Available online at www.sciencedirect.com

ScienceDirect

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

Computers & Security

Determining the base address of MIPS firmware based on absolute address statistics and string reference matching

*Xiaodong Zhu, Yi Zhang, Liehui Jiang, Rui Chang**

State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450001, China

ARTICLE INFO

Article history:
Received 30 January 2018
Revised 29 August 2018
Accepted 25 February 2019
Available online 11 April 2019

Keywords:
Embedded system
Reverse engineering
Base address determination
MIPS architecture
Absolute address statistics
String reference matching

ABSTRACT

Getting the accurate firmware base address is not only the prerequisite for disassembling firmware correctly, but also the basis for reverse analysis. Currently, most existing base address determination methods are applied to the ARM architecture, and a few address determination methods of MIPS firmware highly rely on researcher's manual analysis and experience. To address this problem, based on 32-bit absolute address statistics and string reference matching, an automatic method for the base address determination of MIPS firmware is proposed. Firstly, the 32-bit immediate value loading, addressing mode and string referencing of the MIPS architecture systems are analyzed. Secondly, the Absolute Address Searching (AAS) algorithm and String Reference Matching (SRM) algorithm are proposed based on the analysis. The AAS algorithm utilizes the *lui-ori, lui-lw* and *lui-addiu* instruction pairs to identify and record the absolute addresses loaded to registers. According to the distributions of addresses recorded by AAS, the range of candidate base addresses can be determined. Then the *lui-addiu* instruction pair is used by the SRM algorithm to search for string reference addresses. For every address in the range of candidate base addresses, the SRM algorithm verifies whether each string reference address points to the beginning of a string under the current candidate base address, and thereby the matching rate is calculated. Based on the matching rates corresponding to each of the candidate base addresses, the right base address can be determined. Lastly, the proposed method is applied to the test set composed of 12 mainstream MIPS firmware files. Experimental results demonstrate that the proposed method can determine the base addresses of MIPS firmware files automatically and accurately. Furthermore, the proposed method is also applied to a firmware section after decompression and the result indicates that the proposed method is efficient for automatically determining the loading addresses of firmware sections.

1. Introduction

Nowadays, various kinds of embedded devices, such as cell phones, wireless routers, Ethernet switches, network printers, and so on, are ubiquitous and bring many conveniences to our daily life (Yan et al., 2015). Furthermore, during the past few years, embedded devices became more connected forming what is called the Internet of Things (IoT) (Costin and Zarras, 2016; Chen et al., 2016), and the data flowing through them is of great concern. However, the embedded systems running on these devices acquired a bad security reputation

* Corresponding author.
E-mail addresses: zxd10@tsinghua.org.cn (X. Zhu), crix1021@meac-skl.cn (R. Chang).
https://doi.org/10.1016/j.cose.2019.02.015
0167-4048/

(Costin et al., 2014), largely because the firmware images rely on security by obscurity, rather than security in-depth (Zaddach and Costin, 2013), that is, the security of the firmware relies on the difficulty of reverse analysis instead of the security mechanisms or security software. Reverse engineering is the process of developing a set of specifications for a complex hardware system by an orderly examination of specimens of that system (Rekoff, 1985). It can be used to find backdoors, vulnerabilities and security weaknesses in firmware files, thus helping us to improve the security of the firmware in time and protecting the firmware against malicious attacks. Therefore, reverse analyzing the firmware of embedded devices is of great significance.

Firmware disassembling is the basis of reverse analysis. When a disassembler starts to disassemble a firmware file, the very first and crucial thing is to obtain the processor type and the base address (Basnight et al., 2013b). Generally, base address is the calculating datum for relative offsets. It refers specifically to the calculating datum for the offsets in firmware, which is actually the address in memory where the first bit of the firmware is loaded. Utilizing some disassembling tools, such as binwalk (Heffner, 2013) and BAT (Hemel and Coughlan, 2009), it is easy to know the processor type, and the basis of disassembling analysis can be set. However, even if the proper firmware processor type is known, it is still impossible to analyze the target firmware without the correct base address, and all we can obtain is some obscure disassembly codes. Only if the correct base address is set, the disassembler can build correct cross-references and generate legible assembly codes. Cross-reference includes code cross-reference (e.g. function call) and data cross-reference (e.g. string reference), which can help us to understand the overall structure of firmware and negotiate code analysis. Conversely, working with an incorrect base address may lead to inaccurate interpretations of segments referenced by absolute addresses (Basnight et al., 2013a). Because the cross-references usually use absolute addresses (Schuett et al., 2013), if a wrong base address is set, some references may point to a location that is beyond the address range of the firmware thus cause errors in disassembling.

However, acquiring the base address has always been a complicated work. It can be acquired by analyzing the boot loader of the firmware, but binary code analysis needs a lot of energy and time and the boot loader is sometimes unavailable. For base address determination in the situation where boot loader is unavailable, researchers have put a great deal of effort and many excellent results have been achieved. Skochinsky (2010) suggested some kinds of hints, such as the self-relocating code, initialization code, jump tables, string tables, for determining the base address of a file with unknown format, which has great implications for base address determination. Then, Heffner (2011) proposed a method to infer the base address of the firmware using the decompress codes. The decompress code is used to decompress the compressed sections to the correct locations in memory before they are executed. By analyzing these decompress codes, the approximate range of the base addresses can be inferred. Basnight et al. (2013a) presented an overview of the reverse engineering process and proposed a method which could manually reason out the base address by immediate values in the instruction. All the above methods can determine the base address of the firmware, regardless of the architecture of the processor, but they cannot give out the base address automatically and heavily rely on the experience of reverse engineers. Therefore, people also need to spend plenty of time and energy in determining the base addresses of the firmware. In automatic determination, Zhu et al. (2016a) successively proposed two base address determining algorithms for the firmware of ARM devices based on the characteristics of function entry tables and the literal pools matching (Zhu et al., 2016b) in 2016. Additionally, they proposed another base address determining methodology for ARM-base industrial control system firmware in 2017 (Zhu et al., 2017). These methods respectively uses function entry table (FET) and LDR pseudo-instruction of ARM architecture and can automatically detect the base addresses of the ARM firmware files. However, there are no FETs and LDR pseudo-instructions in MIPS architecture thus these methods cannot be used to determine the base addresses of the firmware of MIPS devices, and to the best knowledge of the authors, the automatic base address determination method has not been given out for MIPS firmware.

To address this problem, an automatic base address determination method of MIPS firmware based on absolute address statistics and string reference matching is proposed in this manuscript. Utilizing the characteristics when the absolute addresses are used in a MIPS firmware, the Absolute Address Searching (AAS) algorithm is proposed firstly, in order to search for the absolute addresses loaded to the register by matching instruction pairs *lui-ori, lui-lw* and *lui-addiu*. Based on the found absolute addresses, the range of candidate base addresses can be determined. After that, depending on the characteristic when a string is referenced in MIPS firmware, the String Reference Matching (SRM) algorithm is proposed to determine the base address. By matching instruction pair *lui-addiu*, all the string references can be searched out and the string reference addresses can be recorded. Then for every address in the candidate base address range, the number of string reference addresses that exactly point to the beginning of a string is counted and used to calculate the matching rate. Lastly, the correct base address can be decided according to the matching results.

Research contributions of this manuscript can be summarized as follows:

- An automatic method composed of AAS algorithm and SRM algorithm is proposed. It can automatically determine the correct base address of MIPS firmware by absolute address searching and string reference matching, which is helpful for the disassembling in the reverse engineering.
- The proposed method can also be flexibly used to determine the loading address of firmware sections such as code after being decompressed or moved, and help the memory layout analysis of the firmware.

The rest of this manuscript is organized as follows. Some preliminaries are presented in Section 2, including an overview of MIPS, the instruction formats of MIPS32 architecture, and three typical situations involving 32-bit absolute addresses. The method composed of Absolute Address Searching (AAS) algorithm, candidate range determination

and String Reference Matching (SRM) algorithm are proposed in Section 3. In Section 4, the proposed method is applied to the test set and the experimental results are presented. In Section 5, the experimental results are discussed and another application scenario of the proposed method is described. Lastly, Section 6 summarizes the whole manuscript and a possible future is pointed out.

2. Preliminaries

In order to make the proposed method more comprehensible and present some foundations of this method, this section mainly introduces preliminary works in the following three aspects: firstly, an overview of MIPS architecture is presented; secondly, the formats of MIPS32 instructions are depicted and four main instructions involved are emphatically described; lastly, three typical situations involving 32-bit absolute addresses are depicted and this reveals the principle of the proposed method.

2.1. MIPS overview

MIPS is a popular RISC processor in the world, which means Microprocessor without Interlocked Piped Stages. The mechanism is to use software as far as possible to avoid data related issues in the pipeline (Sweetman, 2010). It was firstly developed by the research team led by Professor Hennessy of Stanford University in the early 1980s. The system structure and design concept in MIPS are fairly advanced. The instruction system has developed from MIPS I, MIPS II, MIPS III, MIPS IV to MIPS V, and the embedded instruction system has developed from MIPS16, MIPS32 to MIPS64, which has been very mature.

The MIPS32 architecture refreshes the performance standard for 32-bit embedded processors. It lays a steady foundation for the next-generation high-performance MIPS-Based processor SoC in MIPS Technologies, and it is upwardly compatible with MIPS 64-bit architecture. The MIPS32 architecture is an expansion of the previous MIPS I and MIPS II Instruction Set Architecture (ISA), which incorporates powerful new instructions specifically for embedded applications, as well as the proven memory management and privilege mode control mechanisms, which can only be seen in the 64-bit R4000 and R5000 MIPS processors.

The MIPS32 architecture is based on a fixed-length, periodic coding instruction set, and it uses the load/store data model. The improved architecture can support optimized execution of high-level languages. Its arithmetic and logic operations take the form of three operands, allowing the compiler to optimize complex expressions. In addition, it also has 32 general registers, which allow the compiler to further optimize code generation performance by maintaining frequent loading and accessing processes to data within the registers.

2.2. Instruction formats of MIPS32

In MIPS32, each CPU instruction consists of a single 32-bit word, aligned on a word boundary. This makes it practicable for us to easily partition the boundary of instructions and search for specific instructions. There are three instruction formats, including immediate (I-type), jump (J-type), and register (R-type), as shown in Fig. 1. The use of a small number of instruction formats simplifies instruction decoding, allowing the compiler to synthesize more complicated (and less frequently used) operations and addressing modes from these three formats as needed (Sweetman, 2010).

In I-Type instructions, there are four instructions that is usually used to load immediate values to the register, including *lui, ori, lw* and *addiu*, which could be useful for the base address determination. For instruction *lui*, which loads the highest 16 bits of a 32-bit immediate value to a register, its format is shown in Fig. 2(a). Because the data is moved from the memory, 5 bits of region *rs* is all 0, and region *rt* is the serial number of the target register. Region *immediate* is a 16-bit unsigned immediate value. Similarly, the instruction formats of *ori, lw* and *addiu* is respectively shown in Fig. 2(b), (c) and (d). In particular, the *immediate* region of *lw* or *addiu* is a 16-bit signed immediate value.

Knowing the instruction formats of these instructions, these instructions can be easily searched by their opcode segments, and their source registers, target registers and immediate values can also be extracted.

2.3. Three typical situations involving 32-bit absolute addresses

There are three typical situations involving 32-bit absolute addresses in MIPS firmware, including 32-bit immediate value loading, word-loading addressing and string referencing.

For 32-bit processors, loading absolute addresses means loading 32-bit values to the registers. However, the length of both ARM instructions and MIPS instructions is 32 bits. In other words, a 32-bit value cannot be loaded to the register by only one instruction, because in this operation, the operand occupies all the 32 bits and there is no space left for the opcode. In order to solve this problem, *LDR* instruction is used in ARM to load a 32-bit value to the register: *LDR cond* $<Rd>$, $<$addressing mode$>$ (Sloss et al., 2004). *LDR* can load a value from the memory to register Rd. When loading a 32-bit immediate value, this value needs to be stored in memory first, then *LDR* instruction will be called to load this value from memory into the register (Blem et al., 2013).

In contrast, MIPS takes a different approach. It uses pseudo-instruction *li* to load 32-bit immediate values to registers. When *li* is used for loading a 32-bit immediate value, it is implemented by an instruction pair of *lui* and *ori*. The instruction *lui* can load a 16-bit value to the higher 16 bits of a register, while the lower 16 bits are set to 0. Therefore, the process of loading a 32-bit immediate value to a register using *li* consists of two steps: firstly, *lui* instruction is used to load the higher 16 bits of this immediate value to the higher 16 bits of the target register and set its lower 16 bits to 0; secondly, *ori* instruction is called to implement the bitwise OR operation between the lower 16 bits of the target register and the lower 16 bits of the immediate value. Because the lower 16 bits of the register are all 0, the lower 16 bits immediate values are loaded to the register. Therefore, all the 32 bits of the immediate value are loaded to the target register.

Besides, under MIPS architecture, the only addressing mode directly supported by load/store instructions is "base

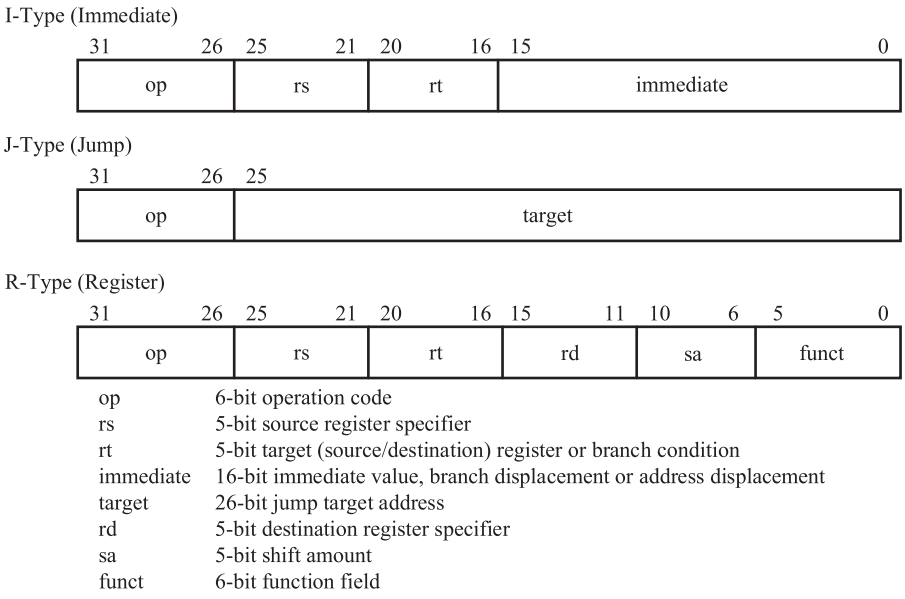


Fig. 1 – Instruction formats of MIPS32 (Sweetman, 2010).

address register + offset of 16-bit signed immediate". Similar to the 32-bit immediate value loading, the process also consists of two steps: firstly, *lui* is used to load the 16-bit base address to the base address register; secondly, *lw* is used to add the 16-bit signed offset.

The third situation involving absolute addresses is about string references. MIPS uses pseudo-instruction *la* to reference a string, and its operand is the address of the string. Similar to the former two situations, pseudo-instruction *la* is implemented by instruction pair *lui-addiu* and the process also consists of two steps.

As we can see, all the above situations can be identified by instruction pair *lui-ori*, *lui-lw* and *lui-addiu* respectively. Therefore, the absolute addresses involved can be recorded by searching for these character instruction pairs and extracting their operands.

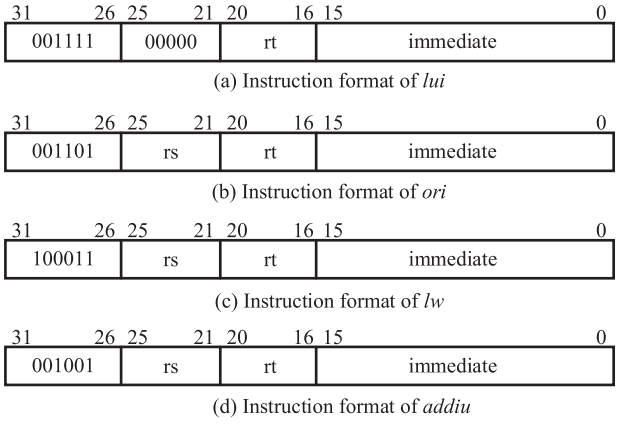


Fig. 2 – Instruction formats of four specific instructions.

3. Proposed method

Based on the above analysis, the AAS algorithm and SRM algorithm are proposed in this section, and the process of determining the range of candidate base addresses is also given out, thus the base address of MIPS firmware can be determined.

3.1. *Statistics of absolute addresses based on instruction pairs searching*

In this section, the Absolute Address Searching (AAS) algorithm is proposed to get the statistics of absolute addresses loaded to the registers.

As described in Section 2, three typical situations involving absolute addresses can be identified by instruction pairs *lui-ori*, *lui-lw* and *lui-addiu*. Depending on the instruction formats, it is easily to search for every instruction and get its source register, target register, and immediate value. Therefore, it is practicable to search for these instruction pairs and record the absolute addresses involved.

However, in the analysis of MIPS firmware, when searching for typical instruction pairs, there are two situations that may lead to inaccurate results.

Situation 1: When *li* (*lui* and *ori* in fact) is used to load 32-bit immediate values to registers, *lui-lw* pair is used for addressing, or *la* (*lui* and *addiu* in fact) is used to reference a string, there might be several other instructions between *lui* and *ori*, *lw*, or *addiu*. In this case, only judging adjacent instructions might lead to identification omissions. In Fig. 3, the *li* instruction (at ROM:81F80448) is actually an *ori* instruction paired to the *lui* instruction (at ROM:81F80440). To facilitate reading, the MIPS disassembler displays it directly as *li*. And we can see there is an *sll* instruction (at ROM:81F80444) between the *lui* and *ori*. In fact, this situation is very common in MIPS firmware.

In this situation, although there might be several instructions between the instruction pairs, the interval size is always

```
ROM:81F80440    lui     $v0, 0x81FA
ROM:81F80444    sll     $a0, 2
ROM:81F80448    li      $v0, 0x81F989E0
```

Fig. 3 – An isolated instruction pair *lui*.

```
ROM:81F80DE4    lui     $v0, 0x81F9
ROM:81F80DE8    lw      $a0, 0x10($k0)
```

Fig. 4 – Two typical instructions have different operation registers.

an integer multiple of 4 bytes. To avoid these omissions, sliding window and its size *wndsize* are defined as follows:

Definition 1. The sliding window corresponds to a contiguous file content, and the number of bits it holds is a constant.

Definition 2. The size of sliding window *wndsize* is defined as the number of instructions (i.e. the multiples of 4 bytes) in sliding window.

Therefore, the specific instructions existing in the scope of sliding window are considered as typical instruction pairs no matter there is interval or not between them.

Situation 2: In every typical instruction pair, the target register of instruction *lui* must be the source register of instruction *ori, lw* or *addiu*. But there is such a situation that the target register of *lui* is not the source register of *ori, lw* or *addiu*, even though two typical instructions are adjacent to each other, as shown Fig. 4. Because the target register of *lui* is $v0, while the source register of *lw* is $k0, this is not a typical instruction pair, and the address is not 0x81F90010. Therefore, if only adjacency is judged without register comparing, identification mistakes might take place.

Aiming at this situation, after two typical instructions are found, the target register of *lui* and the source register of *ori, lw* or *addiu* are compared. If these two registers are the same register, these two instructions are considered as a typical instruction pair; otherwise, they will be excluded.

Taking these two situations into account, the process of absolute address searching (*wndsize*=4) is shown in Fig. 5. The red rectangle represents the sliding window. When searching for the entire file, the sliding window constantly slides, and every sliding distance is the length of an instruction (4 bytes). Each time the sliding window slides to a location, judge whether the first instruction in the window is *lui* (i.e. judge whether the first 6 bits of the instruction machine code is the operation code of *lui* instruction 001111). If not (as shown in Fig. 5(a)), slide the sliding window forward to the position of the next instruction (as shown in Fig. 5(b)). If the first instruction is *lui* (as shown in Fig. 5(b)), record its target register *rt* and successively judge every instruction from the instruction next to *lui* to the last instruction in the sliding window whether it is *ori, lw* or *addiu*. If any of the three target instructions is found, judge whether its source register *rs* is same as the target register of *lui*. If they are the same register, these two instructions are considered as an instruction pair. Then record the 16-bit operand of *lui* (such as 0xABCD in Fig. 5(b)), and represented as *H*, as well as that of *ori, lw* or *addiu* (such as 0xEFGH in Fig. 5(b)), and represented as *L*. If the instruction pair is *lui-ori*, since the immediate region of *ori* is a 16-bit unsigned value, the absolute address *Addr* can be calculated as the following formula:

$$Addr = H \times 2^{16} + L. \tag{1}$$

But if the instruction pair is *lui-lw* or *lui-addiu*, different from *ori*, the *immediate* region of *lw* or *addiu* is a 16-bit signed value. Thus the 15*th* bit of *L* is the sign bit, which is represented as L_{15}. If L_{15}=1, it means that *L* is a negative number, and *H* is the result after being compensated by adding 1 to the original higher 16 bits of the 32-bit address. Therefore, to calculate the original 32-bit address, we need to subtract 1 from *H*. The absolute address *Addr* can be calculated as the following formula:

$$Addr = \begin{cases} H \times 2^{16} + L, & if \quad L_{15} = 0 \\ (H-1) \times 2^{16} + L, & if \quad L_{15} = 1 \end{cases}, \tag{2}$$

where L_{15} means the 15th bit of *L*.

Then, save the value of *Addr* and continue the searching. If another *lui* instruction is found in the sliding window, slide the window to this location and begin to search *ori, lw* or *addiu* from the next instruction. If the entire window has been searched, slide the window to a distance of *wndsize* (to the location shown in Fig. 5(c)). Continue this process until to the end of the file.

The details of this algorithm are shown in Algorithm 1 .

Given the above, the AAS algorithm uses specific instruction pairs to search for and record absolute addresses, and utilizing these addresses, the range of candidate base addresses can be determined, which builds the foundation of base address determining.

3.2. *Determination of the range of candidate base addresses*

In this section, the range of MIPS firmware candidate base address based on the result of AAS algorithm is determined at first. Then the String Reference Matching (SRM) algorithm is proposed to search for the string reference addresses and calculate the matching rates corresponding to different candidate base addresses. At last, the correct base address is decided according to the matching results.

Since the firmware code is self-contained, it can be assumed that all the absolute addresses loaded or accessed in the firmware must be within the address range where the

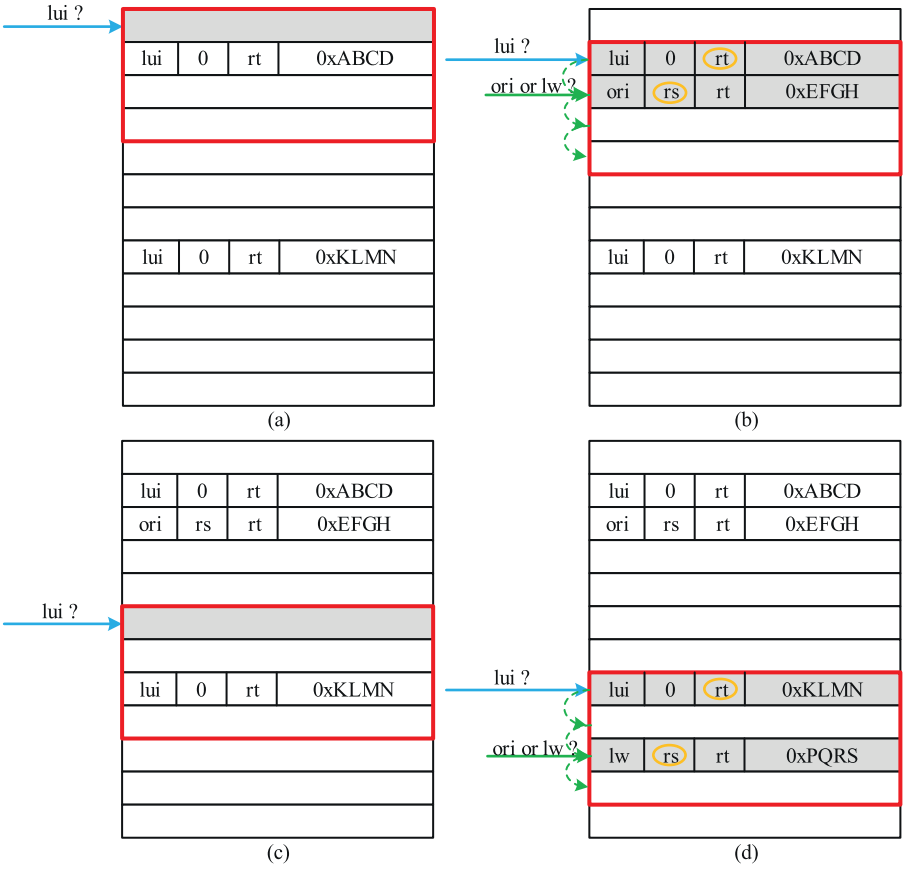


Fig. 5 – Process of absolute address searching (*wndsize*=4).

firmware is loaded into memory. Inspired by the ideas in Zhu et al. (2016a), an algorithm to determine the range of candidate base addresses is proposed based on the recorded absolute addresses in *Addr*.

After the file search process above, a series of absolute addresses in *Addr* can be obtained. These absolute addresses are unordered and may have some duplicates. After removing all the duplicates, sorting them in ascending order, and storing the results in array *location*. Assume that there are *n* addresses in *location*. Then *location*[0] is the lowest absolute address which have been loaded or accessed, and *location*[*n*-1] is the highest.

If the binary firmware file is loaded into memory, it must include *location*[*n*-1]. Assume that *location*[*n*-1] is the highest address loaded in memory, and *fileSize* is the size of the firmware file, then *location*[*n*-1]-*fileSize* is calculated to be the starting address of the binary firmware file in memory, that is, the lowest possible location of base address, as seen in Fig. 6(a). Since *location*[0] is the lowest address loaded or accessed, the address where the file is loaded cannot be higher than *location*[0]. Thus *location*[0] is the highest possible location of base address, as shown in Fig. 6(b). From the above analysis, it can be concluded that the candidate range is from (*location*[*n*-1]-*fileSize*) to *location*[0], as shown in Fig. 6(c).

In theory, the correct base address must be one of the addresses in this range. However, in practice, we find that this method is invalid, for that *location*[0] is usually smaller than *location*[*n*-1]-*fileSize* and the theoretical range does not exist.

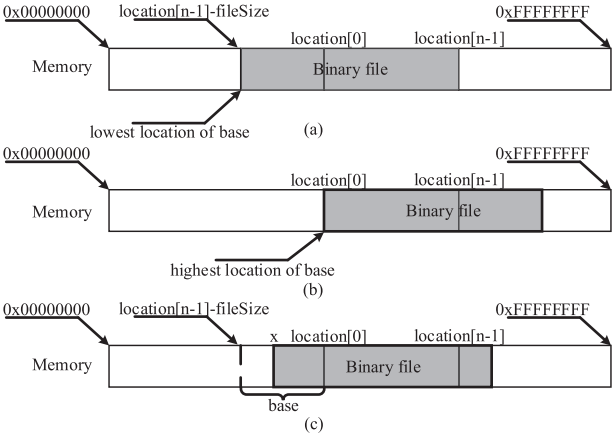
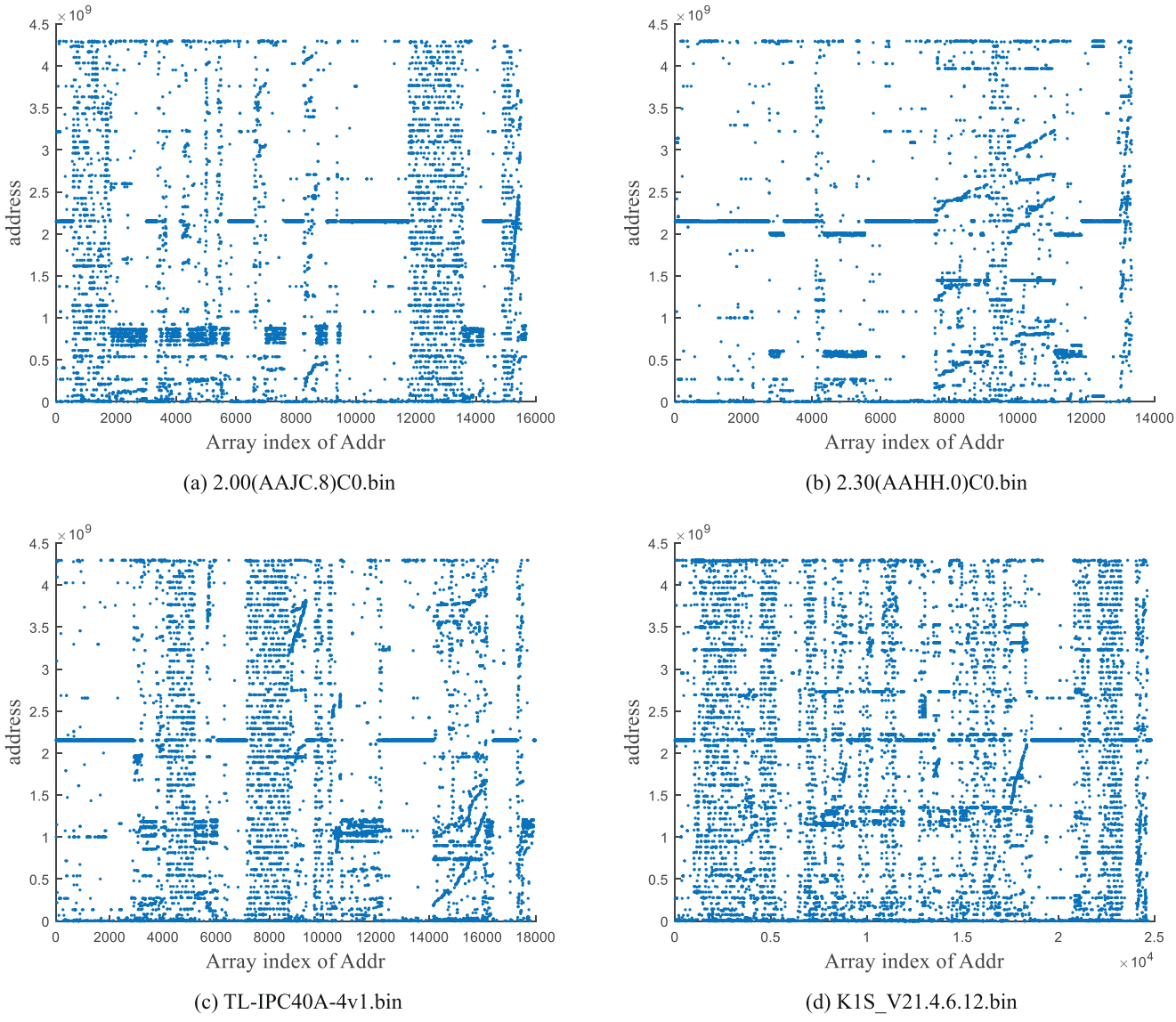


Fig. 6 – Process of determining the range of candidate base addresses (Zhu et al., 2016a).

The reason is that some immediate values loaded by *lui-ori* are not absolute addresses. These values in *Addr* will not be loaded in the range of binary file shown in Fig. 6 (c). Therefore, another effective way is needed to determine the range of candidates.

The immediate values stored in *Addr* of 4 common firmware files are shown respectively in Fig. 7(a)～(d). The *x* coordinates are the indexes of array *Addr*, while the *y* coordinates are the immediate values stored in *Addr*. From Fig. 7 it



(a) 2.00(AAJC.8)C0.bin

(b) 2.30(AAHH.0)C0.bin

(c) TL-IPC40A-4v1.bin

(d) K1S_V21.4.6.12.bin

Fig. 7 – Values in *Addr* of 4 common firmware files.

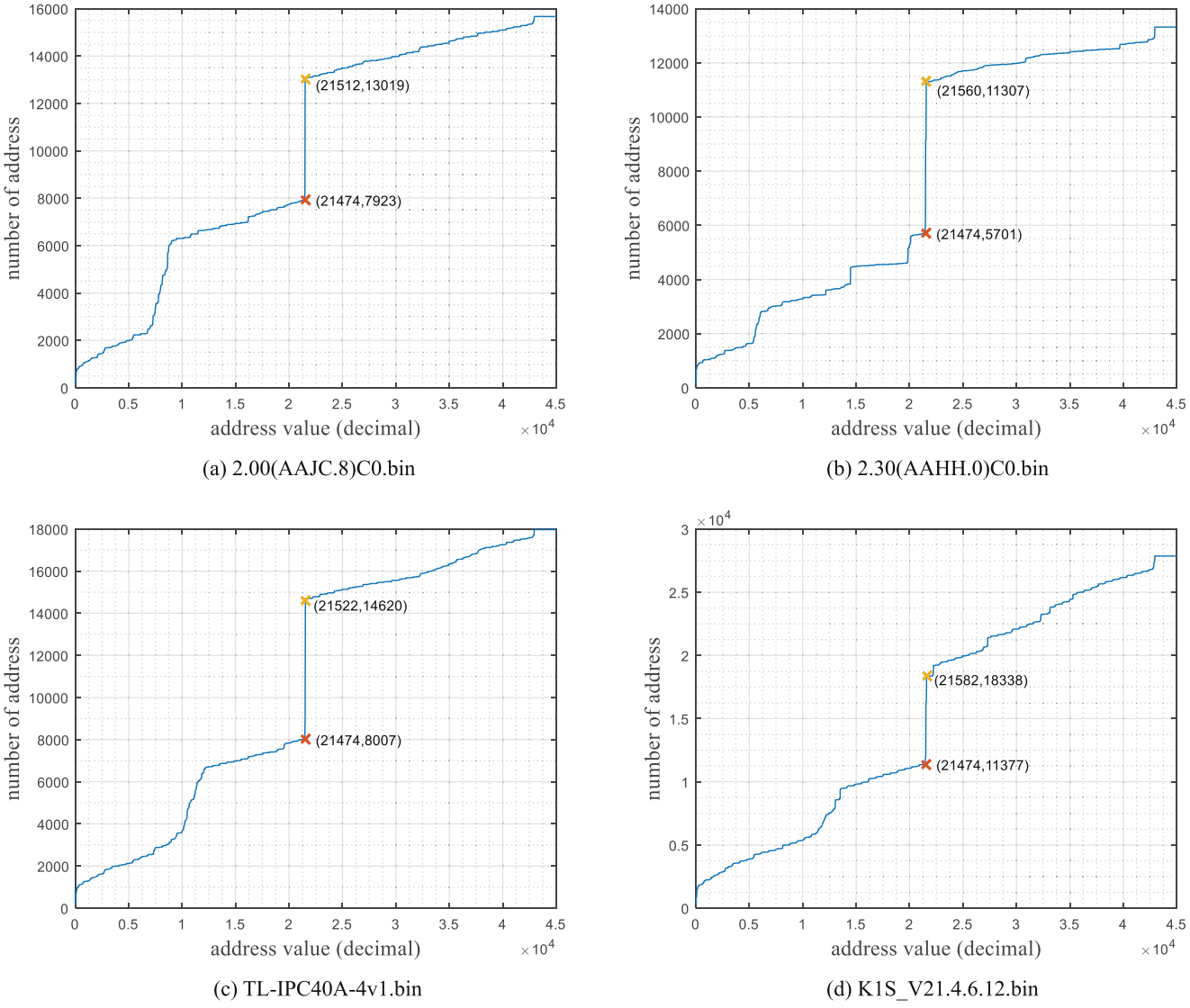
can be seen that in the neighborhood of some specific y values, these points present well-regulated distribution, such as the intermittent horizontal line between 2×10^9 and 2.5×10^9.

In order to visualize the number of recorded immediate values in different address interval, the address space is divided into 4.5×10^4 sections, and every section contains 10^5 addresses. The highest address is 4.5×10^9 (0x1,0C38,8D00), which means that it has covered the entire address space. Take the maximum value of each section as the *x* coordinate, and the number of the values that is smaller than the *x* coordinate as the *y* coordinate, and the accumulative curves of immediate values in *Addr* can be drawn. Fig. 8(a) ~ (d) respectively shows the accumulative curves of the firmware files shown in Fig. 7(a) ~ (d).

It can be seen that there always exists a narrow range in which the number of addresses grows rapidly, such as range 2,147,400,000 ~ 2,151,200,000 in Fig. 8(a), range 2,147,400,000 ~ 2,156,000,000 in Fig. 8(b), range 2,147,400,000 ~ 2,152,200,000 in Fig. 8(c) and range 2,147,400,000 ~ 2,158,200,000 in Fig. 8(d). It means that a large number of addresses are in this range, that is, addresses of this range are accessed frequently in firmware. As mentioned above, all the absolute addresses accessed in firmware must be within the address range where the firmware is loaded into memory. Thus it can be inferred that the base address of the firmware is in this address range. This inference can be proved by the size of the range. The sizes of the range in Fig. 8(a) ~ (d) are respectively 3,800,000 (3710 K), 8,600,000 (8398 K), 4,800,000 (4687 K) and 10,800,000 (10546 K), and the size of the corresponding firmware files are 3103 K, 7569 K, 4391 K and 10138 K (Byte). The size of the address range approximately equals to the size of firmware file.

Therefore, an inductive method of determining the range of candidate base addresses has come out:

1. the range of the recorded absolute addresses is determined.



(a) 2.00(AAJC.8)C0.bin

(b) 2.30(AAHH.0)C0.bin

(c) TL-IPC40A-4v1.bin

(d) K1S_V21.4.6.12.bin

Fig. 8 – Accumulative curves of 32-bit immediate values in *Addr*.

2. the distribution curve of the recorded absolute addresses is drawn.
3. the range of candidate base addresses is determined by reading the *x* coordinates of the beginning and ending of the sharp rise in the distribution curve.

3.3. *Determination of the base address based on string reference matching*

The next work is to find out the correct base address from the candidate range. As described in Section 2.3, string referencing in MIPS is implemented by loading a 32-bit address of target string to the register using *la*. Therefore, under the correct base address, every address loaded by *la* must point to the beginning of a string. Based on this characteristic, for every address in the range of candidates, the correct base address can be determined by string reference matching.

Firstly, use instruction pair *lui-addiu* to identify string references and record the 32-bit string reference addresses to *srAddr*. Because these addresses are locations in memory when the firmware is running, the offsets in the firmware can be calculated by the following formula:

$$Offset = srAddr - base', \tag{3}$$

where *srAddr* denotes the recorded string reference addresses, while $base'$ denotes the candidate base addresses.

Then judge whether *srAddr* points to the beginning of a string. 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. Besides, strings are actually stored in a firmware and there is a string terminator " \0 " (ASCII code "00") at the end of each string, followed by the beginning of next string. Therefore, *srAddr* is considered as pointing to the beginning of a string only if the *Offset* corresponding to it satisfies the following three conditions (the smallest length of a string is considered as 2 chars):

Algorithm 1: The Absolute Address Searching (AAS) Algorithm.

```
Input: The target firmware file, wndsize
Output: count, Addr[]
start = the beginning of the firmware file;
end = the ending of the firmware file;
pos = start, count = 0;
while start ≤ pos < end - wndsize do
    if the instruction pos points to is lui then
        regnum = the rt region of lui instruction;
        h = the immediate region of lui instruction;
        for search = pos; search ≤ pos + wndsize; search ++ do
            if the instruction search points to is ori, lw or
            addiu && its rs region == regnum then
                count += 1;
                l = the immediate region of ori, lw of addiu
                instruction;
                if the instruction search points to is ori then
                    use formula (1) to calculate Addr[count];
                end
                else
                    use formula (2) to calculate Addr[count];
                end
            end
            if the instruction search points to is lui then
                pos = search;
                Break;
            end
        end
    end
    else
        pos ++;
    end
end
return count, Addr[]
```

1. The ASCII code of Byte(*Offset*) $\in$ [0x20, 0x7E];
2. The ASCII code of Byte(*Offset* +1) $\in$ [0x20, 0x7E];
3. The ASCII code of Byte(*Offset* -1) = 0,

where Byte(*x*) means the byte data whose offset in firmware is *x*.

Assuming that there are n string reference addresses in *srAddr* and they are presented as $sraddr_1$, $sraddr_2$, ..., $sraddr_n$. Corresponding to n string reference addresses, the judging results are stored in $J = (j_1, j_2, \ldots, j_n)$, where

$$j_i = \begin{cases} 1, & if\ sraddr_i\ points\ to\ the\ beginning\ of\ a\ string \\ 0, & else \end{cases}. \tag{4}$$

At last, the matching rate (represented by M) is defined as follows:

$$M = \frac{\sum_{i=1}^{n} j_i}{n}. \tag{5}$$

As described before, under the correct base address, every string reference address loaded by *la* must point to the beginning of a string. Therefore, under the correct base address, the matching rate is higher than that of any other candidate base addresses. Based on this characteristic, after all the matching rates corresponding to the candidates are calculated, the correct base address can be found out from the candidates.

The details of this algorithm are shown in Algorithm 2. Parameters *lower_addr* and *upper_addr* are respectively the bounding addresses of the range of candidate base address.

Algorithm 2: The String Reference Matching (SRM) Algorithm.

```
Input: lower_addr, upper_addr, wndsize
Output: matching rates M[]
use AAS algorithm with window size wndsize to search
for lui-addiu instruction pair and record the string
reference addresses in srAddr[];
use O[] to store the offsets, J[] to store the matching
results and M[] to store the matching rates;
n = sizeof(srAddr[]);
for base = lower_addr; base ≤ upper_addr; base ++ do
    J[base] = 0;
    for i = 0; i < n; i ++ do
        O[i] = srAddr[i] - base;
        if Byte(O[i] - 1) == 0 && Byte(O[i]) ≥ 0x20 &&
        Byte(O[i]) ≤ 0x7E && Byte(O[i] + 1) ≥ 0x20 &&
        Byte(O[i] + 1) ≤ 0x7E then
            J[base] += 1;
        end
    end
    M[base] = J[base]/n;
end
return M[]
```

From the above, the SRM algorithm uses string references to calculate matching rates under different candidate base addresses and the candidate corresponding to the highest matching rate is determined as the correct base address.

4. Evaluation

In this section, the proposed method is tested using the experimental set composed of 12 kinds of widely-used firmware files and the experimental results and some analysis are respectively presented.

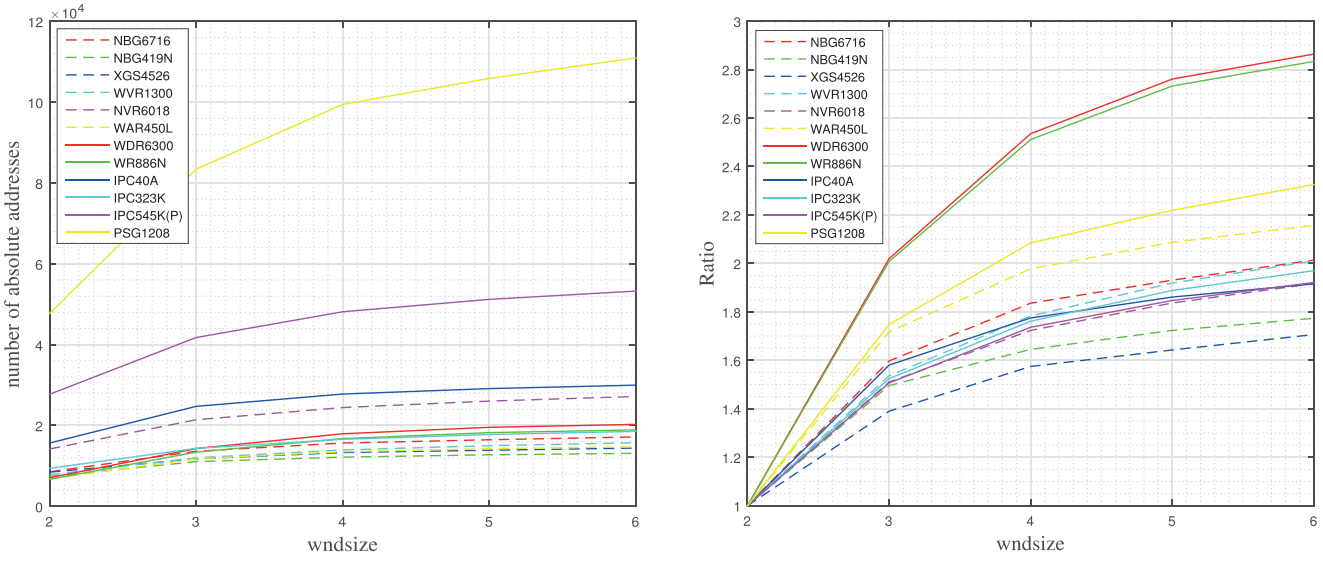
Since there is no common test set can be used in this research field, 12 firmware files are collected, composing a test set to evaluate the validity of the proposed algorithms. This test set covers the most used embedded devices, including IP cameras (IPC), wireless routers and network switches, and the mainstream manufacturers whose products using MIPS architecture processors, including ZyXEL, TP-Link and Phicomm. Besides, this test set also covers big endian and little endian storage patterns to verify the independence to storage patterns. Some information of the firmware files is shown in Table 1. The proposed method is implemented in IDC language using IDA Pro 6.6, and the experiments are performed on the PC with Intel Core i5-4210H 2.90 GHz processor, 8.00 GB memory, and Microsoft Windows 10 operation system.

Table 1 – Information of the tested firmware files.

Manufacturers	Device	Model	File	Big/Little Endian
ZyXEL	Router	NBG6716	2.00(AAJC.8)C0.bin	Big
	Router	NBG419N	100BFQ7C0.bin	Little
	Switch	XGS4526	2.30(AAHH.0)C0.bix	Big
TP-Link	Router	TL-WVR1300	TL-WVR1300Gv1.bin	Big
	Router	TL-WDR6300	wdr6300gv2.bin	Little
	Router	TL-WR886N	wr886nv6.bin	Big
	IP Camera	TL-IPC40A	TL-IPC40A-4v1.bin	Little
	IP Camera	TL-IPC323K	TL-IPC323K-W10_1.0.bin	Little
Phicomm	Router	PSG1208	K1S_V21.4.6.12.bin	Little
	Router	PSG1218	K2_V22.6.503.31.bin	Little
	Router	K2P	K2P_V22.5.14.28.bin	Little
	Switch	FS6800	FS6800-36_v1.0.4.bin	Big

Table 2 – Numbers of absolute addresses under different *wndsize*.

Model	File	Numbers of addresses under different *wndsize*				
		2	3	4	5	6
NBG6716	2.00(AAJC.8)C0.bin	8528	13,622	15,656	16,466	17,174
NBG419N	100BFQ7C0.bin	7395	11,060	12,166	12,745	13,114
XGS4526	2.30(AAHH.0)C0.bix	8447	11,741	13,302	13,876	14,413
TL-WVR1300	TL-WVR1300Gv1.bin	7820	12,009	13,943	15,007	15,718
TL-WDR6300	wdr6300gv2.bin	14,179	21,416	24,438	26,041	27,180
TL-WR886N	wr886nv6.bin	6831	11,732	13,509	14,257	14,734
TL-IPC40A	TL-IPC40A-4v1.bin	7079	14,297	17,947	19,544	20,273
TL-IPC323K	TL-IPC323K-W10_1.0.bin	6673	13,399	16,757	18,232	18,909
PSG1208	K1S_V21.4.6.12.bin	15,652	24,734	27,781	29,129	29,983
PSG1218	K2_V22.6.503.31.bin	9409	14,342	16,574	17,766	18,536
K2P	K2P_V22.5.14.28.bin	27,725	41,765	48,143	51,208	53,263
FS6800	FS6800-36_v1.0.4.bin	47,719	83,438	99,460	105,902	110,969



(a) (b)

Fig. 9 – Changes of absolute address numbers under different *wndsize*.

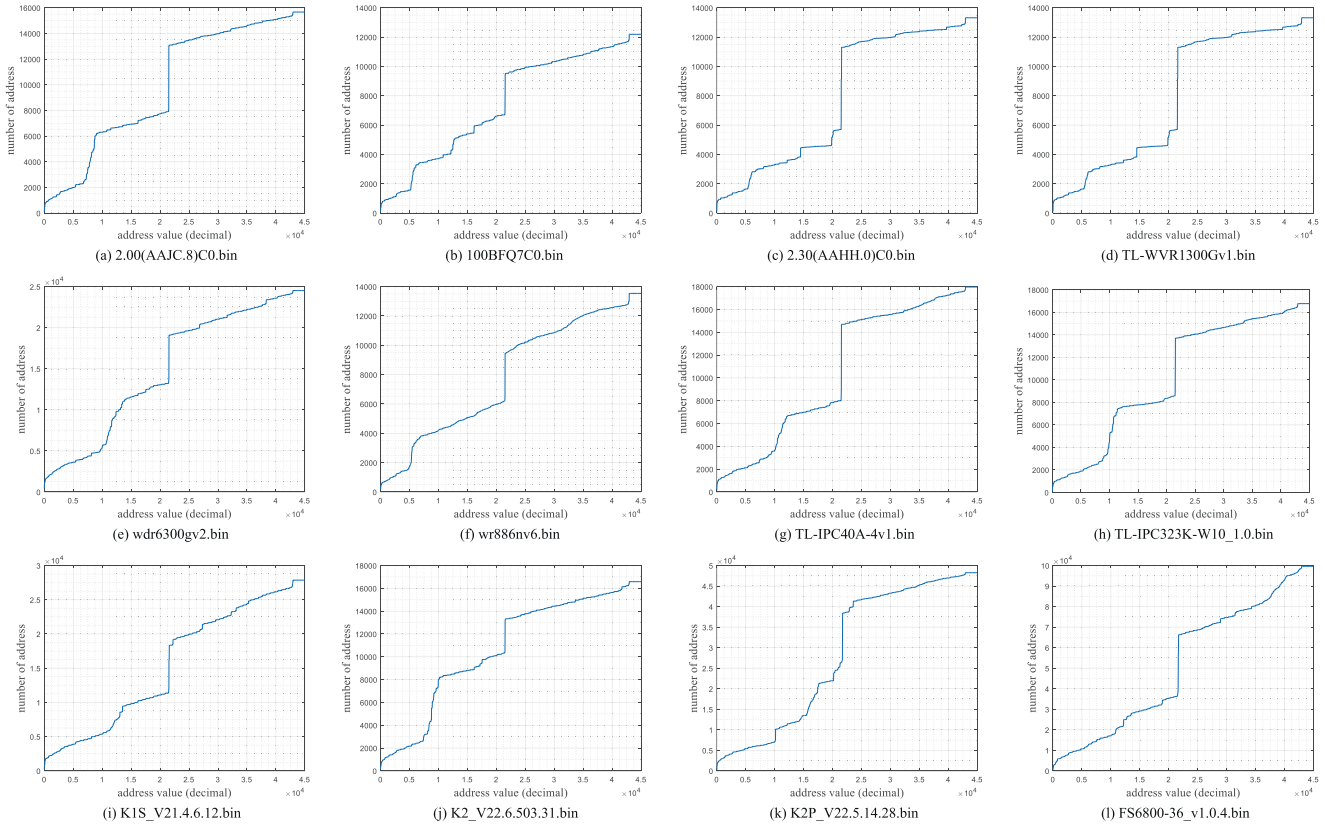


Fig. 10 – Accumulative curves of the absolute addresses.

4.1. *Experiment of absolute addresses statistics*

In this experiment, AAS algorithm is performed on the test set while its wndsize is set to 2, 3, 4, 5 and 6, to respectively determine the absolute addresses in these firmware files. The results are shown in Table 2.

Based on the results shown in Table 2, the numbers of absolute addresses recorded by AAS algorithm and their changing trends as *wndsize* gets bigger are shown in Fig. 9 (a). However, the changing trends of some firmware files are not obvious enough in this figure, although it still tells us that the amounts between different firmware vary greatly. In order to demonstrate the changes as *wndsize* gets bigger more obviously, the data is normalized and Fig. 9 (b) shows the rates of change compared to the numbers of addresses when *wndsize* is 2. It can be seen that for every firmware, the amount of absolute addresses is always increasing as *wndsize* grows and when the *wndsize* changes into 3 from 2, the amount of addresses possesses the most intense increase.

Besides, up to 5, the rates of increase decreases gradually, but from 5 to 6, there is a slight uptick in some firmware. The reason is that when the *wndsize* comes to 6, some instructions that happen to satisfy our conditions but not intend to load absolute addresses are misidentified as characteristic instruction pairs. When loading an absolute address, people rarely put two loading steps very far from each other. Furthermore, when the *wndsize* is getting bigger, the computation complexity is also increasing, which will cause the growth of searching time.

Depending on the above results and analysis, the following conclusion can be drawn about the proper value of *wndsize*. Although with the increase of *wndsize*, the number of absolute addresses searched from the firmware grows correspondingly, it is not always better for bigger *wndsize*. If set the *wndsize* too big, it would not only waste too much time to search the addresses, but also bring in some misrecognitions. Taking these into account, our empirical value of *wndsize* is set to 4.

4.2. *Experiment of range determination*

Based on the absolute addresses statistics, the ranges of candidate base addresses can be determined. Utilizing the approach introduced in Section 3.2, the accumulative curves of recoded absolute addresses can be drawn and the accumulative curves of the tested firmware files are shown in Fig. 10.

The ranges of candidate base address could be determined by reading the *x* coordinates of the boundings of the longest vertical lines on accumulative curves in Fig. 10. Results are shown in Table 3.

Table 3 demonstrates that the ranges of candidate base addresses of all the firmware in the test set can be determined, which verifies the effectiveness and usability of the range determination process.

4.3. *Experiment of base address determination*

After ascertaining the range of candidates, the SRM algorithm can be performed to determine the correct base address from

Table 3 – Ranges of candidate base addresses.

Model	File	Candidate base address range
NBG6716	2.00(AAJC.8)C0.bin	0x7FFEB940 ~ 0x8038B500
NBG419N	100BFQ7C0.bin	0x7FFEB940 ~ 0x804C8B20
XGS4526	2.30(AAHH.0)C0.bin	0x7FFEB940 ~ 0x8081F300
TL-WVR1300	TL-WVR1300Gv1.bin	0x7FFEB940 ~ 0x8038B500
TL-WDR6300	wdr6300gv2.bin	0x7FFEB940 ~ 0x80511F00
TL-WR886N	wr886nv6.bin	0x7FFEB940 ~ 0x8038B500
TL-IPC40A	TL-IPC40A-4v1.bin	0x7FFEB940 ~ 0x8047F740
TL-IPC323K	TL-IPC323K-W10_1.0.bin	0x7FFEB940 ~ 0x8044EA00
PSG1208	K1S_V21.4.6.12.bin	0x7FFEB940 ~ 0x80A384C0
PSG1218	K2_V22.6.503.31.bin	0x7FFEB940 ~ 0x803ECF80
K2P	K2P_V22.5.14.28.bin	0x7FFEB940 ~ 0x8189ED20
FS6800	FS6800-36_v1.0.4.bin	0x7FFEB940 ~ 0x8149D6E0

Table 4 – Numbers of searched string reference addresses (*wndsize* = 4).

Model	File	Size(KB)	Number of string reference
NBG6716	2.00(AAJC.8)C0.bin	3103	9690
NBG419N	100BFQ7C0.bin	4755	6117
XGS4526	2.30(AAHH.0)C0.bix	7569	6471
TL-WVR1300	TL-WVR1300Gv1.bin	3036	9373
TL-WDR6300	wdr6300gv2.bin	4844	14,611
TL-WR886N	wr886nv6.bin	2307	10,672
TL-IPC40A	TL-IPC40A-4v1.bin	4391	12,982
TL-IPC323K	TL-IPC323K-W10_1.0.bin	4107	12,007
PSG1208	K1S_V21.4.6.12.bin	10,138	14,962
PSG1218	K2_V22.6.503.31.bin	3648	10,194
K2P	K2P_V22.5.14.28.bin	7955	28,746
FS6800	FS6800-36_v1.0.4.bin	19,186	73,651

the candidates. However, as shown in Table 4, the number of identified string references is usually enormous and it takes a lot of calculating resource and time to verify every one of them while the improvement of actual detection effect is not obvious. Therefore, only a portion of the string reference addresses need to be extracted to compose a more simple verification sequence. In order to take the whole firmware file into account, systematic sampling (Khan and Shabbir, 2016) is used to uniformly extract some addresses from all the string reference addresses. The numbers of the searched string reference addresses in the firmware and the corresponding file sizes are shown Table 4.

From the results it can be seen that their amounts vary with size of the files. Generally, with the increase of file size, the amount of string reference addresses increases as well, which means that more addresses are needed to ensure the correctness of matching. In our experiment, for the files that are smaller than 10,000 KB, a little over five hundred of string reference addresses are extracted uniformly from all the researched addresses and for those bigger than 10,000 KB but smaller than 20,000 KB, the value of extracted addresses m is set to a little over one thousand. Following is the formula to roughly calculate the extraction number:

$$m = \left\lfloor \frac{M}{Step} \right\rfloor, \tag{6}$$

where M is the total number of recorded string reference addresses and $Step$ is the step length of extraction. $Step$ is calculated by the following formula:

$$Step = \left\lfloor \frac{M}{500 \times \lceil Size/10000 \rceil} \right\rfloor, \tag{7}$$

where $Size$ is the size of the firmware file (KB).

Utilizing these formulas, the number of string reference addresses needs to be extracted is determined and the performance of matching can be greatly improved while the correctness can still be maintained.

After that, the matching rates of all the extracted string reference addresses under each candidate base address are calculated. The matching results of all the firmware files in the test set are shown in Fig. 11.

It is pretty obvious that there always exists an address in every figure where the matching rate is higher than any other addresses. Depending on all the above analysis it can be asserted that these addresses are the base addresses. It is noticed that the matching rate under the right base address is not 100%. The reason is that a number of *la* instructions are not for string referencing. When the matching rate is calculated the number of these not-string-referencing *la* instructions is added to the denominator n in Formula 5 and thereby results in a lower matching rate. Although the matching rate of the correct base address gets lower, it far exceeds the matching rates of all the other candidates, thus the experimental result still behaves a good performance.

The results of base address determination are shown in Table 5.

It is shown in Table 5 that for all the firmware in the test set, the proposed method can automatically determine their base addresses and shows a very good performance.

5. Discussions

In this section, the correctness of the determined base addresses are verified and another application scenario of the proposed method is presented.

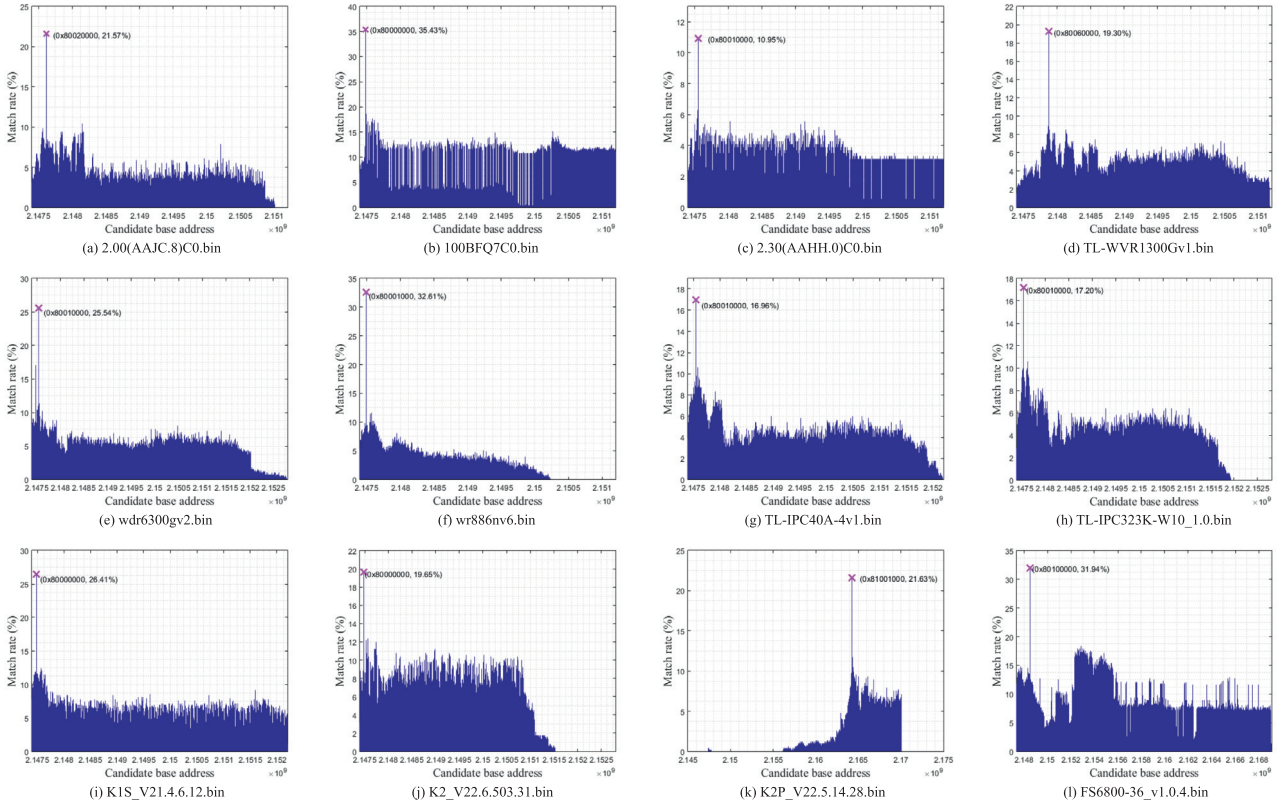


Fig. 11 – Matching rates of candidate addresses.

```
ROM:80020180 loc_80020180:                                  # CODE XREF: sub_800200C0+98↑j
ROM:80020180                la      $v0, aSbinInit          # "/sbin/init"
ROM:80020188                move    $a0, $v0
ROM:8002018C                addiu   $a1, $s0, 0x54F0
ROM:80020190                addiu   $a2, $s1, (off_802E5460 - 0x802E0000)
ROM:80020194                jal     sub_8002C69C
ROM:80020198                sw      $v0, 0x54F0($s0)
ROM:8002019C                la      $v1, aEtcInit           # "/etc/init"
ROM:800201A4                move    $a0, $v1
ROM:800201A8                addiu   $a1, $s0, 0x54F0
ROM:800201AC                addiu   $a2, $s1, (off_802E5460 - 0x802E0000)
ROM:800201B0                jal     sub_8002C69C
ROM:800201B4                sw      $v1, 0x54F0($s0)
ROM:800201B8                la      $v1, aBinInit           # "/bin/init"
ROM:800201C0                move    $a0, $v1
ROM:800201C4                addiu   $a1, $s0, 0x54F0
ROM:800201C8                addiu   $a2, $s1, (off_802E5460 - 0x802E0000)
ROM:800201CC                jal     sub_8002C69C
ROM:800201D0                sw      $v1, 0x54F0($s0)
ROM:800201D4                la      $v1, aBinSh             # "/bin/sh"
ROM:800201DC                move    $a0, $v1
ROM:800201E0                addiu   $a1, $s0, 0x54F0
ROM:800201E4                addiu   $a2, $s1, (off_802E5460 - 0x802E0000)
ROM:800201E8                jal     sub_8002C69C
ROM:800201EC                sw      $v1, 0x54F0($s0)
ROM:800201F0                lui     $a0, 0x802B
ROM:800201F4                jal     sub_800504B4
ROM:800201F8                la      $a0, aNoInitFound_Tr    # "No init found.  Try passing init= optio"
ROM:800201FC
ROM:800201FC loc_800201FC:                                  # CODE XREF: sub_800200C0+60↑j
ROM:800201FC                j       loc_80020150

string reference
code reference
data reference
```

Fig. 12 – IDA loading result when the base address is set to 0x80020000.

Table 5 – Base address determination results.

Model	File	Base address	Verified
NBG6716	2.00(AAJC.8)C0.bin	0x80020000	Yes
NBG419N	100BFQ7C0.bin	0x80000000	Yes
XGS4526	2.30(AAHH.0)C0.bin	0x80010000	Yes
TL-WVR1300	TL-WVR1300Gv1.bin	0x80060000	Yes
TL-WDR6300	wdr6300gv2.bin	0x80010000	Yes
TL-WR886N	wr886nv6.bin	0x80001000	Yes
TL-IPC40A	TL-IPC40A-4v1.bin	0x80010000	Yes
TL-IPC323K	TL-IPC323K-W10_1.0.bin	0x80010000	Yes
PSG1208	K1S_V21.4.6.12.bin	0x80000000	Yes
PSG1218	K2_V22.6.503.31.bin	0x80000000	Yes
K2P	K2P_V22.5.14.28.bin	0x81000000	Yes
FS6800	FS6800-36_v1.0.4.bin	0x80100000	Yes

5.1. *The verification of experimental results*

To verify whether the experimental result is correct, *2.00(AAJC.8)C0.bin* file of ZyXEL Router NBG6716 is loaded to IDA Pro, with the processor type set to MIPS big-endian and the base address to 0x80020000. The result is shown in Fig. 12. By comparison, the same file loaded by IDA Pro without setting the proper base address is shown in Fig. 13.

From Fig. 12 it can be seen that IDA Pro can identify some binary functions, code references and data reference, and some of the absolute addresses loaded by instruction pairs mentioned in this manuscript point to strings and are displayed as meaningful string names (e.g. 32-bit address 0x802812E0 at position ROM 180 in Fig. 13 points to string aS-binInit whose content is "/sbin/init" in Fig. 12). This means that cross references match nicely, which indicates the base address determined by our algorithm is correct. In contrast, in Fig. 13 there are some extra data references. Tracing to the referencing code and we can only find that there are data segment, as shown in Fig. 14, which means the references are aberrant.

With the same method, all the base addresses shown in Table 5 are verified and the results indicate that the determined base addresses are exactly the correct base addresses.

```
ROM:00000180 loc_180:                                # DATA XREF: ROM:002C97FC↓o
ROM:00000180                 lui     $v0, 0x802B
ROM:00000184
ROM:00000184 loc_184:                                # DATA XREF: sub_261DA4+6D4↓r
ROM:00000184                 addiu   $v0, 0x12E0
ROM:00000188
ROM:00000188 loc_188:                                # DATA XREF: sub_191C14+20044↓r
ROM:00000188                                         # sub_261DA4+6DC↓r ...
ROM:00000188                 move    $a0, $v0
ROM:0000018C                 addiu   $a1, $s0, 0x54F0
ROM:00000190
ROM:00000190 loc_190:                                # DATA XREF: sub_21E4A8+56414↓r
ROM:00000190                                         # sub_2787E8+78↓r
ROM:00000190                 addiu   $a2, $s1, 0x5460
ROM:00000194
ROM:00000194 loc_194:                                # DATA XREF: sub_261DA4+6E4↓r
ROM:00000194                 jal     sub_2C69C
ROM:00000198
ROM:00000198 loc_198:                                # DATA XREF: sub_261DA4+6EC↓r
ROM:00000198                                         # sub_22BF84+40084↓r
ROM:00000198                 sw      $v0, 0x54F0($s0)
ROM:0000019C
ROM:0000019C loc_19C:                                # DATA XREF: sub_1F37E0+200EC↓o
ROM:0000019C                                         # sub_2138F0:loc_2138F4↓o ...
ROM:0000019C                 lui     $v1, 0x802B
ROM:000001A0
ROM:000001A0 loc_1A0:                                # DATA XREF: ROM:00194FD4↓r
ROM:000001A0                                         # ROM:00302078↓o ...
ROM:000001A0                 addiu   $v1, 0x12EC
ROM:000001A4
ROM:000001A4 loc_1A4:                                # DATA XREF: sub_261DA4+6F4↓r
ROM:000001A4                                         # sub_24745C+20A04↓r ...
ROM:000001A4                 move    $a0, $v1
ROM:000001A8
ROM:000001A8 loc_1A8:                                # DATA XREF: ROM:002CC5F4↓o
ROM:000001A8                 addiu   $a1, $s0, 0x54F0
ROM:000001AC                 addiu   $a2, $s1, 0x5460
ROM:000001B0
ROM:000001B0 loc_1B0:                                # DATA XREF: sub_261DA4+568↓r
ROM:000001B0                                         # sub_261DA4+570↓r ...
```

Fig. 13 – IDA loading result when the base address is set to 0.

```
ROM:002C97EC                .word 0x80120C3C, 0
ROM:002C97F4                .word 0x802C09B0, 0
ROM:002C97FC                .word loc_180
ROM:002C9800                .byte 0x80  # ■
ROM:002C9801                .byte 0x12, 0xC, 0x20
ROM:002C9804                .word 0x80120DE0, 0x802BAE44, 0
ROM:002C9810                .word loc_1A4
ROM:002C9814                .word 0x80120C08, 0x80120A80, 0x802BAE5C, 0
ROM:002C9824                .word loc_1A4
ROM:002C9828                .word 0x80120BDC, 0x80120B08, 0
ROM:002C9834                .word 0
ROM:002C9838                .byte    0
ROM:002C9839                .byte 0, 0, 0
ROM:002C983C                .word 0, 0, 0, 0
```

Fig. 14 – Segment of firmware referencing the data at ROM 180.

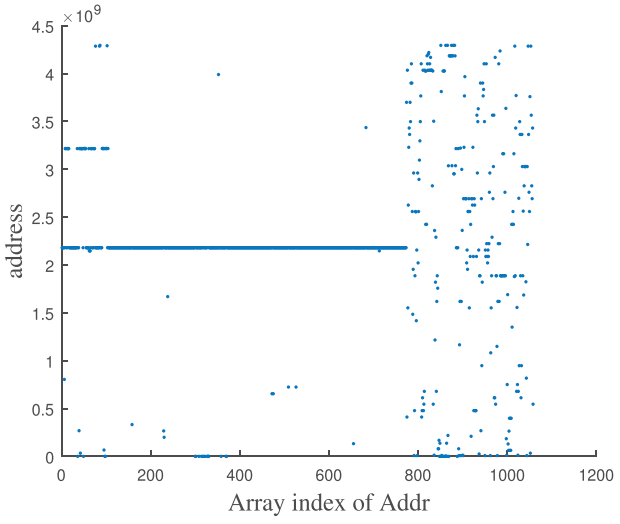


Fig. 15 – Addresses in *Addr* of *decompressed.bin*.

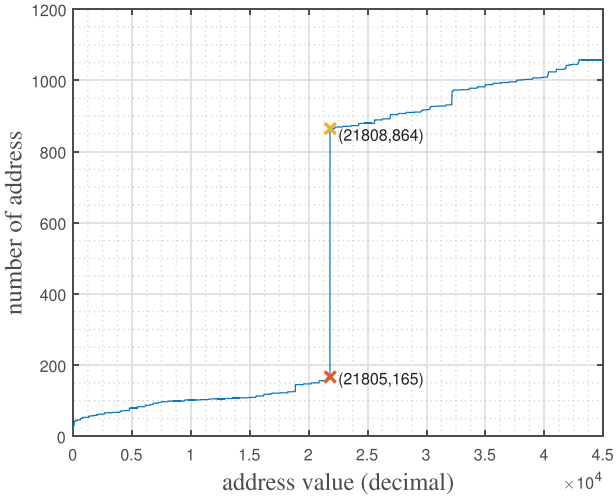


Fig. 16 – Accumulative curve of addresses in *Addr* of *decompressed.bin*.

5.2. *Another application scenario*

The proposed method can also be used to determine the loading addresses of a part of the firmware file, such as a section of decompressed data or moved codes. By analyzing the loading addresses, the boot process and memory layout of the embedded device can be depicted, which is of great significance for the reverse engineering (Cui et al., 2013).

In firmware *2.00(AAJC.8)C0.bin*, there is a section that is compressed by LZMA (Lempel-Ziv-Markov chain-Algorithm) algorithm (Konstantinou and Maniatakos, 2015). Decompress it as *decompressed.bin* and use AAS algorithm to detect its absolute addresses, the recorded addresses are shown in Fig. 15.

Compared with the absolute addresses in the whole firmware, the addresses in this compressed section has a more obvious accumulation. Using the method introduced in Section 3.2, their accumulative curve is drawn in Fig. 16.

From the accumulative curve, it can be ascertained that the range of possible loading addresses is 2,180,500,000 (0x81F7CA20) ∼ 2,180,800,000 (0x81FC5E00). Use SRM to calculate the matching rates of these possible loading addresses and the results are drawn in Fig. 17.

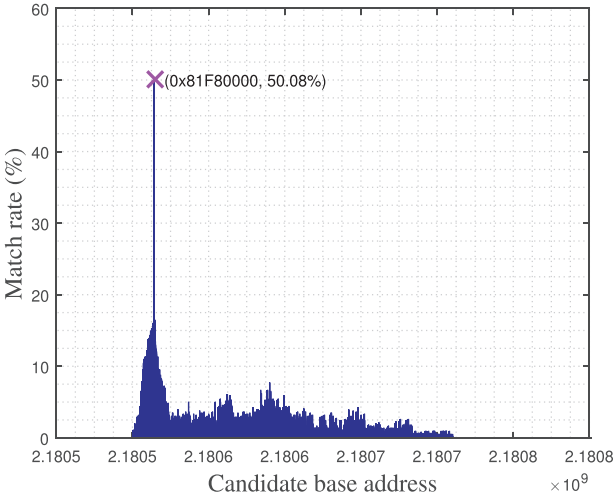


Fig. 17 – Matching rates of candidate base addresses of *decompressed.bin*.

The results indicates that the loading address is 0x81F80000. It means that this section of LZMA compressed firmware will be decompressed to ROM address 0x81F80000 in the root process of the device. The correctness is also verified by loading to IDA.

Therefore, the proposed method is also efficient for automatically determining the loading address of a section of the firmware, which is very helpful for the layout analysis.

6. Conclusions and future work

Aiming at the automatic base address determination of MIPS firmware files, a method based on absolute address statistics and string reference matching is proposed. It utilizes specific instruction pairs to recognize target operations, adopts a statistical method to determine the approximate range and uses string references to determine the base addresses. Firstly, the 32-bit immediate value loading, addressing mode and string referencing of the MIPS architecture systems are analyzed. Secondly, the AAS algorithm and SRM algorithm are proposed to realize the determination of the base address. The AAS algorithm can recognize three types of absolute address loading and record the absolute addresses involved. Using the distribution curves of the recorded addresses, the ranges of candidate base addresses can be determined. In order to pick out the right base address from the candidates, the SRM algorithm is proposed. It recognizes all the string references in the firmware, records the string reference addresses and calculates the matching rates corresponding to each candidate base address. Among all the candidates, the one corresponding to the highest matching rate is determined as the right base address. Lastly, the experiments are performed and the results demonstrate that the proposed method not only can be used to determine the base addresses of MIPS firmware automatically and accurately, but also can be applied to determine the loading addresses of firmware sections efficiently.

A possible goals for our future research would be to identify the *la* instructions that are for string referencing and find a more appropriate sampling method to improve the performance.

Conflict of interest

None.

Acknowledgment

This research was supported by the National Natural Science Foundation of China (No. 61802431).

REFERENCES

Basnight Z, Butts J, JL Jr, et al. Firmware modification attacks on programmable logic controllers. Int J Crit Infr Prot 2013a;6(2):76–84.

Basnight Z, Butts J, Lopez J, et al. Analysis of programmable logic controller firmware for threat assessment and forensic investigation. In: Proceedings of the 8th international conference on information warfare and security; 2013b. p. 9.

Blem E, Menon J, Sankaralingam K. Power struggles: revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures. In: Proceedings of IEEE international symposium on high performance computer architecture. IEEE Computer Society; 2013. p. 1–12.

Chen DD, Egele M, Woo M, et al. Towards automated dynamic analysis for linux-based dmbedded firmware. Proceedings of symposium on network and distributed system security, 2016.

Costin A, Zaddach J, Francillon A, et al. A large-range analysis of the security of embedded firmwares. In: Proceedings of USENIX security symposium; 2014. p. 95–110.

Costin A, Zarras A. Automated dynamic firmware analysis at range: a case study on embedded web interfaces. In: Proceedings of ACM on Asia conference on computer and communications security. ACM; 2016. p. 437–48.

Cui A, Costello M, Stolfo SJ. When firmware modifications attack: a case study of embedded exploitation. Proceedings of symposium on network and distributed system security, 2013.

Heffner C. Reverse engineering vxworks firmware: Wrt54gv8, 2011. Avaliable from: http://www.devttys0.com/2011/07/reverse-engineering-vxworks-firmware-wrt54gv8/.

Heffner C. Binwalk firmware analysis tool, 2013. Available from: http://binwalk.org/.

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

Khan Z, Shabbir J. Some remarks on the traditional way of circular and diagonal circular systematic sampling schemes. Commun in Stat-Theor M 2016;45(7):2105–17.

Konstantinou C, Maniatakos M. Impact of firmware modification attacks on power systems field devices. In: Proceedings of IEEE international conference on smart grid communications. IEEE; 2015. p. 283–8.

Rekoff MG. On reverse engineering. IEEE Trans Syst Man Cy-S 1985;15(2):244–52.

Schuett C, Butts J, Dunlap S. An evaluation of modification attacks on programmable logic controllers. Int J Crit Infr Prot 2013;7(1):61–8.

Skochinsky I. Intro to embedded reverse engineering for pc reversers. Proceedings of the REcon conference, Montreal, Canada, 2010.

Sloss A, Symes D, Wright C. ARM system developer's guide: designing and optimizing system software. Elsevier; 2004.

Sweetman D. MIPS run. Elsevier; 2010.

Yan S, Wang R, Hauser C, et al. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. Proceedings of symposium on network and distributed system security, 2015.

Zaddach J, Costin A. Embedded devices security and firmware reverse engineering. Black-Hat USA; 2013.

Zhu R, Tan Y, Zhang Q, et al. Determining image base of firmware files for ARM devices. IEICE Trans Inf Syst 2016a;99(2):351–9.

Zhu R, Tan Y, Zhang Q, et al. Determining image base of firmware for ARM devices by matching literal pools. Digit Invest 2016b;16:19–28.

Zhu R, Zhang B, Mao J, et al. A methodology for determining the image base of arm-based industrial control system firmware. Int J Crit Infr Prot 2017;16:26–35.

Xiaodong Zhu is a Ph.D. candidate in State Key Laboratory of Mathematical Engineering and Advanced Computing, China. His research focuses on reverse engineering and embedded system security.

Yi Zhang is a Ph.D. candidate in State Key Laboratory of Mathematical Engineering and Advanced Computing, China. Her research focuses on image steganography and information security.

Liehui Jiang is a professor in State Key Laboratory of Mathematical Engineering and Advanced Computing, China. His research focuses on the theory and application of reverse engineering, code analytics and formal sematic.

Rui Chang is an associate professor in State Key Laboratory of Mathematical Engineering and Advanced Computing, China. Her research focuses on embedded system security, formal verification and security enhancement using TrustZone.