figure) and stored in adjacent position in the firmware (as shown in the bottom portion). On the other hand, the compiler might add some padding bytes in firmware, such as 0x00, 0xFF, etc (Zaddach and Costin, 2013). When searching for literal pool, the algorithm will eliminate these bytes. Accordingly, we define the following rules:

Rule 1. The distance between the string addresses in the sliding window is no more than 64 KB.

Rule 2. The string addresses in the same sliding window cannot be the same address.

According to the characteristics of string in firmware, we can obtain the string offsets and lengths. We analyzed

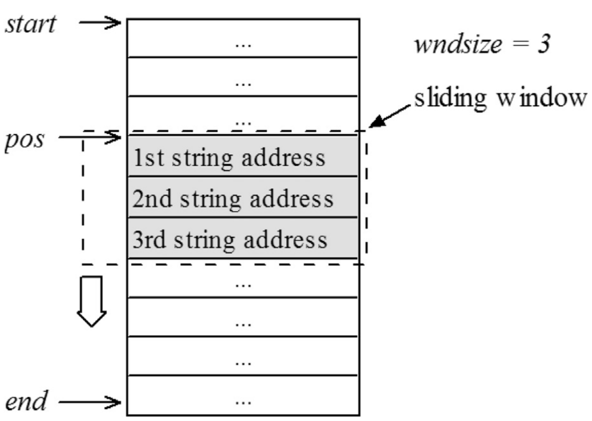


Fig. 3. The recognition process of literal pool.

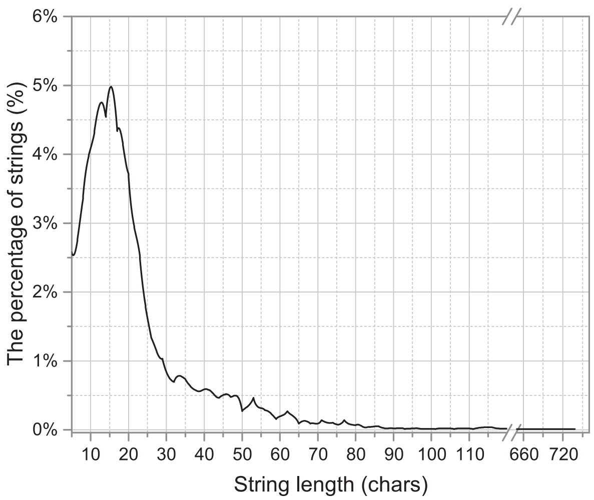


Fig. 4. Statistics about string lengths and corresponding percentage of string.

the strings of all firmwares in the test set (the building of the test set is described in Experimental results and analysis), got all of the string lengths and computed the percentage of each length. The statistical results are shown in Fig. 4. We can see that the lengths of the majority string are less than 100 bytes. By sorting the string addresses in a literal pool and subtracting them sequentially we can get the lengths of strings (if the string is stored in word-aligned, it is the sum of string length and the trailing padding length). For example, the 4 string addresses in literal pool in Fig. 1 can be subtracted in sequence and produce 3 lengths of string. The calculation of the first string length is $0xC023D000 - 0xC023CFF4 = 0xC = (12)_{10}$. From Fig. 1 we can find that the first string is *aTime_init* of which the actual contents is "time_init()\0" and length is exactly 12. With the same calculation methods we can get other string lengths are 15, 55 respectively. Then we have the following rule.

Rule 3. In a literal pool, the difference value between the adjacent string address after sorted is less than 100.

In the algorithm FIND-LP, we first determine whether the addresses in the sliding window satisfy *Rule 1* and *Rule 2*. If they satisfy, the area of current sliding window is considered as a part of possible literal pool. Then the sliding window continues to move forward, until the addresses in the sliding window no longer satisfy the rules. Then we obtain a possible literal pool. Next, we verify whether the possible literal pool satisfies *Rule 3*. If it does not, we consider it is not a literal pool and ignore it. Otherwise, it is considered as a possible literal pool containing string addresses. The algorithm FIND-LP is detailed in Algorithm 1.

Algorithm 1 FIND-LP algorithm

```
function FindLP(start, fileSize, wndsize)
    pos ← start
    end ← start + fileSize
    while start ⩽ pos < end do
        if match(Rule1)&&match(Rule2) then
            head ← pos
            poolSize ← wndsize
            MoveWindow()
            while match(Rule1)&&match(Rule2) do
                poolSize++
                MoveWindow()
            end while
            if match(Rule3) then
                Output : head, poolSize
            end if
            pos += poolSize
        end if
        pos++
    end while
end function
```

Determining of image base

In this section, we will use strings and literal pools obtained in Strings and literal pools in firmware to determine the candidate image base. The compiler usually stores adjacent strings referenced in the code in the same area in reference sequence, and stores the addresses of these strings in the same literal pool in the same sequence. Then the order of string addresses in literal pool is consistent with the order of strings in firmware. According to this characteristic and the idea of subvector, we propose an algorithm called Determining image Base by Matching Literal Pools (DBMLP) to determine the image base of firmware.

Using the features of strings as detailed in Strings in firmware, we can obtain the string information of the firmware, including the offset $(o_1,o_2,\ldots,o_m)$ in the firmware where $o_i<o_{i+1}(1 \le i \le m-1)$ and m is the number of strings in the firmware, and corresponding string lengths $(k_1,k_2,\ldots,k_m)$. $(k_1,k_2,\ldots,k_m)$ can be considered as a m dimensional vector, it is referred to as string length vector henceforth and is denoted as V_k. Similarly, $(o_1,o_2,\ldots,o_m)$ is referred to as offset vector and is denoted as O.

If the strings in firmware are stored in word-aligned, each string length k_i is multiple of 4. Otherwise, if the strings are not word-aligned, we align it by increasing each string length k_i to the closest multiple of 4. Then we get a word-aligned string length vector $(align\_k_1,align\_k_2,\ldots,align\_k_m)$ which is denoted as $align\_V_k$.

By the algorithm FIND-LP we have obtained all the possible literal pools of the firmware. Note that FIND-LP may output some dummy literal pools, because even if part content of a firmware satisfies the 3 rules, it may still not be a real literal pool that contains string addresses. Next, we handle each possible literal pool sequentially. We assume that the literal pools obtained by FIND-LP are correct literal pools, and the values stored in the literal pool are string addresses. The string addresses in literal pool after sorted in ascending order is denoted as $(sa_1,sa_2,\ldots,sa_n)$, where n is the number of string addresses in the literal pool and $n \ll m$.

In general, the strings in a firmware file are stored either word-aligned or unaligned, which is related to the compiler, project configuration and architecture. However, because algorithm FIND-LP may output some dummy literal pools, we cannot judge whether or not strings are stored in word-aligned in accordance with all literal pools obtained by algorithm FIND-LP. Therefore, for each literal pool, we judge whether or not the strings are stored in word-aligned according to the features of addresses. If every string address in a literal pool is an exact multiple of 4, we consider that these strings are stored in the word-aligned; otherwise, they are stored in unaligned. We discuss both cases respectively as follows.

(1) When the strings are stored in unaligned, we can subtract the n string addresses in a literal pool in sequence and get $n-1$ string lengths $(l_1,l_2,\ldots,l_{n-1})$, i.e. $l_j = sa_{j+1} - sa_j (1 \le j \le n-1)$ is the length of string pointed by address sa_j. The set $(l_1,l_2,\ldots,l_{n-1})$ can be considered as a $n-1$ dimensional vector and denoted as V_{lp}. For example, in Fig. 1, there are 4 string addresses in the literal pool and we can get 3 string lengths. Then the vector V_{lp} is $\{12,15,55\}$.

Fig. 5 is a match schematic where V_{lp} has 3 components. The contiguous $n-1$ components of V_k can be referred to as a $n-1$ dimensional sub-vector. V_k has $m-n+2$ such sub-vectors in total and all of these sub-vectors constitute a

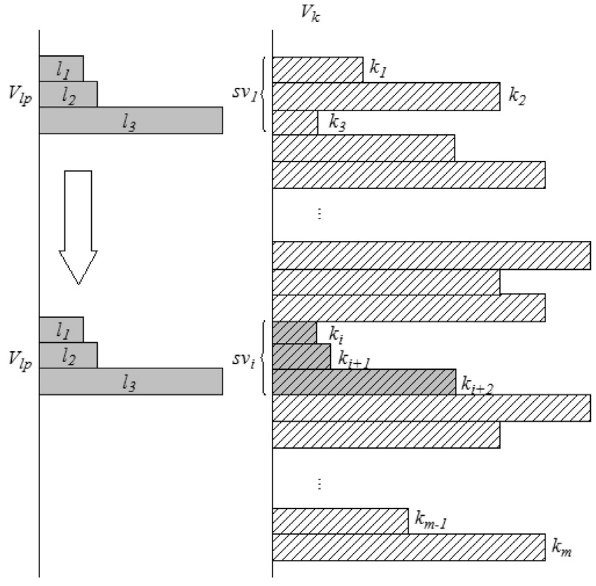


Fig. 5. Matching process.

collection $E = \{(k_1,k_2,\ldots k_{n-1}), (k_2,k_3,\ldots,k_n), \ldots, (k_{m-n+2},k_{m-n+3},\ldots,k_m)\}$. Each element $(k_i,k_{i+1},\ldots,k_{n+i-2})$ in E is denoted by sv_i, where $(1 \le i \le m-n+2)$. Next, we calculate the distance d from V_{lp} to each sub-vector sv_i in collection E, where d is defined as follows:

$$d(V_{lp}, sv_i) = |l_1 - k_i| + |l_2 - k_{i+1}| + \cdots + |l_{n-1} - k_{n+i-2}| \quad (1)$$

If the distance d from V_{lp} to a certain sub-vector sv_i is 0, we consider that it matches successfully. Through the Eq. (1) we can get that the components l_j of V_{lp} respectively equal to the components k_{i+j-1} of sub-vector sv_i. Take the first component l_1 and k_i as an example. Since l_1 is the length of string pointed by sa_1 and k_i is the length of *ith* string in the firmware, we can infer that the *ith* string with offset o_i in the firmware is mapped to the memory location sa_1, as shown in Fig. 6. Therefore, the candidate image base can be determined as $base = sa_1 - o_i$. Next, we continue to calculate the distance d from V_{lp} to the next sub-vector, and calculate the candidate image base in the same way when $d = 0$, until to sv_{m-n+2}. Then we get all of the candidate image base.

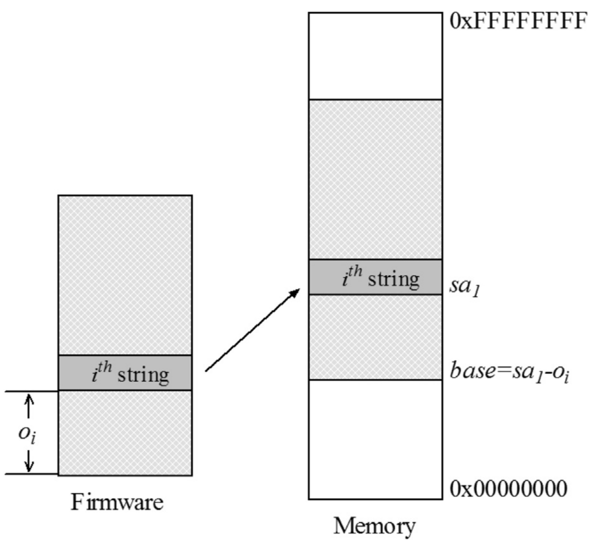


Fig. 6. Map firmware into memory.

Taking the literal pool in Fig. 1 as an example, for $V_{lp} = \{12,15,55\}$ we get $d(V_{lp},sv_{403}) = 0$. It means that the 3 string lengths in sv_{403} is exact {12,15,55}. We can figure out that the offset of the first string in sv_{403} is 0x224FF4. Through Fig. 1 we know that the string address corresponding to the first element of V_{lp} is 0xC023CFF4. Then we can calculate the candidate image base: $base = sa_1 - o_i =$ 0xC023CFF4 − 0x224FF4 = 0xC0018000.

(2) When the strings in the literal pool are stored in word-aligned mode, similarly, $n-1$ string lengths can be obtained by the addresses of strings in a literal pool. However, these lengths are not the actual lengths of the strings, but the sum of the actual length of the string and the length of the trailing padding. In this case, the collection E is defined as $E = \{ (align\_k_1, align\_k_2, \ldots, align\_k_{n-1}) \ldots, (align\_k_{m-n+2}, align\_k_{m-n+3}, \ldots, align\_k_m)\}$ and each element $(align\_k_i, align\_k_{i+1}, \ldots, align\_k_{n+i-2})$ is denoted by $align\_sv_i$, where $1 \le i \le m-n+2$. Then we calculate distance d form V_{lp} to each vector $align\_sv_i$. If the distance between V_{lp} and a certain sub-vector $algin\_sv_i$ is 0, it can be considered that the match is successful. In the same way, the candidate image base is determined as $base = sa_1 - o_i$.

The DBMLP algorithm is detailed in Algorithm 2.

Algorithm 2 DBMLP algorithm

```
P: set of possible literal pools (lp) of firmware
V_k: an m-dimensional vector containing lengths of all strings in firmware
O: an m-dimensional vector containing offsets of all strings in firmware
function DBMLP(P, V_k, O)
    for all k_i ∈ V_k do
        align_k_i ← word_aligned(k_i)
    end for
    for all lp ∈ P do
        sort(lp)
        for all sa_i ∈ lp do
            if sa_i%4 ≠ 0 then
                aligned ← false
                break
            else
                aligned ← true
            end if
        end for
        for all l_j ∈ V_lp do
            l_j ← sa_{j+1} − sa_j
        end for
        for i ← 0, m − n + 2 do
            if aligned == true then
                distance ← d(V_lp, align_sv_i)
            else
                distance ← d(V_lp, sv_i)
            end if
            if distance == 0 then
                base ← sa_1 − o_i
                output : base
            end if
        end for
    end for
end function
```

Experimental results and analysis

Since there is no common test set can be used in our experiments, we collected multiple firmwares from some embedded devices, such as digital video cameras (DV), smart watches, MP3 players, solid-state drives (SSD), satellite phones etc., from the Internet and created a test set to evaluate the validity of our algorithms. The download links of all firmwares are listed in Table 2 of the Appendix. Furthermore, to facilitate the readers to download, we share all the firmware through an online storage system (Zhu, 2016). We implement our algorithms in C language. The experiments are performed in the PC with Pentium Dual-Core 3.0 GHz processor, 4 GB of memory, Microsoft Windows 7 SP1 and Visual C++ 6.0.

Recognition of literal pool

In the following experiment, we firstly identify the strings of firmwares in the test set. Secondly, the FIND-LP algorithm is performed to recognize possible literal pools in the firmware. With a lot of experiments, we find that the number of string addresses in most literal pools is more than 3, so we set the parameter *wndsize* to 3, 4 and 5 respectively in our experiments. The experimental results of *wndsize* = 3 are shown in Table 1 and the impact of *wndsize* on the determination of image base is illustrated in Fig. 7. In Table 1, we can also find that in some firmwares the number of literal pools recognized is 0. The reason will be detailed in Possible reasons for determination failure.

Determination of image base

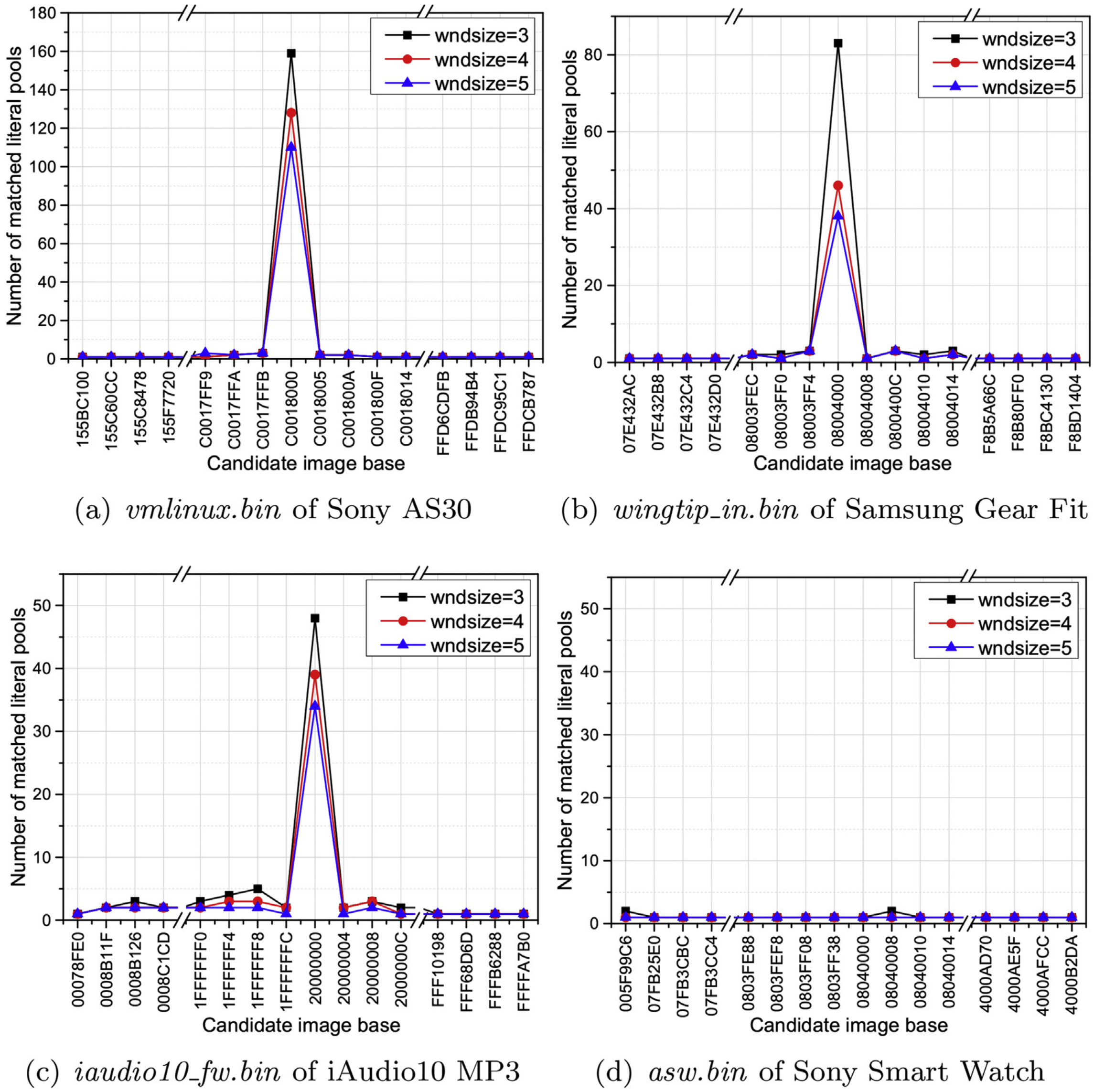
The experimental results of algorithm DBMLP are shown in Table 1. The number of matching literal pools is shown in column *Matched LPs* and the image base of firmware file is shown in column *Image base*. The symbol N/A means that the proposed algorithm is not available for this firmware file, and the reasons will be detailed in next section. Fig. 7(a) shows the experimental results of the firmware file *vmlinux.bin* of Sony AS30 under *wndsize* = 3, *wndsize* = 4 and *wndsize* = 5, respectively. We can find that at the candidate base address 0xC0018000 the number of matched literal pools is respective 159, 128 and 110. This is the location that the number of matched literal pools reaches the maximum and is far larger than other locations of which the number is less than 4. Then the memory location 0xC0018000 is considered the image base. Fig. 7(b) and (c) are the experimental results of firmwares *wingtip_in.bin* of Samsung Gear Fit and *iaudio10_fw.bin* of iAudio10 MP3, and the image base is determined successfully. If there is no location where the matched literal pools is far more than other locations, we consider the determination is failure, as shown in Fig. 7(d).

We can see that the maximum number of matched literal pools reduces with the increase of *wndsize* and its corresponding candidate image base does not change. On the one hand, the literal pools of which string addresses are less than the current *wndsize* are ignored. Hence, with the increase of *wndsize*, there are more and more literal pools being ignored and the maximum number of matched literal pools decreases. On the other hand, since the candidate image base corresponding to the maximum of matched literal pools is the correct image base of the firmware, it does not change with the *wndsize*. That is to say, no matter *wndsize* is set to 3, 4 or 5 (or even larger), we can determine the correct image base with our algorithm. However, in order to get the most obvious experiment result, i.e. the number of matched literal pools at the correct image base is far more than other locations, we usually set the *wndsize* to be 3 in practice.

To verify whether the experimental result is correct, we load *vmlinux.bin* file using IDA Pro, set the processor type to “ARM little-endian” and the image base to 0xC0018000. Then IDA Pro can identify some binary functions, and some of the addresses loaded by the LDR instructions point to strings or functions and display as meaningful string names (data cross-references) or

Table 1
The experimental results (*wndsize* = 3).

Model	File	Strings	Literal pools	Matched LPs	Image base	Validated
Sony AS30 DV	vmlinux.bin	7752	770	159	0xC0018000	Yes
Sony AS30 DV	av-cam.bin	14,135	1764	58	0xC3431000	Yes
Cowon AF2 DVR	loader.af2	1362	176	85	0x67DFFFF0	Yes
Cowon AW1 DVR	update.bin	33,939	1218	274	0xC0007F5C	Yes
Samsung Gear Fit	wingtip_ex.bin	4903	530	19	0x60080000	Yes
Samsung Gear Fit	wingtip_in.bin	6956	294	83	0x08004000	Yes
Pebble Smart Watch	tintin_fw.bin	1580	175	53	0x08010000	Yes
Cowon X9 PMP	cowon_x9_fw.bin	4185	1538	50	0x20000000	Yes
iAudio 10 MP3	iaudio10_fw.bin	3878	1270	48	0x20000000	Yes
iAudio E2 MP3	e2_eu_fw.sb	2394	2059	13	0x00059000	Yes
Sony NEX-7 Camera	vmlinux.bin	5804	596	129	0xC0018000	Yes
Sony NEX-7 Camera	av-cam.bin	16,370	1907	146	0xC3421000	Yes
HTC One	pn07diag.nbh	6196	226	37	0x0ADFFCD8	Yes
HTC Desire 816T	signedbyaa.img	5146	855	8	0x0F4FFFD8	Yes
Thuraya SO-2510	143,005.bin	55,166	10,476	766	0x00800000	Yes
Sony Smart Watch 2	asw.bin	862	158	0	N/A	N/A
Canon EOS 6D Camera	6d000116.fir	8	0	0	N/A	N/A
Samsung 830 SSD	cxm03b1q.enc	1	0	0	N/A	N/A



(a) *vmlinux.bin* of Sony AS30 (b) *wingtip_in.bin* of Samsung Gear Fit

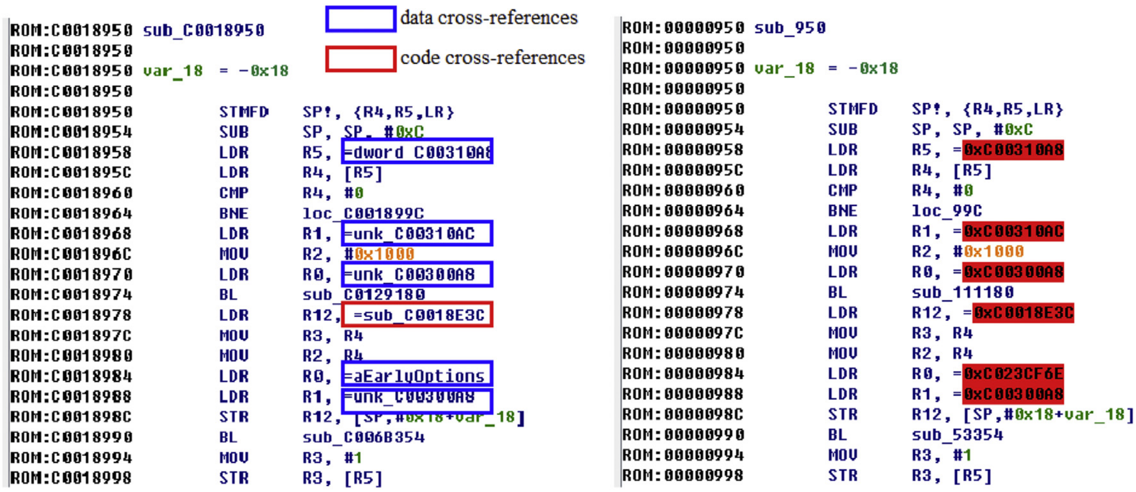
(c) *iaudio10_fw.bin* of iAudio10 MP3 (d) *asw.bin* of Sony Smart Watch

Fig. 7. The image base determination result.

function names (code cross-references), as shown in Fig. 8(a). This means that cross references match nicely, which indicates the image base determined by our algorithm is correct. By comparison, the same file loaded by IDA Pro without setting the proper image base is shown in Fig. 8(b). We can see that LDR instruction does not find the right memory locations and IDA Pro marks these memory locations in red which means these memory locations are not accessed. These red locations introduce many obstacles for reverse engineers to understand the firmware. In this case, IDA Pro cannot build the cross references that use absolute addresses. These cross references have important significance for reverse engineer to understand disassembly listing.

With the same method we verified other files in Table 1 and found all image bases are valid for building correct cross-reference. The validation results are shown in column *Validated* in Table 1.

Through manual analysis, we find that some firmwares has header which may include device name, firmware version, checksum, etc (Basnight et al., 2013a; Peck and Peterson, 2009; Schuett, 2014). In this case, the base address determined by our algorithm is pseudo image base. For example, Cowon AF2 DVR firmware file *loader.af2* contains a 10-byte header. So the base address 0x67DFFFF0 is the pseudo image base. The firmwares containing header in Table 1 also include: *update.bin* of Cowon AW1, *pn07diag.nbh* of HTC One and *signedbyaa.img* of HTC Desire 816T.



(a) Base = 0xC0018000 (b) Base = 0

Fig. 8. The disassembly result of correct image base and incorrect image base.

For the firmwares containing header, although the base address is not the real image base, disassembling the firmware by setting image base to be the pseudo image base and setting the offset to be 0, it still can build the cross references accurately. Besides, our algorithm is also helpful to determine the real image base. Through manual analysis with the pseudo image base, we can easily obtain the header size of the firmware and calculate the real image base finally. For example, the header size of *loader.af2* is 10-byte and the real image base is: 0x67DFFFF0 + 0x10 = 0x67E00000. The real image base of other three firmwares mentioned above can be calculated in the same way. They are respective 0xC0008000, 0x0AE00000, 0x0F500000.

Possible reasons for determination failure

From Table 1, we can see that for some firmwares the number of recognized literal pools is 0, and the image base is not determined successfully (even though it recognizes some literal pools). The possible reasons are analyzed as follows:

(1) Some manufacturer's firmware files are encrypted, such as the *6d000116.fir* of Canon EOS 6D Camera and *cxm03b1q.enc* of Samsung 830 SSD. If we perform the FIND-LP algorithm directly on the encrypted firmware, it will recognize 0 literal pool. This kind of firmwares must be decrypted before using the proposed algorithm to determine image base.

(2) Some of the firmwares load the addresses of strings into a register using ADR instruction rather than LDR. ADR instruction is position independent code (PIC) which adopts relative addressing mode based on PC and does not require absolute address of string (ARM, 2014). So ADR instruction need not literal pool to store absolute addresses. Since the proposed algorithm need the absolute address of string in literal pool, it is invalid for such firmware file, such as the *asw.bin* of Sony Smart Watch 2.

Conclusion

In this paper, we focus on the image base determination method of firmwares under the most common ARM architecture. Firstly, we study the storage mode and characteristics of literal pool in firmware, and propose an algorithm FIND-LP which can recognize literal pools containing the addresses of strings. Secondly, according to the idea of the subvector we propose the DBMLP algorithm to determine the candidate image base with the string information and literal pools. The experimental results and manual verification indicate that the proposed method can effectively determine the image base of firmware which use the literal pools to store the string addresses.

Although we introduce our algorithms taking little-endian firmware as example, they also can be applied to big-endian firmware. In this paper we did not consider the Unicode strings because most of the existing firmwares contain enough ANSI strings with which we can get the correct image base.

For future work, it is still a challenge to automatically determine the image base of other types firmware. We will continue to develop new methods for other kinds of firmwares on ARM-based devices. We believe that these automated approaches can effectively reduce the difficulty of reverse analysis and bring convenience to the reverse engineer.

Acknowledgments

This research was supported by the National High Technology Research and Development Program of China

(Grant No.2013AA01A212), National Natural Science Foundation of China (No.61370063, No.61272511).

Appendix

Table 2
The download links of firmwares in test set.

Model	URL
Sony AS30 DV	http://service.sony.com.cn/DI/Download/63021.htm
Cowon AF2 DVR	http://download.cowon.com/data/C08/AF2/AF2_V2.575_ISP_V3.5.zip
Cowon AW1 DVR	http://dn2.cowon.com/data/C08/AW1/AW1_V3.0.0.zip
Samsung Gear Fit	http://forum.xda-developers.com/showthread.php?t=2719278
Pebble Smart Watch	http://www.pebbledev.org/wiki/Firmware_Updates/
Cowon X9 PMP	http://www.cowonglobal.com/download/Firmware/cowonx9/COWON_X9_2.08.zip
iAudio 10 MP3	http://dn2.cowon.com/data/C08/i10/iAUDIO10_1.10.zip
iAudio E2 MP3	http://www.iaudio.com.cn/manage/UpFile/20121211171624480.zip
Sony NEX-7 Camera	http://di.update.sony.net/NEX/yPM9KL6Nlx/Update_NEX7V103.exe
HTC One	http://www.htc.com/us/support/rom-downloads.html
HTC Desire 816T	http://www.htc.com/us/support/rom-downloads.html
Thuraya SO-2510	http://www.thuraya.com/sites/all/modules/ckeditor/ckfinder/userfiles/files/thuraya-so-2510/20090713_sov68a.zip
Sony Smart Watch 2	http://developer.sonymobile.com/downloads/smart-extras-accessories/smartwatch-firmware-standard/
Canon EOS 6D Camera	http://gdlp01.c-wss.com/gds/2/0400001772/01/eos6d-v116-win.zip
Samsung 830 SSD	http://downloadcenter.samsung.com/content/FM/201310/20131029140712774/Samsung_SSD_830_CXM03B1Q_Mac.iso

References

ARM. ARM compiler armasm user guide. 5th ed. 2013.

ARM. ARM architecture reference manual, armv7-a and armv7-r Edition. 2014.

Basnight Z, Butts J, Lopez J, Dube T. Analysis of programmable logic controller firmware for threat assessment and forensic investigation. In: Proceedings of the 8th international conference on information warfare and Security: ICIW 2013. Academic Conferences Limited; 2013. p. 9–15.

Basnight Z, Butts J, Lopez Jr J, Dube T. Firmware modification attacks on programmable logic controllers. Int J Crit Infrastruct Prot 2013;6(2):76–84. http://dx.doi.org/10.1016/j.ijcip.2013.04.004.

Costin A, Zaddach J, Francillon A, Balzarotti D. A large-scale analysis of the security of embedded firmwares. In: Proceedings of the 23rd USENIX conference on security symposium, USENIX Association, San Diego, CA, United States; 2014. p. 95–110.

Cui A, Costello M, Stolfo SJ. When firmware modifications attack: a case study of embedded exploitation. In: Symposium on network and distributed system security (NDSS), San Diego, CA, United States; 2013.

Driessen B, Hund R, Willems C, Paar C, Holz T. Don't trust satellite phones: a security analysis of two satphone standards. In: IEEE symposium on security and privacy (SP), IEEE; 2012. p. 128–42. http://dx.doi.org/10.1109/SP.2012.18.

Heffner C. Reverse engineering vxworks firmware: Wrt54gv8. 2011. http://www.devttys0.com/?s=wrt54gv8.

Heffner C. Reverse engineering a d-link backdoor. 2013. http://www.devttys0.com/2013/10/reverse-engineering-a-d-link-backdoor/.

Heffner C. Binwalk – firmware analysis tool. 2013. http://binwalk.org/.

Heffner C. Reversing belkin's wps pin algorithm. 2015. http://www.devttys0.com/2015/04/reversing-belkins-wps-pin-algorithm/.

Hemel A, Coughlan S. Binary analysis tool (bat). 2009. http://www.binaryanalysis.org/.

Peck D, Peterson D. Leveraging ethernet card vulnerabilities in field devices. In: SCADA security scientific symposium; 2009. p. 1–19.

Santamarta R. Reverse mode – reversing industrial firmware for fun and backdoors i. 2011. http://www.reversemode.com/index.php?option=com_content&task=view&id=80&Itemid=1.

Schuett CD. Programmable logic controller modification attacks for use in detection analysis. Master's thesis. Air Force Institute of Technology; 2014.

Schuett C, Butts J, Dunlap S. An evaluation of modification attacks on programmable logic controllers. Int J Crit Infrastruct Prot 2014;7(1):61–8. http://dx.doi.org/10.1016/j.ijcip.2014.01.004.

Skochinsky I. Intro to embedded reverse engineering for pc reversers. In: REcon conference, Montreal, Canada; 2010.

Yoo C, Kang B-T, Kim HK. Case study of the vulnerability of otp implemented in internet banking systems of south korea. Multimedia Tools Appl 2015;74(10):3289–303. http://dx.doi.org/10.1007/s11042-014-1888-3.

Zaddach J, Costin A. Embedded devices security and firmware reverse engineering. In: Black Hat, Las Vegas, NV, United States; 2013.

Zaddach J, Bruno L, Francillon A, Balzarotti D. Avatar: a framework to support dynamic security analysis of embedded systems' firmwares. In: Symposium on network and distributed system security (NDSS), San Diego, CA, United States; 2014.

Zhang L, Hao S-g, Zheng J, Tan Y-a, Zhang Q-x, Li Y-z. Descrambling data on solid-state disks by reverse-engineering the firmware. Digit Investig 2015;12(0):77–87. http://dx.doi.org/10.1016/j.diin.2014.12.003.

Zhu R. Firmware test set. 2016. http://pan.baidu.com/s/1nu5kE0p.

View publication stats