Zhang *et al. EURASIP Journal on Information Security* (2016) 2016:14
DOI 10.1186/s13635-016-0038-z

EURASIP Journal on
Information Security

RESEARCH Open Access

An adaptive approach for Linux memory analysis based on kernel code reconstruction

Shuhui Zhang[1,2*], Xiangxu Meng[1] and Lianhai Wang[2]

Abstract

Memory forensics plays an important role in security and forensic investigations. Hence, numerous studies have investigated Windows memory forensics, and considerable progress has been made. In contrast, research on Linux memory forensics is relatively sparse, and the current knowledge does not meet the requirements of forensic investigators. Existing solutions are not especially sophisticated, and their complicated operation and limited treatment range are unsatisfactory. This paper describes an adaptive approach for Linux memory analysis that can automatically identify the kernel version and recovery symbol information from an image. In particular, given a memory image or a memory snapshot without any additional information, the proposed technique can automatically reconstruct the kernel code, identify the kernel version, recover symbol table files, and extract live system information. Experimental results indicate that our method runs satisfactorily across a wide range of operating system versions.

Keywords: Memory forensics, Linux memory analysis, Kernel symbol

1 Introduction

The physical memory of a computer is highly useful but can be a challenging resource for the collection of digital evidence. Physical memory may first appear to be a large, amorphous, and unstructured collection of data. In fact, by examining a memory image, we can extract details of volatile data, such as running processes, logged-in users, current network connections, users' sessions, drivers, and open files. Although criminals tend to avoid leaving any evidence in a computer's persistent storage, it is extremely hard for them to completely remove their footprints from the memory. In some cases, physical memory is the only place where evidence can be found. In a computer operating system (OS) that boots and runs completely from CD-ROM, nearly all of the valuable information exists in the physical memory of the computer. Therefore, memory forensics is becoming increasingly important.

Before 2005, the physical memory of a computer was mainly captured to retrieve strings, e.g., passwords, credit card numbers, fragments of chat conversations, IP addresses, or email addresses. In 2005, the Digital Forensics Research Workshop (DFRWS) organized a memory analysis challenge [1]. Since then, the capture and analysis of the content of physical memory, known as memory forensics, has become an area of intense research and experimentation [2]. Numerous studies have analyzed Windows memory images. The MemParser tool enables an examiner to load a physical memory dump of certain Windows systems, reconstruct process information, and extract data relating to specific processes [3]. PTFinder is a proof-of-concept implementation with the ability to reveal hidden and terminated processes and threads [4]. A method based on the Kernel Processor Control Region (KPCR) structure in Windows was proposed to determine the OS version and realize the translation from virtual address to physical address [5]. Moreover, technical details related to memory analysis have been discussed, including address translation [6, 7], pool allocation [8], swap integration [6], carving out memory [9, 10], sensitive information extraction [11, 12], and malicious code detection [13]. In short, memory analysis has been used in the wider context of digital forensics, virtual machine introspection, and malware detection [14].

Compared with Windows memory forensics, memory analysis of Linux systems presents some practical challenges. Current Linux memory analysis technologies

---

*Correspondence: zhangshh@sdas.org
[1] School of Computer Science and Technology, Shandong University, Shunhua Road, Jinan, China
[2] Shandong Computer Science Center (National Supercomputer Center in Jinan), Shandong Provincial Key Laboratory of Computer Networks, Keyuan Road, 250014 Jinan, China

require precise knowledge of the OS edition and kernel symbol information, which is generated at compile time. Kernel symbol information varies with the different OS editions. Furthermore, the Linux kernel is highly configurable. During the kernel build process, users may specify a large number of different options through the kernel's configuration system. These options affect the kernel symbol information, resulting in distinct key structures for the same OS version. To obtain kernel symbol information, one must configure an environment in which the OS version and configuration options are exactly the same as those in the target system. For incident response applications, obtaining precise and relevant information is currently a slow, manual process, which limits its usefulness in rapid triaging.

To overcome these problems, this paper describes techniques that allow for the automatic adaptation of memory analysis tools for a wide range of kernel versions. Using dynamic reconstruction of the kernel code, it is possible to identify the OS version, disassemble correlative functions, and acquire kernel symbol information. Some other memory analysis systems rely on information that may not be available, whereas the proposed system only needs the analyzed memory dump. The main contributions of this paper are as follows:

- We present a multi-aspect approach to automatically identify the precise kernel version when provided with only a physical memory dump. The approach is universal, and does not rely on any prior knowledge for particular OSs.
- We devise a set of novel techniques to obtain kernel symbols from the physical memory dump instead of obtaining symbol information from the target kernel's "System.map" file. Each time a new kernel is compiled, various symbols are assigned different addresses. New kernel versions of Linux are released frequently, and it is inconvenient to find all System.map files.
- As the symbols in the System.map file are important, and symbols exported from modules are critical for investigators, a method of parsing symbols exported from modules is presented. To recover and analyze loaded kernel module information, it is essential to understand the relevant data structures used by the target OS. As the inclusion or exclusion of a kernel configuration option can cause the insertion or removal of several members of key structures, analysis methods that rely on a stable key structure layout are inadvisable. A method to accurately and dynamically build representative data structures is also presented.

Based on the above techniques, we develop a new Linux memory analysis system named RAMAnalyzer that can identify the OS version and acquire symbol information automatically. Live system information can subsequently be retrieved. We examine the performance of RAMAnalyzer on various recent Linux kernels, and show that it is an adaptive solution for the Linux memory analysis problem.

The remainder of this paper is organized as follows. Section 2 introduces some background information. The proposed techniques based on dynamic reconstruction are described thoroughly in Sections 3 and 4. In Section 5, we evaluate the proposed forensics tool in terms of effectiveness and performance. The final section summarizes this study and states our conclusions and indicates some opportunities for future research in this area.

2 Background and related work

2.1 Problem statement

The main problem encountered by memory analysis tools when parsing the Linux kernel memory is the need for prior knowledge of the precise kernel version and symbol information. In an incident response and live analysis context, this prior knowledge may not always be obtainable. We assume the following scenarios:

- The specific target kernel version is unknown.
- The kernel version is known but neither the System.map file nor */proc/kallsyms* information of the target system is available.

Under these scenarios, our system has three major goals: precision, efficiency, and generality.

- Precision. The OS family and precise version are both required. For instance, given a Linux kernel, we need to know not only its major version (e.g., 2.6 or 3.10), but also its minor version because the symbol's information and data structures of various Linux kernels are different.
- Efficiency. It is necessary to automatically obtain information for the OS version and symbols within a short period of time.
- Generality. The system should be adaptive and analyze the mainstream Linux kernel memory image, rather than support only certain versions of Linux.

2.2 Kernel symbols

In the Linux kernel 2.6.×, *kallsyms* is used to extract all the non-stack symbols from a kernel and build a data blob. CONFIG_KALLSYMS should be configured as follows:

```
make menuconfig
General setup —>
  [*] Configure standard kernel features (for small
systems) —>
    [*] Load all symbols for debugging/ksymoops
    [*] Include all symbols in kallsyms
    [*] Do an extra kallsyms pass
```

In the last stage of the kernel compile, the following command is executed:

```
nm -n vmlinux|scripts/kallsyms
```

Therefore, all the kernel symbols are generated and sorted according to their addresses. This list is used to create the "kallsyms.S" file, which includes several special symbols: *kallsyms_addresses*, *kallsyms_num_syms*, *kallsyms_names*, *kallsyms_makers*, *kallsyms_token_table*, and *kallsyms_token_index*. Among these symbols, *kallsyms_addresses* points to the addresses of all kernel symbols in order, *kallsyms_num_syms* points to the "num" value of kernel symbols, and *kallsyms_names* corresponds to the symbols' name arrays. For convenience, *kallsyms_markers*, *kallsyms_token_table*, and *kallsyms_token_index* are used for the offset index and high-frequency string compression.

The acquisition of kernel symbols is essential for analyzing the information contained within a physical memory dump. For example, if system calls are needed during an investigation, their addresses are stored in a kernel structure called the system call table. The *sys_call_table* symbol stores an address for this table, and may be used to enumerate the addresses of system calls. There are several ways to obtain the symbols:

- Copy */proc/kallsyms* or System.map and analyze the file [15, 16]. Care should be taken when copying the System.map file because systems with multiple kernels have multiple System.map files. Unlike */boot/System.map*, */proc/kallsyms* is a "proc file" that is created when a kernel boots up. This is not actually a disk file and is always correct for the kernel that is currently running. Furthermore, */proc/kallsyms* contains not only kernel symbols but also symbols exported from modules.
- Additionally, the kernel build system puts the System.map inside the kernel's executable and linkable format (ELF) executable. Symbols can be extracted using the following commands:

```
$ ./scripts/extract-vmlinux
/tmp/vmlinuz-3.13.0-63-generic >
/tmp/vmlinuz-3.13.0-63-generic.elf
$ readelf -Wa /tmp/vmlinuz-3.13.0-63-generic.elf
$ objcopy -j _ksymtab_strings -O binary
/tmp/vmlinuz-3.13.0-63-generic.elf
vmlinux.bin-_ksymtab_strings
```

Even a simple recompile of the same kernel is sufficient to change the symbol addresses. In previous solutions for obtaining symbol tables, methods that select symbol table profiles according to the kernel versions are obviously inaccurate. Furthermore, there is a strong need to reliably determine the correct profile for unknown kernels, which are often encountered during incident response situations [17].

2.3 Linux memory analysis

In this section, we survey some related studies on Linux memory analysis. Assuming that the kernel data structures are known, a modular, extensible framework named FATKit can realize general virtual address space reconstruction and visualization [18]. The open source volatility framework has been adapted to work with Linux memory dumps, including Android, but it must be configured for the specific version of Linux being examined [15]. SecondLook is a commercial application with a GUI and command-line interface that can extract and display memory structures including processes, loaded kernel modules, and system call tables [19]. RAMPARSER was designed to reconstruct kernel data structures such as *task_struct*, *mm_struct*, *File*, *Dentry*, *Qstr*, *inet_sock*, *Sock*, and *Socket* [20]. Linux kernel versions from 2.6.9 to 2.6.27 were tested to verify its feasibility.

Each memory forensics solution has different features along with several limitations: first, the accurate OS version of a memory image must be known in advance, which means that Linux memory images without precise OS version information cannot be parsed correctly. Second, the analyzed system's System.map file and kernel information are needed. These data may not be immediately available, or may have been modified by attackers to thwart forensic analysis. For example, the Kernel Debugger Block can be easily overwritten by malware [21]. Furthermore, some tools only work on specific versions and require substantial manual intervention.

3 Linux memory analysis framework based on kernel code reconstruction

In this paper, based on kernel code reconstruction, we propose a new Linux memory analysis framework that can automatically detect the kernel version and recover the symbol table file from the memory image. As shown in Fig. 1, there are five key components in our framework:

- Kernel version identification (KVI): this component allows the OS version to be detected in two ways: *linux_banner* content identification and *vmcoreinfo_data* content identification.
- kallsyms location symbol values recovery (KLSR): the symbol table file can be recovered from memory using kallsyms location symbols such as *kallsyms_addresses*, *kallsyms_num_syms*, *kallsyms_names*, *kallsyms_token_table*, and *kallsyms_token_index*. Based on kernel code reconstruction, this component provides a method for discovering the above symbol values.

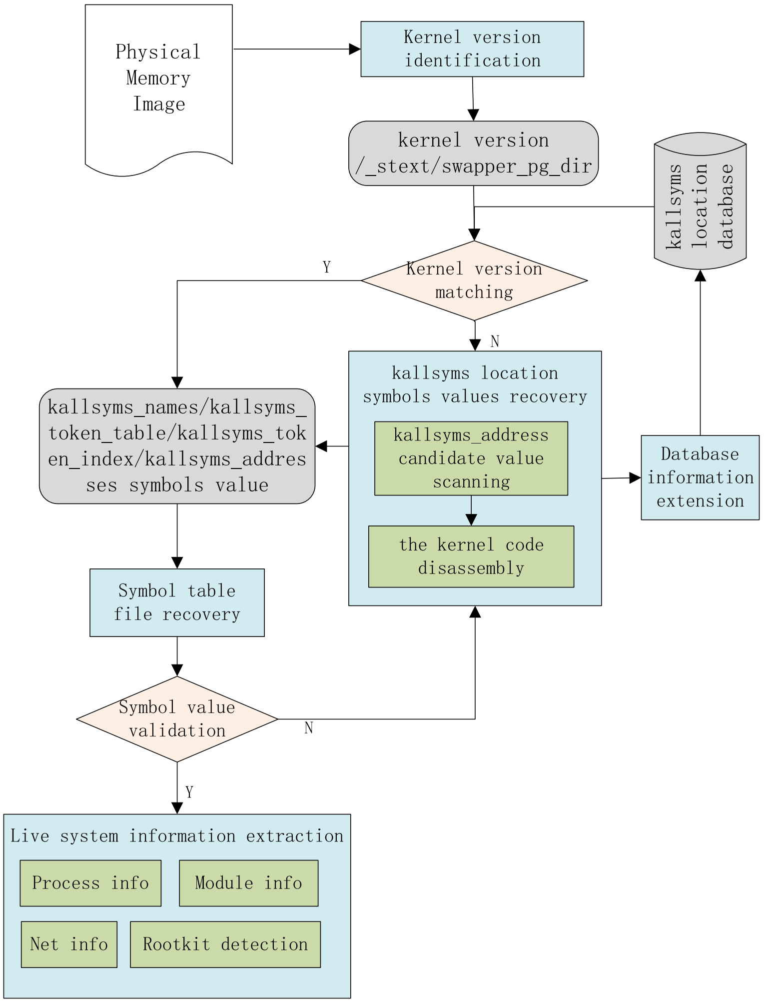


Fig. 1 Overview of RAMAnalyzer

- Symbol table file recovery (STFR): using the kallsyms location symbol values obtained from KLSR, the symbol table file content and kernel symbol information can be recovered.
- Live system information extraction (LSIE): several key kernel symbols are selected to extract live system information, such as process information and module information. Furthermore, the symbols exported from modules can be parsed.
- Database information extension: the addresses of the symbols are identical for identical kernel versions and compile configurations. To improve system efficiency, a database records symbol information for known kernel versions. When the kallsyms location symbol values of new versions are recovered from KLSR, these values are saved in the database.

The detailed flow of our algorithm is described as follows:

Step I: given a physical memory image, identify the precise kernel version of the target system. Using the KVI module, the kernel version and the values of *_stext* and *swapper_pg_dir* can be obtained.
Step II: check the database for preexisting symbol information. If the processed kernel version exists, extract the symbol addresses from the database and go to step IV; otherwise, go to step III.
Step III: recover kallsyms location symbol values using the KLSR module. New acquisition data are recorded in the database.
Step IV: after the acquisition of kallsyms location symbol values, the symbol table file is recovered using the STFR module.
Step V: extract the *_stext* symbol from the symbol table file and compare its value with that obtained from step I. If these two values are equal, the kallsyms location symbol values are correct; go to step VI. Otherwise, go to step III, adjust the *kallsyms_address*

candidate value and retrieve the kallsyms location symbol value again.

Step VI: using the kernel symbols in the symbol table file, live system information can be extracted. In particular, symbols exported by loaded kernel modules are analyzed.

From the above description, we can see that our solution has an adaptive ability to cope with different kernel versions. More details are introduced in the next section.

4 Research methodology

In this section, we describe the detailed processes of kernel version identification, kallsyms location symbol values recovery, symbol table file recovery, and live system information extraction.

4.1 Kernel version identification

There are two ways to obtain kernel version information from the memory image: *vmcoreinfo_data* content identification and *linux_banner* content identification.

4.1.1 linux_banner content identification

The *start_kernel()* function, which is called by the *startup_32()* function, initializes all of the data structures needed by the kernel, enables interrupts, and creates another kernel thread named process 1. Finally, *linux_banner* information is printed in the following format:

const char ***linux_banner[]*** *=*
“Linux version ” UTS_RELEASE “ (" LINUX_ COMPILE_BY “@”
LINUX_COMPILE_HOST “) (”LINUX_COMPILER “) ”UTS_VERSION “\n”;

By searching for the characteristic character “Linux version”, kernel version information can be obtained.

4.1.2 vmcoreinfo_data Content Identification

In the system initialization phase, the *crash_save_vmcoreinfo_init()* function is triggered to initialize the content of *vmcoreinfo_data*, which includes general crash kernel information such as the kernel version, page size, and symbol information. *vmcoreinfo_data* starts with the character string *“OSRELEASE=”*; the character strings *“SYMBOL(swapper_pg_dir)=”* and *“SYMBOL(_stext)=”* are also included. By searching for these three strings, the address of *vmcoreinfo_data* can be located. The partial content of *vmcoreinfo_data* in the memory image is shown in Fig. 2.

The kernel version in *linux_banner* and *vmcoreinfo_data* content should be the same. The latter contains information about the *_stext*, *swapper_pg_dir*, *vmlist*, *mem_map*, and *init_uts_ns symbols*, which are also stored in the symbol table files. The values of these symbols are virtual addresses. Generally, if *swapper_pg_dir* has a length value of 8, the OS is 32 bit and the physical address of *swapper_pg_dir* is its virtual address minus 0×c0000000. If *swapper_pg_dir* has a length value of 16, the OS is 64 bit and its physical address is its virtual address minus 0×ffffffff80000000. *swapper_pg_dir* is the page global directory (pdg) for a process named *“swapper”* and can be used to translate between the virtual address and the physical address in the kernel address space. This symbol name differs between architectures in the symbol table file, being called *swapper_pg_dir* on both ×86 and PPC64, but it is named *init_level4_pgt* on ×86_64.

4.2 Kallsyms location symbol values recovery

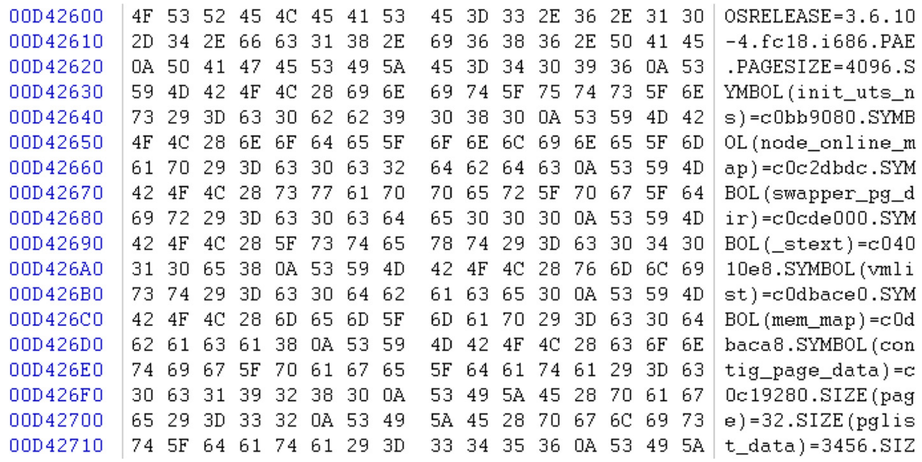
The algorithm for the kallsyms location symbol values recovery has four main steps:

Step I: Kallsyms_address candidate value scanning. Because *_stext* is one of the kernel symbols, the values obtained from the procedure described in Section 4.1.2 can be used to locate the *kallsyms_addresses.* During the search procedure, the value of *_stext* may be found in multiple places. To enhance the efficiency of the algorithm, we impose some restrictions. For 32-bit systems, for example, the content before and after the found address are the addresses of kernel symbols, and so the values should be greater than 0xc0000000. Tracing back from the found address, we can obtain the value of the *startup_32* symbol, which can be calculated from *_stext*&0×ffff0000. The first symbol in */proc/kallsyms* is generally *startup_32*, and this is where the address of the *startup_32* symbol resides. This provides a candidate physical address for *kallsyms_addresses.* However, in 64-bit systems, there are several symbols before the *startup_64* symbol, and so the test times are higher than for 32-bit systems.

Step II: The kernel code disassembly. To obtain the other four symbol values, the kernel code in memory must be disassembled correctly. Unfortunately, because the instructions for different systems and architectures are of various lengths, starting from the wrong instruction location will disassemble a completely different instruction sequence. To address this challenge, we decompile the smallest amount of kernel code, instead of the whole block of required function calls.

Analyzing the source code of a Linux kernel, it is clear that operations related to the kernel symbols are mainly present in Linux/kernel/kallsyms.c. The symbols that will be re-linked against their real values during the second link stage are defined below:

extern const unsigned long kallsyms_addresses[] _weak;
extern const u8 kallsyms_names[] _weak;
extern const unsigned long kallsyms_num_syms;



```
00D42600  4F 53 52 45 4C 45 41 53  45 3D 33 2E 36 2E 31 30  OSRELEASE=3.6.10
00D42610  2D 34 2E 66 63 31 38 2E  69 36 38 36 2E 50 41 45  -4.fc18.i686.PAE
00D42620  0A 50 41 47 45 53 49 5A  45 3D 34 30 39 36 0A 53  .PAGESIZE=4096.S
00D42630  59 4D 42 4F 4C 28 69 6E  69 74 5F 75 74 73 5F 6E  YMBOL(init_uts_n
00D42640  73 29 3D 63 30 62 62 39  30 38 30 0A 53 59 4D 42  s)=c0bb9080.SYMB
00D42650  4F 4C 28 6E 6F 64 65 5F  6F 6E 6C 69 6E 65 5F 6D  OL(node_online_m
00D42660  61 70 29 3D 63 30 63 32  64 62 64 63 0A 53 59 4D  ap)=c0c2dbdc.SYM
00D42670  42 4F 4C 28 73 77 61 70  70 65 72 5F 70 67 5F 64  BOL(swapper_pg_d
00D42680  69 72 29 3D 63 30 63 64  65 30 30 30 0A 53 59 4D  ir)=c0cde000.SYM
00D42690  42 4F 4C 28 5F 73 74 65  78 74 29 3D 63 30 34 30  BOL(_stext)=c040
00D426A0  31 30 65 38 0A 53 59 4D  42 4F 4C 28 76 6D 6C 69  10e8.SYMBOL(vmli
00D426B0  73 74 29 3D 63 30 64 62  61 63 65 30 0A 53 59 4D  st)=c0dbace0.SYM
00D426C0  42 4F 4C 28 6D 65 6D 5F  6D 61 70 29 3D 63 30 64  BOL(mem_map)=c0d
00D426D0  62 61 63 61 38 0A 53 59  4D 42 4F 4C 28 63 6F 6E  baca8.SYMBOL(con
00D426E0  74 69 67 5F 70 61 67 65  5F 64 61 74 61 29 3D 63  tig_page_data)=c
00D426F0  30 63 31 39 32 38 30 0A  53 49 5A 45 28 70 61 67  0c19280.SIZE(pag
00D42700  65 29 3D 33 32 0A 53 49  5A 45 28 70 67 6C 69 73  e)=32.SIZE(pglis
00D42710  74 5F 64 61 74 61 29 3D  33 34 35 36 0A 53 49 5A  t_data)=3456.SIZ
```

Fig. 2 Partial content of vmcoreinfo_data

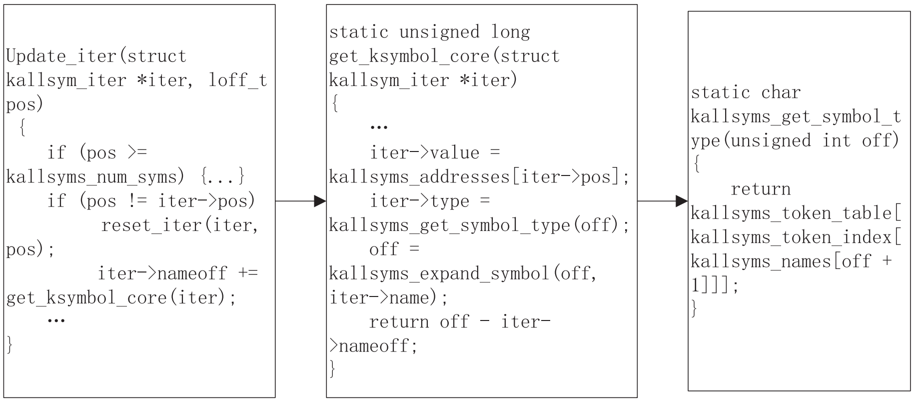
extern const u8 kallsyms_token_table[] _weak;
extern const u16 kallsyms_token_index[] _weak;

The call relationship of the *update_iter* function is described in Fig. 3. In the *update_iter* function, the *get_ksymbol_core* function is called using *kallsyms_addresses*. Next to the instruction *"iter->value = kallsyms_addresses[iter->pos],"* the *kallsyms_get_symbol_type* function is called using *kallsyms_token_table*, *kallsyms_token_index*, and *kallsyms_names[off + 1]*.

Once the principle of the *update_iter* function is fully understood, we can use the candidate value of *kallsyms_addresses* obtained from step I to obtain the values of the other four symbols.

An image from the 3.6.10-4.fc18.i686.PAE system is used to illustrate the method. The value of the *kallsyms_addresses* symbol is found at offset 0xc4 for the *update_iter* function's binary code in the image. Therefore, we step back three bytes and disassemble the binary code. As mentioned above, our principle is to reduce the amount of disassembled code by as much as possible and improve the precision of our method. Approximately 0×2a bytes are chosen to be decompiled, and the results are as follows:

```
4aadc1: 8b 04 95 0c 8c 9e c0          mov 0xc09e8c0c
                                       (%edx, 4), %eax
4aadc8: 8d 56 11                  lea 0x11(%esi), %edx
4aadcb: C6 86 91 00 00 00 00  movb $0x0, 0x91 (%esi)
4aadd2: 89 46 08                     mov %eax, 0x8(%esi)
4aadd5: 0f b6 83 b5 47 a2 c0      movzbl 0xc0a247b5
                                       (%ebx), %eax
4aaddc: 0f b7 84 00 a0 95 ad c0   movzwl 0xc0ad95a0,
                                        %eax
4aade4: 0f b6 80 10 92 ad c0      movzbl 0xc0ad9210
                                       (%eax), %eax
```

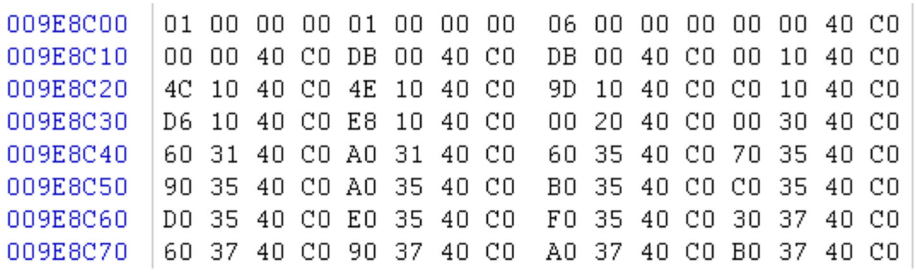


```
Update_iter(struct
kallsym_iter *iter, loff_t
pos)
 {
    if (pos >=
kallsyms_num_syms) {...}
    if (pos != iter->pos)
          reset_iter(iter,
pos);
         iter->nameoff +=
get_ksymbol_core(iter);
    …
}
```

```
static unsigned long
get_ksymbol_core(struct
kallsym_iter *iter)
{
    …
    iter->value =
kallsyms_addresses[iter->pos];
    iter->type =
kallsyms_get_symbol_type(off);
    off =
kallsyms_expand_symbol(off,
iter->name);
    return off - iter-
>nameoff;
}
```

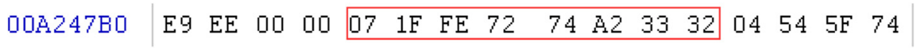
```
static char
kallsyms_get_symbol_t
ype(unsigned int off)
{
    return
kallsyms_token_table[
kallsyms_token_index[
kallsyms_names[off +
1]]];
}
```

Fig. 3 Call relationship of update_iter function



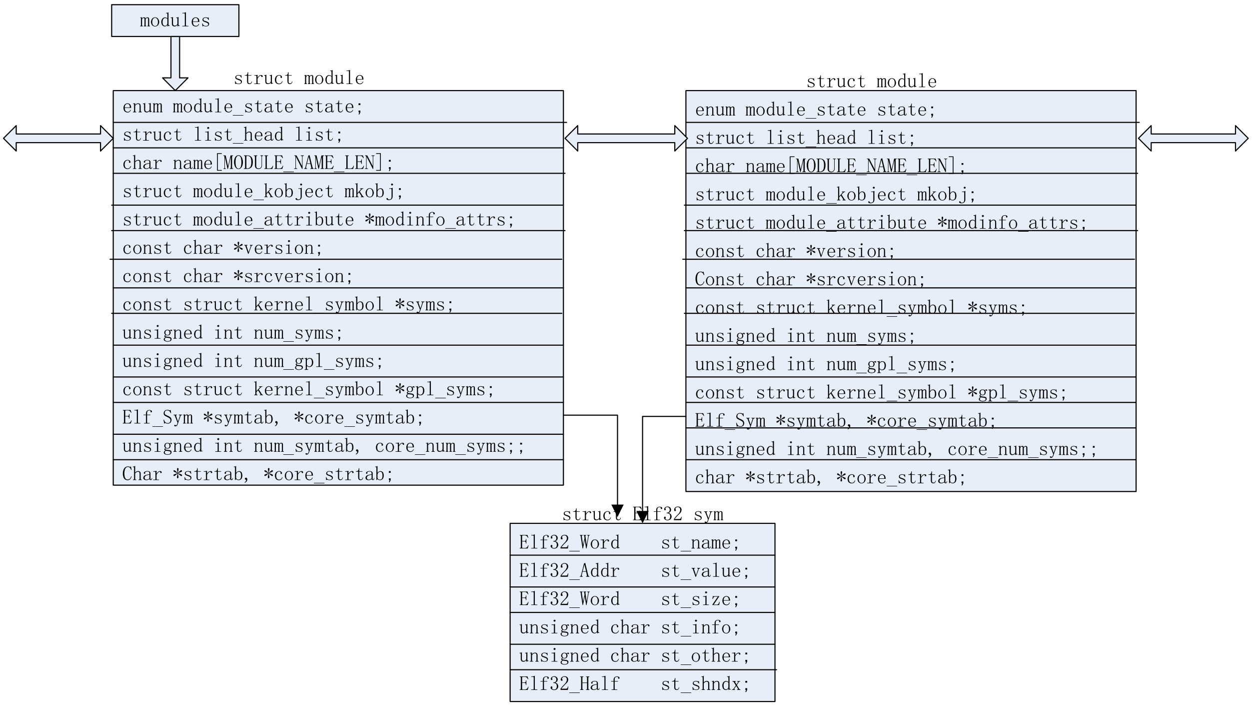
```
009E8C00  01 00 00 00 01 00 00 00  06 00 00 00 00 00 40 C0
009E8C10  00 00 40 C0 DB 00 40 C0  DB 00 40 C0 00 10 40 C0
009E8C20  4C 10 40 C0 4E 10 40 C0  9D 10 40 C0 C0 10 40 C0
009E8C30  D6 10 40 C0 E8 10 40 C0  00 20 40 C0 00 30 40 C0
009E8C40  60 31 40 C0 A0 31 40 C0  60 35 40 C0 70 35 40 C0
009E8C50  90 35 40 C0 A0 35 40 C0  B0 35 40 C0 C0 35 40 C0
009E8C60  D0 35 40 C0 E0 35 40 C0  F0 35 40 C0 30 37 40 C0
009E8C70  60 37 40 C0 90 37 40 C0  A0 37 40 C0 B0 37 40 C0
```

Fig. 4 Partial content of kallsyms_address



```
00A247B0  E9 EE 00 00 07 1F FE 72  74 A2 33 32 04 54 5F 74
```

Fig. 5 Partial content of kallsyms_names

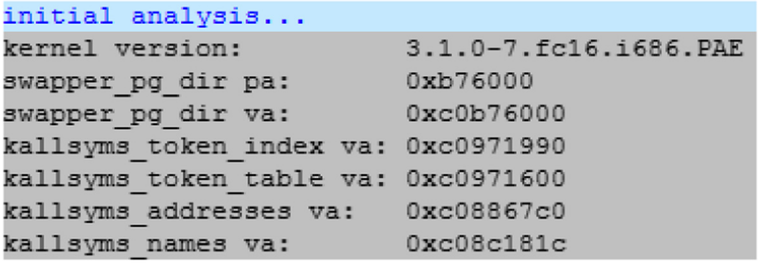


```
enum module_state state;
struct list_head list;
char name[MODULE_NAME_LEN];
struct module_kobject mkobj;
struct module_attribute *modinfo_attrs;
const char *version;
const char *srcversion;
const struct kernel_symbol *syms;
unsigned int num_syms;
unsigned int num_gpl_syms;
const struct kernel_symbol *gpl_syms;
Elf_Sym *symtab, *core_symtab;
unsigned int num_symtab, core_num_syms;;
Char *strtab, *core_strtab;
```

```
enum module_state state;
struct list_head list;
char name[MODULE_NAME_LEN];
struct module_kobject mkobj;
struct module_attribute *modinfo_attrs;
const char *version;
Const char *srcversion;
const struct kernel_symbol *syms;
unsigned int num_syms;
unsigned int num_gpl_syms;
const struct kernel_symbol *gpl_syms;
Elf_Sym *symtab, *core_symtab;
unsigned int num_symtab, core_num_syms;;
char *strtab, *core_strtab;
```

```
Elf32_Word     st_name;
Elf32_Addr     st_value;
Elf32_Word     st_size;
unsigned char  st_info;
unsigned char  st_other;
Elf32_Half     st_shndx;
```

Fig. 6 Main members of module structure



```
initial analysis...
kernel version:          3.1.0-7.fc16.i686.PAE
swapper_pg_dir pa:       0xb76000
swapper_pg_dir va:       0xc0b76000
kallsyms_token_index va: 0xc0971990
kallsyms_token_table va: 0xc0971600
kallsyms_addresses va:   0xc08867c0
kallsyms_names va:       0xc08c181c
```

Fig. 7 Initialization result

Through a combined analysis of the disassembled output and the update_iter function's source code, the values of *kallsyms_names*, *kallsyms_token_index*, and *kallsyms_token_table* can be obtained. In 32-bit systems, the value of *kallsyms_num_syms* is the value of *kallsyms_names* minus 4.

There are some differences in 64-bit systems. Bits 0–31 of the symbol addresses are used, and the value of *kallsyms_num_syms* is the value of *kallsyms_names* minus 8. Although some changes take place in the binary code of the *update_iter* function for different Linux systems, the differences in the code segment used here are minimal.

4.3 Symbol table file recovery

After acquiring the kallsyms location symbols, the STFR method proceeds as follows:

The *kallsyms_num_syms* symbol points to the kernel symbol "num" in */proc/kallsyms*. The *kallsyms_addresses* symbol points to the addresses of all kernel symbols in order. Each symbol address has a length of 4 in 32-bit systems and 8 in 64-bit systems. To obtain the corresponding name, the *kallsyms_names*, *kallsyms_token_table*, and *kallsyms_token_index* symbols are needed. The *kallsyms_names* symbol points to a list of length-prefixed byte arrays that encode indexes into the token table. According to the length, the bytes of each array are acquired and used to construct a substring. Finally, the substrings are joined together to form the type and name of the required symbol.

We again use an image from the 3.6.10-4.fc18.i686.PAE system to describe the above method.

The addresses of the required symbols are determined from the database:

c09e8c0c R kallsyms_addresses
c0a247b0 R kallsyms_num_syms
c0a247b4 R kallsyms_names
c0ad9210 R kallsyms_token_table
c0ad95a0 R kallsyms_token_index

First, translate the virtual address of the *kallsyms_num_syms* symbol to a physical address and obtain the num of kernel symbols in */proc/kallsyms*. Likewise, convert the virtual address of the *kallsyms_addresses* address and read the addresses of all kernel symbols in order. The partial content of *kallsyms_addresses* is displayed in Fig. 4.

In the next step, the content pointed to by the *kallsyms_names* symbol is read and split into substrings according to its format. As shown in Fig. 5, the first byte of each substring is the length of the compression bytes. Each compression byte corresponds to several characters through conversion with the *kallsyms_token_table* and *kallsyms_token_index* symbols. Connecting all of the characters together, we obtain the type and name of the symbol. By dealing with the bytes marked in Fig. 5, the type for the first symbol is determined to be T, which means that the symbol is in the text (code) section and the name of the first symbol is *startup_32*. In combination with the result from the *kallsyms_address* symbol, the address of the *startup_32* symbol is 0×c0400000.

To verify the correctness of the symbol values, the value of *_stext* obtained from the process described in this section is compared with that obtained in Section 4.1.2. If the two values are the same, the obtained symbols are available. If not, we recover the symbols using the algorithm described in Section 4.2.

4.4 Live system information extraction

After determining the kernel symbols, several key symbols are selected to obtain the live system information.

4.4.1 Gathering offsets of structure members

The structure layouts vary greatly depending upon the configuration parameters. For example, the layout of the module structure depends on the values of optional configuration parameters such as CONFIG_MODULE_SIG, CONFIG_SYSFS, and CONFIG_UNUSED_SYMBOLS. Thus, to properly analyze a Linux image, the offsets of important structure members must be identified. As shown in Fig. 6, the module structure plays a significant role in the extraction of module information.

Code fragments 1–3 show how equivalent statements can be compiled to form radically different instruction sequences. The C source code in code fragment 1 is from the *module_get_kallsym()* function within */kernel/module.c* of the Linux kernel source base. This function was used to help find the offset of the num_symtab, symtab, and strtab members of the struct module.

```
CODE FRAGMENT #1 (C):
if (symnum < mod->num_symtab)
CODE FRAGMENT #2 (3.1.0-7.fc16.i686.PAE):
00 00 00 1E 8B 93 14 01 00 00    mov edx, dword ptr
                                   [ebx+0x00000114]
00 00 00 24 39 D0                  cmp eax, edx
CODE FRAGMENT #3 (3.6.10-4.fc18.i686.PAE):
00 00 00 38 8B 93 14 01 00 00    mov edx, dword ptr
                                   [ebx+0x00000114]
00 00 00 3E 39 C2                  cmp edx, eax
```

In the above code fragments, the constant 0×114 within the indexed instructions is the offset for the num_symtab member. As the methods used by compilers can be very different, all possible instruction formats for various architectures must be clarified. Code fragment 4 is again from the *module_get_kallsym()* function, and fragments 5–8 illustrate the disassembly of the instruction that accesses the strtab and symtab members of the module for different architectures.

```
CODE FRAGMENT #4(C):
strlcpy(name, mod->strtab +  mod->symtab[symnum].
st_name,
KSYM_NAME_LEN);
CODE FRAGMENT #5(2.6.32-504.el6.i686)
00 00 00 6C 8B 8B F0 00 00 00     mov ecx, dword ptr
                                  [ebx+0x000000F0]
00 00 00 72 8B 93 00 01 00 00     mov edx, dword ptr
                                  [ebx+0x00000100]
00 00 00 78 03 14 01                add edx, dword ptr
                                  [ecx+eax]
CODE FRAGMENT #6 (3.1.0-7.fc16.i686.PAE)
00 00 00 47 8B 8B 0C 01 00 00     mov ecx, dword ptr
                                  [ebx+0x0000010C]
00 00 00 4D 8B 93 1C 01 00 00     mov edx, dword ptr
                                  [ebx+0x0000011C]
00 00 00 53 03 14 01                add edx, dword ptr
                                  [ecx+eax]
CODE FRAGMENT #7 (3.6.10-4.fc18.i686.PAE)
00 00 00 84 8B 8E 10 01 00 00     mov ecx, dword ptr
                                  [esi+0x00000110]
00 00 00 8A 8B 96 20 01 00 00     mov edx, dword ptr
                                  [esi+0x00000120]
00 00 00 90 03 14 01                add edx, dword ptr
                                  [ecx+eax]
CODE FRAGMENT #8 (3.10.0-123.el7.x86_64)
00 00 00 C0 8B 93 78 01 00 00     mov edx, dword ptr
                                  [ebx+0x00000178]
00 00 00 C6 8B 34 02              mov esi, dword ptr
                                  [edx+eax]
00 00 00 C9 BA 80 00 00 00        mov edx, 00000080
00 00 00 CE 48                    dec eax
00 00 00 CF 03 B3 90 01 00 00 add esi, dword ptr
                                  [ebx+00000190]
```

The *module_get_kallsym* function is exported as a symbol to */proc/kallsyms*, and its address can be obtained from the process described in Section 4.3. In this way, the *state*, *name*, *module_core*, and *source_list* members of module structures can be analyzed based on the *kdb_lsmod* function defined in /kernel/debug/kdb/kdb_main.c.

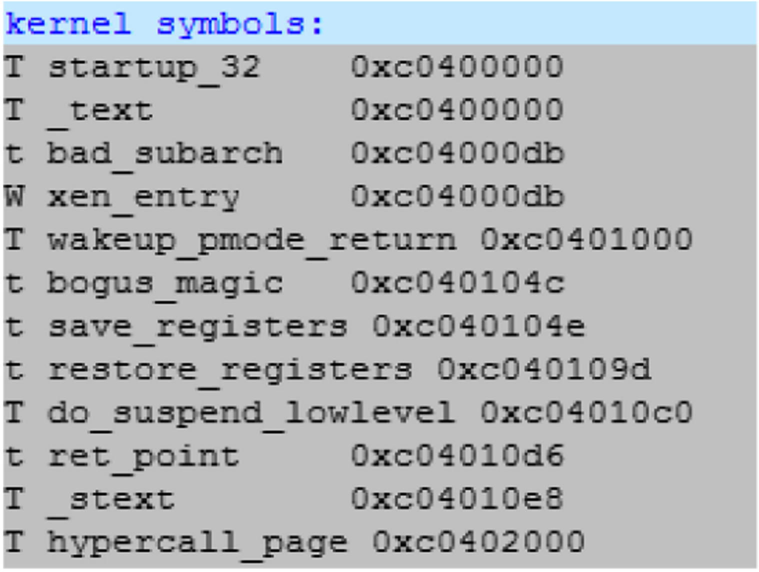
4.4.2 Process information extraction

Every process is represented by a structure named *task_struct*, which is defined in the /usr/src/linxu-2.4/include/linux/sched.h file. The *init_task* symbol corresponds to the *task_struct* structure address of the swapper, where the PID is zero. The *task_struct* structures of all active processes are doubly linked to each other. By traversing the double-linked list, all of the running processes can be identified. Moreover, the *task_struct* structure includes some objects that correspond to information regarding the current state of a process, such as *struct mm_struct *mm*, *struct fs_struct *fs*, *struct files_struct *files*, and *struct thread_struct thread*. Using these objects, we can obtain information on the memory management, file, and thread of the processes.

4.4.3 Module information extraction

Similar to the process information, all module structures are doubly linked to each other. By acquiring a module using the *module* symbol, the other modules can be identified from this doubly linked list.

To link a module, the *sys_init_module()* service initializes the syms and gpl_syms fields of the module object so that they point to the in-memory tables of symbols



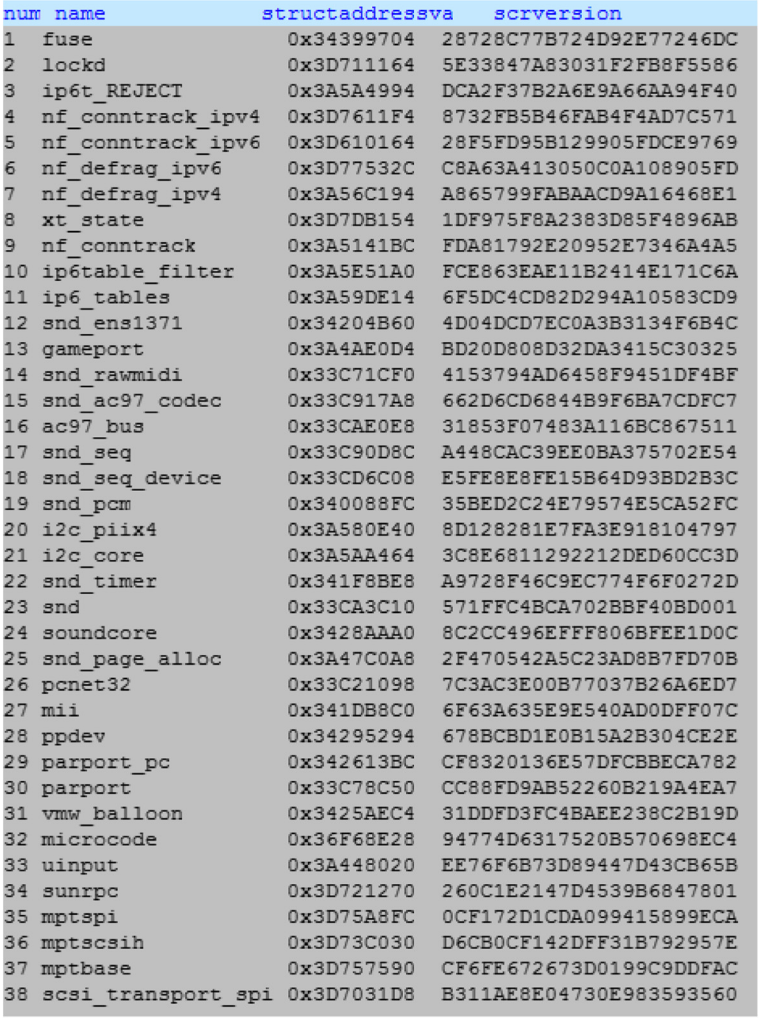
```
kernel symbols:
T startup_32     0xc0400000
T _text          0xc0400000
t bad_subarch    0xc04000db
W xen_entry      0xc04000db
T wakeup_pmode_return 0xc0401000
t bogus_magic    0xc040104c
t save_registers 0xc040104e
t restore_registers 0xc040109d
T do_suspend_lowlevel 0xc04010c0
t ret_point      0xc04010d6
T _stext         0xc04010e8
T hypercall_page 0xc0402000
```

Fig. 8 Partial kernel symbols

exported by the module. Some special kernel symbol tables are used by the kernel to store the symbols that can be accessed by modules with their corresponding addresses. These are contained in three sections of the kernel code segment: the *_kstrtab* section includes the names of the symbols, the *_ksymtab* section includes the addresses of the symbols, and the *_ksymtab_gpl* section includes the addresses of the symbols that can be used by the modules released under a GPL-compatible license. Only the kernel symbols actually used by some existing modules are included in the table. Linked modules can also export their own symbols so that other modules can access them. Although these symbols are critical during an investigation, they have largely been neglected in previous research. For instance, the *vm_list* symbol exported by the kvm module can be used to analyze the virtual machine information running on the current physical machine.

To obtain the exported symbols from the memory image, some objects of the module structure can be used, such as *const struct kernel_symbol *syms*, *const struct kernel_symbol *gpl_syms*, *Elf_Sym *symtab*, and *Char *strtab*. Among these objects, the symtab object is particularly important because it assists in the recovery of the symbol and string tables for kallsyms.

As for other system information, the *rt_hash_mask*, *rt_hash_table*, and *net_namespace_list* symbols are used to obtain information about the network configuration and current network connections; the *boot_cpu_data* symbol is used to obtain CPU information; the *log_buf* symbol corresponds to system log and debug information; and the *iomem_resource* symbol reflects the available physical address space of the target computer.



num	name	structaddressva	scrversion
1	fuse	0x34399704	28728C77B724D92E77246DC
2	lockd	0x3D711164	5E33847A83031F2FB8F5586
3	ip6t_REJECT	0x3A5A4994	DCA2F37B2A6E9A66AA94F40
4	nf_conntrack_ipv4	0x3D7611F4	8732FB5B46FAB4F4AD7C571
5	nf_conntrack_ipv6	0x3D610164	28F5FD95B129905FDCE9769
6	nf_defrag_ipv6	0x3D77532C	C8A63A413050C0A108905FD
7	nf_defrag_ipv4	0x3A56C194	A865799FABAACD9A16468E1
8	xt_state	0x3D7DB154	1DF975F8A2383D85F4896AB
9	nf_conntrack	0x3A5141BC	FDA81792E20952E7346A4A5
10	ip6table_filter	0x3A5E51A0	FCE863EAE11B2414E171C6A
11	ip6_tables	0x3A59DE14	6F5DC4CD82D294A10583CD9
12	snd_ens1371	0x34204B60	4D04DCD7EC0A3B3134F6B4C
13	gameport	0x3A4AE0D4	BD20D808D32DA3415C30325
14	snd_rawmidi	0x33C71CF0	4153794AD6458F9451DF4BF
15	snd_ac97_codec	0x33C917A8	662D6CD6844B9F6BA7CDFC7
16	ac97_bus	0x33CAE0E8	31853F07483A116BC867511
17	snd_seq	0x33C90D8C	A448CAC39EE0BA375702E54
18	snd_seq_device	0x33CD6C08	E5FE8E8FE15B64D93BD2B3C
19	snd_pcm	0x340088FC	35BED2C24E79574E5CA52FC
20	i2c_piix4	0x3A580E40	8D128281E7FA3E918104797
21	i2c_core	0x3A5AA464	3C8E6811292212DED60CC3D
22	snd_timer	0x341F8BE8	A9728F46C9EC774F6F0272D
23	snd	0x33CA3C10	571FFC4BCA702BBF40BD001
24	soundcore	0x3428AAA0	8C2CC496EFFF806BFEE1D0C
25	snd_page_alloc	0x3A47C0A8	2F470542A5C23AD8B7FD70B
26	pcnet32	0x33C21098	7C3AC3E00B77037B26A6ED7
27	mii	0x341DB8C0	6F63A635E9E540AD0DFF07C
28	ppdev	0x34295294	678BCBD1E0B15A2B304CE2E
29	parport_pc	0x342613BC	CF8320136E57DFCBBECA782
30	parport	0x33C78C50	CC88FD9AB52260B219A4EA7
31	vmw_balloon	0x3425AEC4	31DDFD3FC4BAEE238C2B19D
32	microcode	0x36F68E28	94774D6317520B570698EC4
33	uinput	0x3A448020	EE76F6B73D89447D43CB65B
34	sunrpc	0x3D721270	260C1E2147D4539B6847801
35	mptspi	0x3D75A8FC	0CF172D1CDA099415899ECA
36	mptscsih	0x3D73C030	D6CB0CF142DFF31B792957E
37	mptbase	0x3D757590	CF6FE672673D0199C9DDFAC
38	scsi_transport_spi	0x3D7031D8	B311AE8E04730E983593560

Fig. 9 Modules list

5 Evaluation

Based on the techniques described above, we developed a Linux memory analysis system named RAMAnalyzer. In this section, we present our experimental results. An experiment to test the effectiveness of RAMAnalyzer with 26 Linux kernels (from 2.6.18 to 4.2.0) is described in Section 5.1, and the performance of the proposed tool is reported in Section 5.2. All of our experiments were performed on a host machine with an Intel Core i5-4210U CPU, 4-GB memory, and a 64-bit Windows 7 OS.

5.1 Effectiveness

The following memory images were chosen: DFRWS 2008 forensics challenge, volatility memory samples, and memory snapshots from virtual machines running on the VMware Workstation. The test flow and execution results of RAMAnalyzer are described below.

Taking a memory image from the 3.1.0-7.fc16.i686.PAE system as an example, the first step was to identify the kernel version by searching for *linux_banner* content and *vmcoreinfo_data* content. The database was then checked to identify any prior knowledge of this kernel version. If the database returned no results, the kallsyms location symbol values were restored by disassembling the dynamically loaded code of the *update_iter* function in the memory image. The initialization result, including kernel version information and kallsyms location symbol values, is displayed in Fig. 7. Using the kallsyms location symbol values, the kernel symbols were extracted. The partial result is shown in Fig. 8.

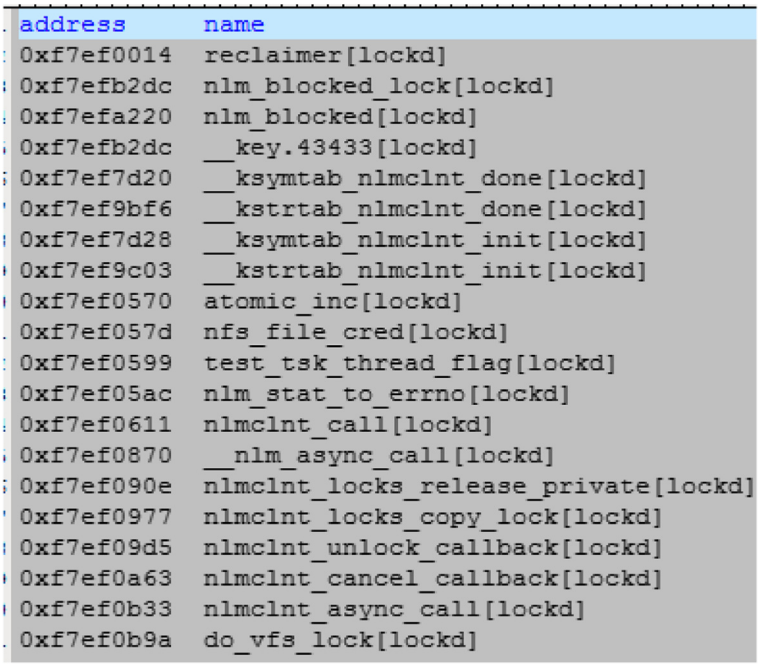
Several symbols were selected to extract live system information from the memory image, including process information, module information, network connection information, and system log information. The loaded kernel modules information extracted from a memory image of the 3.1.0-7.fc16.i686.PAE system are listed in Fig. 9, and the symbols exported from the lockd module are listed in Fig. 10.

5.2 Performance evaluation

To ensure that large changes in the kernel algorithms do not affect the validity of our approach, memory images from various kernel versions were used to test the performance of RAMAnalyzer. Some of the kernels used in the experiment are listed in Table 1.

We measured the execution speed of RAMAnalyzer when only a memory image was provided. As illustrated in Fig. 1, the key steps of our approach are kernel version identification and symbol table file recovery, and these are our evaluation targets. From Figs. 11 and 12, we can see that 15–78 ms were required for kernel version identification, and 347-15,693 ms were needed for the recovery of kallsyms location symbol values and kernel symbols.

After obtaining the kernel symbols, the *modules* symbol was used to find the loaded kernel modules and exported



```
address     name
0xf7ef0014  reclaimer[lockd]
0xf7efb2dc  nlm_blocked_lock[lockd]
0xf7efa220  nlm_blocked[lockd]
0xf7efb2dc  __key.43433[lockd]
0xf7ef7d20  __ksymtab_nlmclnt_done[lockd]
0xf7ef9bf6  __kstrtab_nlmclnt_done[lockd]
0xf7ef7d28  __ksymtab_nlmclnt_init[lockd]
0xf7ef9c03  __kstrtab_nlmclnt_init[lockd]
0xf7ef0570  atomic_inc[lockd]
0xf7ef057d  nfs_file_cred[lockd]
0xf7ef0599  test_tsk_thread_flag[lockd]
0xf7ef05ac  nlm_stat_to_errno[lockd]
0xf7ef0611  nlmclnt_call[lockd]
0xf7ef0870  __nlm_async_call[lockd]
0xf7ef090e  nlmclnt_locks_release_private[lockd]
0xf7ef0977  nlmclnt_locks_copy_lock[lockd]
0xf7ef09d5  nlmclnt_unlock_callback[lockd]
0xf7ef0a63  nlmclnt_cancel_callback[lockd]
0xf7ef0b33  nlmclnt_async_call[lockd]
0xf7ef0b9a  do_vfs_lock[lockd]
```

Fig. 10 Symbols exported by lockd

Table 1 Sample of kernel versions used for testing RAMAnalyzer

OS-kernels	Linux version	Architecture
2.6.18-8.1.15.el5	Centos 5	×86
2.6.18-238.el5	Centos 5.6	×86
2.6.24-26-generic	Ubuntu-8.04.4	×86_64
2.6.26-2-686	Debian 2.6.26-26	×86
2.6.32-33-generic	Ubuntu-10.04.3	×86
2.6.32-279.el6.x86_64	Centos-6.3	×86_64
2.6.32-300.10.1.el5uek	Oracle Linux 5	×86_64
2.6.32-504.el6.i686	Centos 6.6	×86
2.6.38-generic	Ubuntu-11.04	×86
2.6.43.8-1.fc15.x86_64	Fedora 15	×86_64
3.1.0-7.fc16.i686.PAE	Fedora 16	×86
3.2.0-23-generic	Ubuntu-12.04	×86_64
3.3.4-5.fc17.x86_64	Fedora 17	×86_64
3.6.10-4.fc18.i686.PAE	Fedora 18	×86
3.6.11-4.fc16.i686.PAE	Fedora 16	×86
3.6.11-4.fc16.x86_64	Fedora 16	×86_64
3.8.0-19-generic	Ubuntu-13.04	×86_64
3.9.5-301.fc19.i686.PAE	Fedora 19	×86
3.9.5-301.fc19.x86_64	Fedora 19	×86_64
3.10.0-123.el7.x86_64	Centos 7	×86_64
3.11.1-200.fc19.x86_64	Fedora 19	×86_64
3.11.10-301.fc20.x86_64	Fedora 20	×86_64
3.13.0-24-generic	Ubuntu 14.04	×86
3.16.0-30-generic	Ubuntu-14.04.02	×86_64
3.19.0-15-generic	Ubuntu 15	×86
4.2.0-1-686-pae	Deepin 15	×86

symbols. The time required for this process is shown in Fig. 13.

The experimental results prove that RAMAnalyzer can deal with memory images from a wide range of kernel versions and demonstrate that its execution time is acceptable.

6 Conclusions

Based on kernel code reconstruction, this paper has proposed an adaptive approach for Linux memory analysis that can address a Linux memory image without information about the kernel version or System.map file. We implemented a prototype named RAMAnalyzer that is made up of five main components: kernel version identification, symbol table file recovery, kallsyms location symbol value recovery, live system information extraction, and database information extension. Our experimental results with a number of Linux memory images show that RAMAnalyzer can automatically identify the kernel version and recovery kernel symbols. Based on the kernel symbols, RAMAnalyzer then extracts live system information about the target system at the time of memory acquisition.

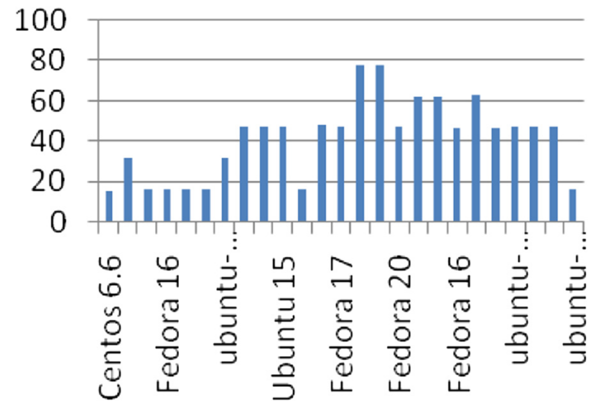


Fig. 11 KVI execution time

The primary advantages of RAMAnalyzer are:

- The ability to deal with memory images without precise kernel version information and symbol information.
- The ability to identify the precise kernel version and recovery kernel symbols automatically, which means it can deal with memory images from different kernel versions. Furthermore, kernel symbol information obtained in this way is more accurate because symbol information from identical kernel versions can vary under different configuration options.
- As well as kernel symbol information, RAMAnalyzer can acquire the symbols exported from modules, which play an important role in the investigation procedure.
- Based on the above techniques, RAMAnalyzer has adaptability to deal with mainstream Linux kernel memory images and has high execution efficiency.

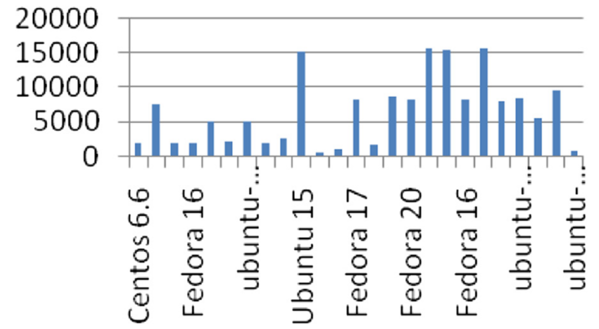


Fig. 12 KLSR and STFR execution time

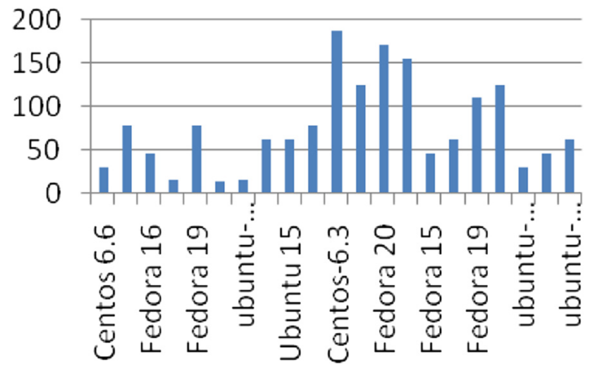


Fig. 13 Modules symbol recovery execution time

From the advantages of RAMAnalyzer, we can see that our solution can provide a solution for the challenge described in Section 1 and meet the need of scenarios described in Section 2.1. With the advent of mobile cloud computing, the development of Linux is accelerating and its security is becoming increasingly crucial. The techniques proposed in this paper provide forensics researchers with a starting point to delve into Linux memory forensics, which plays an important role in security and forensics investigations. Furthermore, these techniques can be conveniently embedded into other forensics frameworks.

To enhance the performance of RAMAnalyzer, the following research will be undertaken: first, owing to the differences in the kallsyms configurations of Linux kernel versions, there are various initial */proc/kallsyms*. To improve the processing speed, it is essential to scan the *kalsyms_address* candidate values. Second, to improve the adaptive capacity of cloud environments, RAMAnalyzer was verified to be effective using memory images from the KVM host machine. However, further experiments are required to verify its efficacy on memory images from Xen host machines.

Competing interests

The authors declare that they have no competing interests.

Authors' contributions

SH and XX carried out the main research of this work. LH performed the experiments. SH and LH conceived of the study, participated in its design and coordination, and helped to draft the manuscript. All authors read and approved the final manuscript.

Acknowledgements
This research is supported by the National Natural Science Foundation of China (61472225), Shandong Natural Science Foundation (ZR2014FM003), and Shandong Province Outstanding Young Scientists Research Award Fund Project(BS2014DX007).

Received: 6 January 2016 Accepted: 14 June 2016

References

1. DFRWS 2005 Forensics Challenge. http://www.dfrws.org/conferences/dfrws-usa-2005
2. L Wang, L Xu, S Zhang, Network connections information extraction of 64-bit windows 7 memory images. Forensics in Telecommunications, Information, and Multimedia. **56**, 90–98 (2010)
3. DFRWS 2005 Forensics Challenge: Memparser. http://sourceforge.net/projects/memparser
4. A Schuster, Searching for processes and threads in microsoft windows memory dumps. Digit. Investig. **3**, 10–16 (2006)
5. R Zhang, L Wang, S Zhang, Windows memory analysis based on kpcr. Fifth International Conference On Information Assurance and Security. **2**, 677–680 (2009)
6. JD Kornblum, Using every part of the buffalo in windows memory analysis. Digit Investig. **4**(1), 24–29 (2007)
7. A Schuster, The impact of microsoft windows pool allocation strategies on memory forensics. Digit. Investig. **5**, 58–64 (2008)
8. S Zhang, L Wang, R Zhang, Q Guo, Exploratory study on memory analysis of windows 7 operating system. 3rd International Conference On Advanced Computer Theory and Engineering (ICACTE). **6**, 373–377 (2010)
9. B Dolan-Gavitt, Forensic analysis of the windows registry in memory. Digit. Investig. **5**, 26–32 (2008)
10. R Van Baar, W Alink, A Van Ballegooij, Forensic memory analysis: Files mapped in memory. Digit. Investig. **5**, 52–57 (2008)
11. S Hejazi, C Talhi, M Debbabi, Extraction of forensically sensitive information from windows physical memory. Digit. Investig. **6**, 121–131 (2009)
12. Q Zhao, T Cao, Collecting sensitive information from windows physical memory. J. Comput. **4**(1), 3–10 (2009)
13. M Sikorski, A Honig, *Practical Malware Analysis: the Hands-on Guide to Dissecting Malicious Software*. (no starch press, Scn Francisco, 2012), pp. 145–157
14. A Ligh, MH Case, J Levy, A Walters, *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*. (John Wiley & Sons, 2014), pp. 611–635
15. Volatility: Linux Memory Forensics. https://code.google.com/archive/p/volatility/wikis/LinuxMemoryForensics.wiki
16. Rekall Memory Forensic Framework. http://www.rekall-forensic.com/
17. MI Cohen, Characterization of the windows kernel version variability for accurate memory analysis. Digit. Investig. **12**, 38–49 (2015)
18. A Petroni, NL Walters, T Fraser, WA Arbaugh, Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory. Digit. Investig. **3**(4), 197–210 (2006)
19. SecondLook Linux Memory Dump Samples. https://www.forcepoint.com/
20. A Case, L Marziale, GG Richard, Dynamic recreation of kernel data structures for live forensics. Digit. Investig. **7**, 32–40 (2010)
21. T Haruyama, H Suzuki. https://media.blackhat.com/bh-eu-12/Haruyama/bh-eu-12-Haruyama-Memory_Forensic-Slides.pdf