ARM Static Library Identification Framework

Qing Yin, Fei Huang, Liehui Jiang
National Digital Switching System Engineering & Technology Research Center
1001-718#, Zhengzhou, Henan Province 450002, China
baxiahf@126.com

***Abstract*—A static library identification framework is proposed through studying library as "dcc", which dynamically extracts binary characteristic file on applications under ARM processor. This framework obtains function modules according to ARM assemble characteristics, on the basis of which dynamic signature is generated due to pattern files through analyzing coding characteristics of assemble addressing mode and that of corresponding binary codes, then signatures of candidate functions are matched with signatures of library functions using hash algorithm to identify library functions. This method can recognize library functions efficiently and solve conflict between massive library files and matching efficiency effectively.**

***Index Terms*—library function; ARM; pattern matching; characteristic signature**

I. INTRODUCTION

Decompilation is one technique that can translate object codes into corresponding high-level language expression. Then, the static linked library functions in legacy software are difficult to decompile because of whose own code characteristics, and static library functions are code modules which are linked statically and depend on compilers, so it is beneficial for decreasing the work of decompilation and increasing validity and readability of identification to recognize static library function modules, which is good to understand program intention and analyzing program's key parts.

Embedded devices are applied abroad in every fields, and its processor types are over one thousand so far. Now microprocessors applied ARM technique mainly take up over 70 percent of RISC microprocessor' share, whose market is broad. However, the related literatures at home and abroad of library identification concentrated on ARM processor are few, while the requirement of reuse, development and key functions' extraction of legacy software are increasingly urgent.

Static library functions are function modules that are embedded into program in function body form, while dynamic library functions are function modules that are applied dynamically. The form of library function existing in executables is close to processor architecture [1]. Static library functions are compiled by compiler of special version, so library files of different version' compilers vary in formation and content.

Existing static library functions identification method on ARM is relatively few, most are for x86 processor, which firstly extracts characteristic library, then compares extracted function modules with characteristic library to fulfill identification. According to characteristic extracting level, there are mainly two aspects:

(1) Library functions pattern extraction based on intermediate code level; *8086C* decompiler brings out a method that includes library functions' main characteristics and secondary characteristics, then to generate pattern table. According to different characteristics, eight matching methods are proposed. This method is very strict in theory and complex in application. On the basis of disassembling library functions, Mr. Li Xiang Yang extracts library pattern on intermediate code level, which is matched with candidate function modules [2]. The arithmetic is simply fast, but its weak point is ignoring the characteristic of operand and addressing type and clashes during match process.

(2) Library functions pattern extraction based on object code level; IDA and DCC use FLIRT technique, which generates pattern files using first n bytes of object programs, on the basis of which binary pattern library [3] is generated. The signature is obtained by hashing pattern files in library, which will be matched in matching process. This method results effective and efficient, and can recognize static library functions in object codes well.

On analyzing static library function identification arithmetic of x86 processor, combined with coding characteristic of ARM architecture in executables, a library function identification arithmetic on ARM whose characteristic length can be defined is proposed. ARM static library function identification apply extracting binary coding of library to fulfill the uniqueness and integrality of characteristic obtaining, using hash arithmetic to fulfill the automation and efficiency of matching process, using characteristic length that can be defined to fulfill the control of matching precision.

II. STATIC LIBRARY RECOGNITION MODEL

Static library functions' popular process is as Figure. 1 [4]:

Library function recognition model generally contains four modules:

doi:10.4304/jsw.5.10.1099-1106

Function Characteristic Recognition Module: object codes are separated to modules by analyzing "BL" and "BLX" etc characteristic instructions after disassembled, which can obtain all the functional code segments.

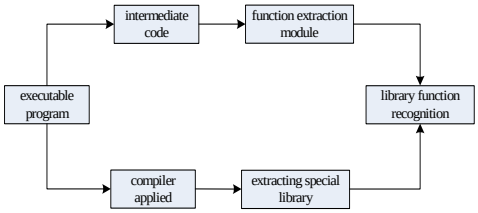

Fig.1 Static library function identification process

Compiler Characteristic Recognition: by extracting and analyzing characteristic codes at the beginning of programs, the type of compiler used in compiling process can be obtained.

Pattern File Characteristic Extraction: by analyzing library functions, characteristic should be extracted to describe a file or program uniquely, which is stored into library.

Matching Module: Given functions are matched with corresponding characteristic library using pattern matching algorithm.

Static library function recognition process based on dynamic signature is as figure 2: executable files first are transformed to equal assemble codes, then recognize function modules based on jump or call, which can obtain beginning address and relate information of binary-form function. The start codes are extracted from codes that are disassembled from executable program, then related information like programming language, compiler and memory model etc. is obtained by matching start codes at intermediate language level. Pattern files [5] library is generated from library files, and corresponding pattern file set is extracted after recognizing compiler type. Finally matching algorithm matches given functions with signature files and outputs the result.

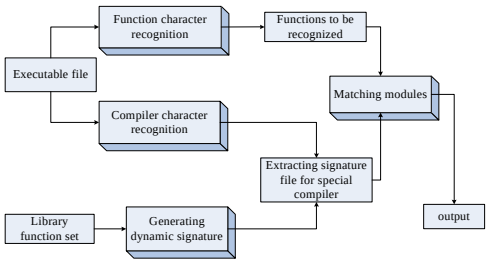

Fig.2 Static library function recognition model based on dynamic signature

Studying static library function recognition, this paper proposes a characteristic matching three-layer model: function module characteristic recognition; compiler characteristic recognition; pattern files' characteristic recognition, which are as figure 3:

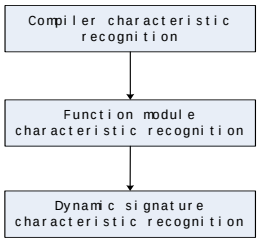

Fig.3 Characteristic matching model

Library function matching model: layer 1, utilizing unique start codes of files, the assemble level characteristic of compiler version is constructed, after which start codes are recognized; layer 2, utilizing feature of ARM assemble codes, function modules are extracted on recognizing entry instructions and ret instructions; layer 3, utilizing generated dynamic signature library, functions are matched and recognized using pattern matching.

III. ARM ASSEMBLE LEVEL CHARACTERISTIC

Static library function recognition first recognizes compiler type and function modules on applications. Due to unique characteristic codes of compiler, and the codes to be recognized are relatively less, so the characteristic of compiler is extracted on assemble level. After object programs are disassembled, function modules are extracted by analyzing and recognizing call instructions.

A. Compiler Characteristic

The expression of static library function in object codes is dependent on compiler, and the expression of the same library function's codes is different if compiled by different types of compiler. Even if the object codes are compiled by the same type of compiler with different version, the expression may be different, so the compiler should be recognized before recognizing static library functions. The library file's types supported by the same compiler vendor for different memory models are different [6], so all the information should be recognized when decompiling library files.

There are some initialized codes at the beginning of executable program called start codes, which run before program's real codes and solve the difference between parameters that transferred by loader and that received by programs. Initial codes should be executed before "main" function gets control. These codes vary between different compilers, so compiler can be recognized effectively by extracting assemble characteristic on start codes. Supposed assemble start codes is "A_c", recognize process is "δ", start code library is $\sum = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4, ...\}$, recognition process is: $A_c \in \sum \Leftrightarrow \delta(A_c, \sum) = 1$.

```
.text:020005A4                 EXPORT _start
.text:020005A4 _start
.text:020005A4                 MOV     R11, #0
.text:020005A8                 LDMFD   SP!, {R1}
.text:020005AC                 MOV     R2, SP
.text:020005B0                 STMFD   SP!, {R0}
.text:020005B4                 LDR     R0, =.term_proc
.text:020005B8                 STMFD   SP!, {R0}
.text:020005BC                 LDR     R0, =main
.text:020005C0                 LDR     R3, =.init_proc
.text:020005C4                 BL      .__libc_start_main
.text:020005C8                 BL      .abort
```

Fig.4 start codes of executable program

This paper extracts assemble level start code as signature, which is based on disassembling of executable program by IDA. The assemble codes between "_start" [7] and "main" are module of recognizing compiler. Compiler characteristic library is constructed by extracting characteristic files from start codes of executable files, which will be matched when recognizing compiler's type.

B. Function Module Characteristic

Static library function recognition is matching with embedded modules in executable programs. Due to compile optimization and relocation [8], library function may varies after compiled. So matching on object codes bit by bit is inaccurate, and object codes' size may be relatively big. Rough matching may wastes lots of time and can't fulfill good impact. So object codes should be disassembled and separated into function modules before matching, which gets all the function modules in executable programs and simplifies matching process.

The intermediate language codes for ARM don't own "call" etc. instructions which call subroutines, and there are not "ret n" etc instructions to indicate end address of function. Function module recognition for ARM is fulfilled on assemble level, whose assemble codes' features are as follows:

- Assemble codes for ARM are equal to object codes, which doesn't have definite entry point like PE files when disassembling.
- The call to function modules in ARM assemble instructions is fulfilled by jump in program flow. The jump in flow is fulfilled by two kinds of instructions: special "jump" instruction; writing address value to "PC".
- Subroutine's call mainly applies "BL, BLX", which stores the value of "PC" into an special register (R14) before jumping and recovers the value of "PC" before returning.

Function recognition algorithm is as follows:
Function: recognizing address section of function modules
Input: intermediate language codes for ARM
Output: function module's address section (b, e)
Algorithm:

```
Cbegin = _start;
Begin:
        Code_scan = next line codes;
      If(Code_scan = null)
        Return ;
      Scan（code section）;
        If (there is not jump instruction incodes)
            Goto Begin;
        Else
            Scan（later codes）
          If（find "PC" value is set）
           Output（address sections）;
             Goto Begin;
```

Fig.5 Function recognition algorithm

IV. DYNAMIC SIGNATURE

Signature file is a special format file that is generated when recognizing static library functions, which can be matched with functions recognized from executable files. On the basis of deeply analyzing library functions, every library function body is different to each other, and most of them are different on bytes of the beginning. So this paper extracts first n bytes of library as the characteristic codes of pattern files due to this feature.

A. Library Files Structure Characteristic

ARM library files' format under Linux mainly contains share library(.so file) and static library(.a file). Static library files are library file that exist in ".a" format, which pack several object files(.o file) into a library file, the structure of which mainly contain the following parts:

File characteristic word (magic number): it is a string used for marking file structure, the length is 8, the value is "!<arch>\n".

Section: ".a" file contains four type's sections; first section, second section, longname section and obj section. The second section and longname section are optional, which may not appear in a file. Every section contains two parts: header and data parts.

The format of header is the same to each other, whose content are as follows:

```
struct{
    char    name[16];          //  name
    char    time[12];          //  time
    char    userid[6];         //  user id
    char    groupid[6];        //group id
    char    mode[8];           //  mode
    char    size[10];          //  length
    char endofheader[2];  // 结束符
  } sectionheader;
```

First section is the first section of a file, the structure of whose data part is as follows:

```
struct
{
unsigned long symbolnum;        // symbol number
unsigned long symboloffset[n];          // object section offset
```

```
char strtable[m];                    // symbol name table
}firstsec;
```

The first eight bytes express the number of symbol in file, which are stored in big-endian mode. The next part express all the symbol's object file offsets in files, which are stored in big-endian mode.

Obj section is object code section, which is ELF-format executable file.

Different types of sections in library file will be parsed according to this format feature, and the code modules of every function name will be obtained. One code module may contains several function, so this module may be disassembled and analyzed, and every function' binary code module will be obtained so as to get corresponding function' characteristics.

B. Dynamic Characteristic Extraction

Characteristic extraction is getting some special bytes from a function as its characteristic, and dynamic characteristic extraction can extract characteristic bytes on the requirement of match precision , which fulfills the automation of extraction process according to coding feature. The extraction model is as follows:

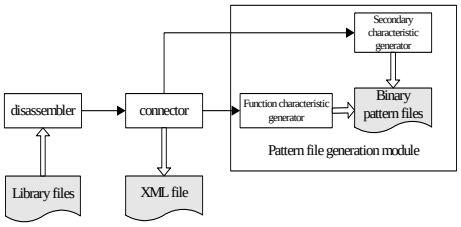

Fig.6 pattern extraction model

The whole process consists of three modules: disassembler, connector, pattern generator. firstly, the executable is disassembled, then the data is extracted and stored in XML files that have fixed structure, later the data in XML file is obtained by pattern generator, whose characteristic is extracted. Finally the binary pattern file is generated by characteristic generator.

Every module's exact function is as follows:

Disassembler: according to the feature of library files' format and ARM instruction set, the assemble codes of output function in library file, the corresponding binary codes as well as function name and some related data like function name is obtained.

Connector: through dealing with the disassembled data generally, all the assemble codes of output functions in library files and binary codes is stored in special formats.

Function body characteristic generator: through analyzing coding form in ARM processor, the characteristic of function codes is extracted on binary level, the variants that change with environment is deleted, then the binary sequences that can specify a function uniquely are obtained.

Auxiliary character generator: through analyzing function body, the characteristic of reference and call in function is extracted, and the short function is announced specially, which make an effect of complement and verification in extraction of characteristic.

Characteristic extraction method:

Opecodes and operands are mixed in first few bytes of object function modules, and the operands of library functions maybe offset or constant, which results in operands in different executables variable. Therefore, variable operands can't be extracted as characteristic uniquely, as well as external reference in codes. These variable bytes should be set as wildcard during pattern files generation process, which leaves constant parts like opecodes and operands which don't change with link and execution. Codes extracted like this are invariant, and is simple to recognize.

It is important understand the operands' coding form in ARM instruction set, and the form and content of operands are close to addressing and coding format. The length of ARM instruction is 32; while the length of Thumb instruction is 16, which is helpful to recognize instruction boundary.

1. operand addressing mode in ARM instruction

ARM instruction set generally consists of nine types of addressing modes:

Immediate addressing:

Operands are stored in instructions as real data, which act in two formats: immediate operand, 32-bits immediate rotated rightly, and the detailed coding is as follows:

Operand type: #<immediate>

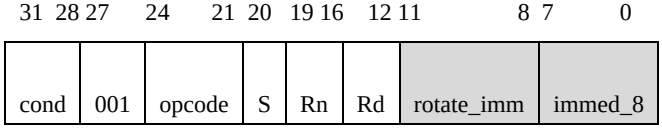

Fig.7 immediate addressing operand coding

The type of operation should be recognized before dealing with an instruction of immediate addressing, then the addressing mode of operand is recognized, finally replace the variant bytes with wildcards according to coding format of instructions.

Register addressing:

Operand is stored in the instruction coding as register number. This form is relatively simple, which don't need replacement with wildcards, the detailed coding is as follows:

Operand type: <Rm>

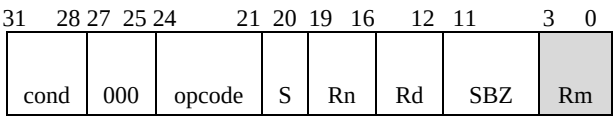

Fig.8 register addressing operand coding

Register shift addressing:

The operand of register addressing is obtained in the form of the register value which is shifted, the value of shift may be immediate or stored in register, the detailed coding is as follows:

Operand type：<Rm>, <shift> <Rs>

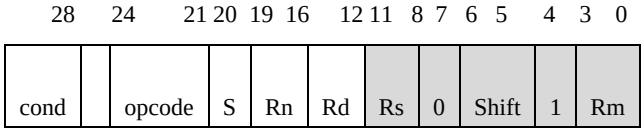

Fig.9 register shift addressing operand coding by register

Operand type：<Rm>, <shift> #<shift_imm>

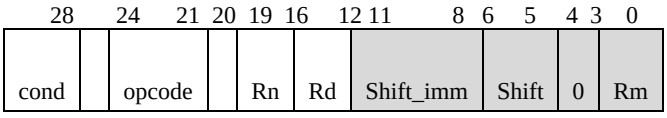

Fig.10 register shift addressing operand coding by immediate

Register indirect addressing:

Register indirect instruction is usually applied in transmission instructions, whose register is stored as the address pointer of operand, which reflects in ARM coding the same as in register addressing, the coding is as follows:

Operand type：[<Rm>]

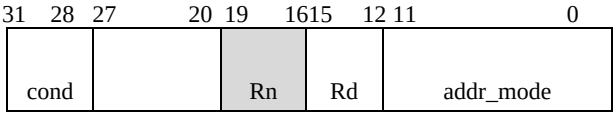

Fig.11 register indirect addressing operand coding

Base plus offset addressing:

Base plus offset addressing is also called indexed addressing, offset value being 4KB, which is divided into three types: pre-indexed, auto-indexed, and post-indexed. The type of indexed addressing is decided by “P” bit and “W” bit. Among which operands have three types, register plus offset, register plus register, and register plus register shifted.

Operand type：[<Rn>, #+/-<offset_12>]

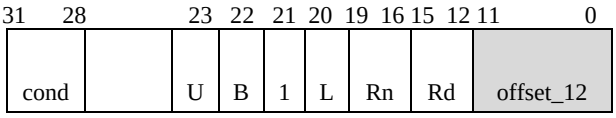

Fig.12 immediate indexed operand coding

Operand type：[<Rn>, +/-<Rm>]

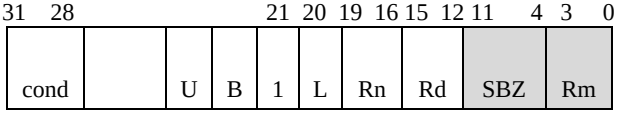

Fig.13 register indexed operand coding

Operand type：[<Rn>, +/-<Rm>, <shift> #<shift_imm>]

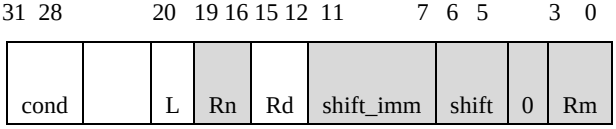

Fig.14 shifted register indexed operand coding

Multi-register addressing:

The instruction that apply multi-register addressing can achieve the transformation of several registers, which may reach 16 at most.

Operand type：<registers>

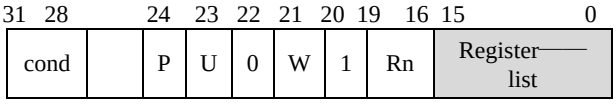

Fig.15 multi-register list operand coding

Stack addressing:

Stack addressing operate on data by visiting stack, which contains two types: the first is up increasing, the second is down increasing.

Operand type：<registers>

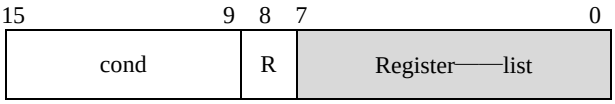

Fig.16 stack operand coding

Relative addressing

Relative addressing can be seen as the jump of object address whose offset to current PC.

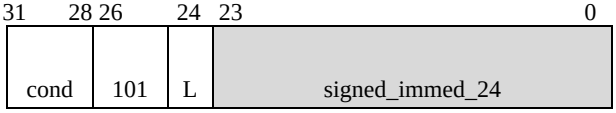

Fig.17 offset operand coding in relative addressing

Block copy addressing:

Block copy addressing can achieve the transmission of successive-address data in memory.

Operand type：<Rn>{!}, <registers>

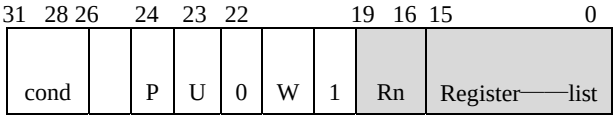

Fig.18 register operand coding in block copy addressing

2. Characteristic extraction mechanism of disassembled codes in ARM processor:

The instructions in executable may have parts that refer to the location of codes in executables, some other parts may refer to program’s data or their location, the other parts may be independent to the location of data or code. However, these parts that refer to location may result in uncertainty in matching process. The invariable parts in instruction usually include opcodes, program relative offsets, stack frame indexes, and some immediate operands. The detailed extraction mechanism is as follows:

1) ARM Instruction :

This is a 32 bits instruction set. Characteristic extraction process is fulfilled by extracting binary

effective instructions that are equal to corresponding assemble codes. Some assemble instructions and their corresponding binary instructions are as follows:

E5 94 F0 00	LDR PC, [R4]
1A FF FF FA	BNE loc_2000500

Fig.19 assemble codes and its binary form

Binary encoding format of "BNE" instruction for ARM is as follows:

31 28	27 25	24	23 0
Cond	101	L	Signed 24-bits offset

Fig.20 "BNE" binary coding form

Because loc_200050 is variant, the first 8 bits will be obtained based on binary encoding format of BNE instruction: 1A . . . . . .

2) Thumb Instruction:

This is a 16 bits instruction set that ARM architecture supports besides efficient 32 bits instruction set, which is an subset of ARM instruction set. Characteristic extraction process is as follows:

1A 00	LDR R3, [R1]
E5 91	B 0x1FFFF34

Fig.21 Thumb instructions and its binary form

Binary encoding format of "B" instruction for Thumb is as follows:

15	14	13	12	11	10 0
1	1	1	0	0	Offset 11

Fig.22 "B" binary coding form

As 0x1FFFF34 is variable, based on binary encoding format of B instruction, the signature is: E . . . .

Based on binary level, characteristic extraction firstly obtains assemble codes which are disassembled, then analyzes characteristic codes based on corresponding instruction type(ARM or Thumb) and binary encoding format of assemble codes, finally extracts codes' characteristics based on corresponding code format. Using this characteristic extraction strategy, the signature is generated automatically, which consequently fulfills dynamic signature.

Some problems:

Link Optimization and Code Confusion:

When executable programs compile and link, its codes' sequence remains relatively stable, if optimization is not applied. When program is optimized, codes' sequence will change, and may make it impossible to fulfill identification using byte matching; code confusion will result in the same effect. If code author uses this technique, it will be hard to be recognize, which makes identification rate descend.

Special function treatment:

When characteristic is extracted, there may be different library routines that own the same first 32 bytes, and it is likely to take place. The method to solve this problem is extract relate information again from this special function, and extract characteristics from later bytes until both library routines can be distinguished, then store byte number and CRC16 when extracting relate information for the second time. If both library routines can't be distinguished at the second extraction, there are two results:

The first case is that one routine is part of another, and the settlement is extracting one more byte from the longer routine, which makes it easy to distinguish these two routines.

The second case is extracted characteristic bytes are the same, which means that both routines' body are identical except for operands and some external references. The identification should be put off if it needs to be recognized completely, and operands or references should be known to fulfill identification. Both library routines' function body are the same now, and recognizing routine body can understand program effectively, so this paper only lists possible library function name instead of the function's real name at this time.

C. Signature Generation

Library pattern files are made up of a series of binary codes, which can be matched with candidate functions. The efficiency is very good if the number of functions to be matched is small, while the efficiency needs to be improved if the size of executables and library functions' pattern files is all huge. Due to which, this paper applies minimal perfect hash arithmetic to generate hash signature for pattern files to fulfill fast match of library functions.

Hash algorithm is a common digital signature algorithm, and it can be called digital fingerprint, which can translate a series of given object codes less than 56 bytes into a set of fixed lengthy unique mark. It is usually used to ensure reality and integrity of data, but its main function is to make a fixed bytes' abstract(hash value). The abstract can identify the abstract of different bytes are unique and mark strings uniquely, which owns good efficiency. The familiar hash algorithms are MD5, SHA-1 etc.

On studying FLIRT algorithm, this paper proposes one fast pattern recognition algorithm. This method sees the whole pattern set as edges in graph, then the pattern files are transformed into vertices in graph, finally the signatures are mapped to vertex pairs, which results into matching patterns applying vertices in graph[9]. The whole signature generation process contains two parts: map process and assignment process.

The map process is the initialization of graph, the number of edges is just that of patterns. Every item in pattern files

should be analyzed and the alphabet size is obtained. A random list is generated for every item in the alphabet, after which the soundness of random list is analyzed to void one edge being mapped to one vertex. The generated graph is tested for cycles until a suitable graph is found.

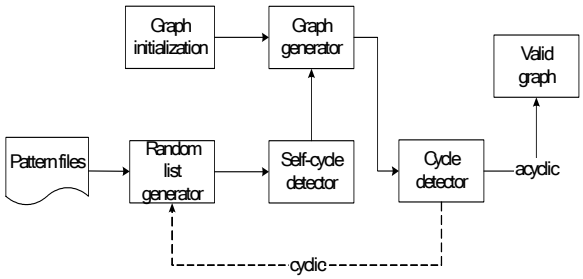

Fig.23 map model in hash signature

Map process needs testing the generated graph. Because the hash signature proposed is minimal perfect hash signature, the key of a pattern responds to its index uniquely, which needs the construction of graph to be standard. If a self-cycle is detected, there isn't an edge associated with two vertices which isn't a standard relation between vertices and edges, and this will result in failure of graph generation. At the same time ,if a cycle is detected after graph generation, there may be many duplicate keys, which is also may be the weak point of random process that makes the vertices generated from different keys are the same. The settlement to duplicates is delete one and leave one, because the characteristic of different functions may be the same in the whole process, this is because the there are many library functions whose body are very alike to each other, especially when the two functions act as the same function. Now this paper applies blur strategy, that functions act the same are identified as the same function. The settlement to the same vertex generated from two unique keys is remap until a sound graph is generated. This iteration will continue forever? No, this will be proved later.

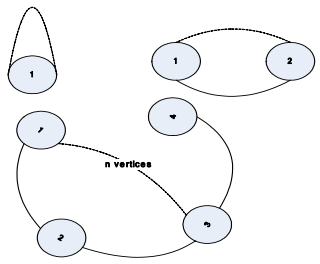

Fig.24 the cycle phenomenon in remap process

The assignment is a process that constructs an one to one association between vertices and keys. This paper proposed an arithmetic that a unique index is obtained given a special key. So the assignment to vertices should make the generation of edges unique. This arithmetic firstly selects an vertex and assign to 0, then obtain the other vertex value according to the edge linked, finally the unique index to edges are reached.

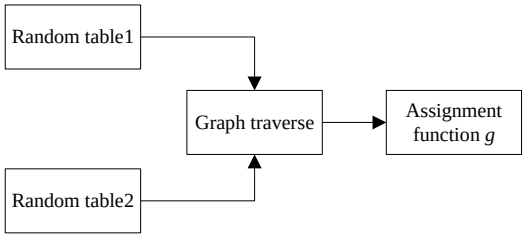

Fig.25 assignment model in hash signature

The minimal perfect hash function requires every edge and two vertices associated with the edge are mapped one to one, but how to fulfill this objection? If the corresponding edge is generated from random values of vertices, the edges generated from different vertices may be the same in some arbitrary times. This paper fulfills the uniqueness of index by obtaining "g" value of every vertex.

D. Match process

Hash signature match process firstly selects the signature library. The usual methods of selection are as follows: 1 the library of operation system is known and the signature is generated from it. 2 the information of library file and compiler is get by recognizing dynamic library. 3 the signature of start code in executables are extracted during disassembling process to obtain the library file information of different versions, which will recognize signature library. The first method is suitable for the identification for executables in known system; the second method is suitable for executables when dynamic library functions is easy to identified and the related information is very sufficient; IDA applies the third method to fulfill the automation of identification process.

Hash signature match process firstly treats with all the pattern files of candidate functions, then tests on the signature library to know if the pattern exists in the library to fulfill the effect of identification. The detailed process is analyze files to get the key of files, then look in the random list generated from alphabet, obtain two random values of key as vertices of graph, get the index of edge after the generation of vertices, finally get the key of corresponding edge and related information.

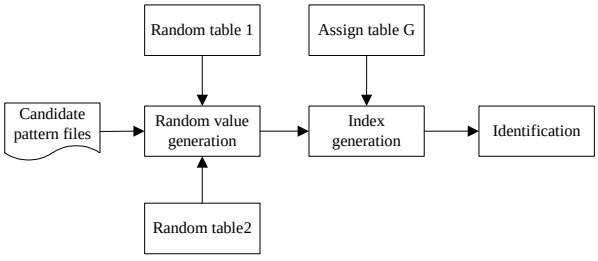

Fig.26 match process in hash signature

The key of match process is find the value of two vertices according to assign table "g", and the key of success in the process is that the graph is acyclic. It's obvious that the assignment of every vertex is once in acyclic graph, which ensures that the index of corresponding edge associated with two vertices is unique.

Supposed "M" is the set of pattern files, "S" is the signature of pattern files, "x" is function modules to be recognized, "y" is recognized library function, "K" is the set of recognized library:

$$K = \{x \mid h(x) \in S\}$$
$$y = \{y \mid y \in M, h(y) = h(x)\}$$

V. ARITHMETIC ANALYSIS

This section mainly talks about complex analysis in generation of hash signature, supposed the number of keys is "m", the number of edges is "n":

First the map section is analyzed. Keys of all the functions should be mapped to the graph during map section, so the generation of "m" edges and "n" vertices should be estimated during map process. During the whole analysis process, the times of random calculation, the size of alphabet and the length of key are all under consideration. The time cost mainly include: generation of random table; the calculation and test of every key; the test of acyclic graph. The time needed by generation of random table is the size of alphabet times the length of key, which is constant. The time complex of test for keys is O(the number of keys); the time complex of test for cycles is O(the number of edges + the number of vertices). So the time complex of map for graph is O(m+n).

Then the assignment is considered. The assignment mainly traverse the whole graph to generate "g" function, so the time complex is O(m+n).

There exists iterations in map step, and the time of iteration is related to the number of vertices in graph. Supposed a graph that with m edges and n vertices, the probability of generating this graph is p, and X is equal to the expected number of iterations. So $P(X=i) = p*(1-p)^{i-1}$, and the mean of X is 1/p; the variance of X is $(1-p)/p^2$; the probability that the number of iteration in map step exceeds k is$(1-p)^k$. The probability that generates an acyclic graph is closely in touch with the size of n, supposed n=cm, c is constant, when n -> ∞, then the number of cycle that own k edges may exceeds 2k/(2kck) [10], this formula is suitable for graph with self-cycle or multi-edges. Then the probability that generates acyclic graph is $\exp(-\sum_{k=1}^{n} 2^k / (2kc^k))$, so when c>2, the probability of acyclic graph is $\sqrt{\frac{c-2}{c}}$; when c<=2, the probability of acyclic graph is 0. So the time complex of this arithmetic is O(m+n). When n=cm, the time complex is O(m). At the same time, the probability that generates an acyclic graph is $\sqrt{\frac{c-2}{c}}$, the mean of the number that iteration takes place is $\sqrt{\frac{c}{c-2}}$ [11].

VI. TEST AND CONCLUSION

Based on matching strategy proposed by this paper, two object program exam1 and exam2 are tested, the result is as follows:

TABLE I. TEST RESULT

Program \ Number	Static library function recognized	Program's static library function
exam1	310	325
exam2	524	541

The test result proved that, static library function identification algorithm proposed by this paper can recognize library function for ARM microprocessor effectively, and solve the problem of functions with the same name. Static library function recognition is based on IDA, and has a good effect on executable programs for ARM embedded microprocessor, whose deficiency is weak recognition for code confusion.

REFERENCES

[1] Chen Fuan, Liu Zongtian. C function recognition technique and its implementation in 8086 C decompiling system. Mini-Micro Computer System. 12,11(1991). Written in Chinese.

[2] Xu Xiang Yang, Lei Tao, Zhu Hong. A Study of Static Library Functions Recognition in Decompiling. Computer engineering and application. 2004,09. Written in Chinese.

[3] *The IDA Pro Book* (C) 2008 by Chris Eagle.chapter 12.

[4] Zhou Rui Ping, Lei Tao, Zhu Hong. Applied Study of Library Functions Recognition in Decompiling. Computer application and research. 2004. Written in Chinese.

[5] Mike Van Emmerik. Identifying Library Functions in Executable File Using Patterns. Software Engineering Conference, 1998.

[6] Eelco Visser . Stategic Pattern Matching. Department of computer science university Utrecht.

[7] Ilfak Guilfanov. Fast Library Identification and Recognition Technology.

[8] Hu Zheng, Chen Kaiming. Study of the Library Functions Recognition in C++ Decompiler. Computer engineering and application. 2006.03. Written in Chinese.

[9] C. Cifuentes. Reverse Compilation Techniques.PhD dissertation. Queensland University of Technology, School of Computing Science, July 1994.

[10] P. Erdos and A. Renyi. On the evolution of random graphs. Publ. Math.Inst. Hung. Acad. Sci., 5:17-61, 1960. Reprinted in J.H. Spencer, editor,The Art of Counting: Selected Writings, Mathematicians of Our Time, pages 574-617. Cambridge, Mass.: MIT Press, 1973.

[11] Z.J. Czech, G. Havas and B.S. Majewski. An optimal algorithm for generating minimal perfect hash functions, Information Processing Letters, 43(5):257-264, October 1992. Also Technical Report TR0217, Key Centre for Software Technology, Department of Computer Science, University of Queensland 4072 Australia.