Fine-grained analysis method for Android volatile memory

Peijun Feng, Qingbao Li and Zhifeng Chen*
State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, China

*E-mail: xiaohouzi06@163.com

Abstract. Android dominates the mobile operating system market. Volatile memory analysis of Android devices has been the focus of research on mobile forensics technology. However, due to the semantic gap between the kernel and the volatile memory allocator, existing Android volatile memory analysis methods are coarse-grained. With the volatile memory capacity of Android devices increasing, these methods cannot satisfy the need of Android volatile memory analysis accuracy. In this paper, we first discuss the address space layout of Android processes and the management mechanism of Jemalloc, the default Android volatile memory allocator. Then, we bridge the semantic gap by utilizing the boundary auto alignment feature of the data structure of Jemalloc. Finally, we propose a Fine-grained Analysis Method for Android volatile Memory, called FAMAM. Experimental results shows that FAMAM has an accurate data analysis capability as well as a good robustness. In addition, we successfully use FAMAM to discover new storage patterns for username and password of Wechat.

1. Introduction
Android has dominated the mobile market in recent years and been the focus of research in the field of mobile device forensics. As more and more Android applications using a variety of data encryption to protect data, for example WeChat[1], the most popular social network application in China, uses SQLCipher[2] to encrypt the entire database, even if the investigators could acquire the application database by some acquisition methods based on non-volatile memory forensics, they cannot direct access to application data, which has brought a huge challenge for Android mobile forensics.

Volatile memory is a dynamic reflection of the current state of the OS. It not only records the system kernel data such as the state of processes and networks, but also retains the application data. In addition, application data is stored in plain text in volatile memory, resulting in the ability to bypass database encryption for application data acquisition. So the research on volatile memory analysis is of great value for mobile forensics[3].

1.1. Motivation
The existing Android volatile memory analysis methods are coarser-grained, which simply take entire process physical address area even the entire volatile memory image as a large chunk of data for analysis, resulting in the inaccuracy of Android memory forensics analysis.

The dynamic data stored in the heap of process, such as username and password of applications, histories of browser and records of instant messaging software, are the most interesting information to investigators. However, existing forensics researches on Android heap data concentrated primarily on
serialized Java objects contained in the heap and not on the way heap objects are managed, which may lead to a fragmentary forensics result.

1.2. Contribution
In this paper, we make the following contributions:
• We discuss the kernel structure of Android process and the management mechanism of Jemalloc (Jason Evans malloc) [4], and explain how the Android heap structures are organized and where the data is located in volatile memory.
• Based on the above analysis results, we bridge the semantic gap between the kernel and the volatile memory allocator by utilizing the boundary auto alignment feature of the data structure of Jemalloc.
• We propose a Fine-grained Analysis method based on volatile Memory Forensics for Android heap data, called FAMAM.
• We successfully use FAMAM to analyze the heap data of Wechat, especially discover a new pattern for username and password of Wechat.

1.3. Outline
The rest of this paper is structured as follows: Section 2 shows the related works; Section 3 presents the principle of FAMAM. Section 4 presents the implementation of FAMAM. Section 5 tests and evaluates FAMAM in terms of functionality and robustness. Section 6 presents the new pattern of username and password of Wechat discovered by FAMAM. Section 7 concludes the paper.

2. Related works
Lime [5] implemented by Sylve is a widely used volatile memory acquisition tool for Linux/Android. Lime extracts device volatile memory through a LKM (Loadable Kernel Module) injected in system kernel and supports dumping memory image by SD card or TCP connections with 99.46% accuracy. Stüttgen [6] proposed a robust volatile memory acquisition method with minimal target impact for Linux/Android, which enhances the versatility of Lime in different kernel versions. In this paper, we use Lime to acquire the volatility memory images of Android device.
Volatility [7] is a open source memory forensics framework, which obtains detailed system state by analyzing the kernel structures. All functions of Volatility are implemented by plugins written in Python, so it is highly extensible. However, these functions mainly focus on the kernel level, especially the analysis of kernel structures, processes and networks, while there are only a few functions on user applications.
Zhou et al. [8] introduces some methods to dump the Android volatility memory and used some software tools to analyze the memory to extract the encrypted chats and deleted messages of WeChat. However, they simply took the entire volatile memory image as a large chunk of data for analysis, resulting in the inaccuracy of forensics result.
Macht [9] and Lepper [10] both perform Android volatility memory forensics by a comprehensive bottom-up methods and analyze the heap data of a set of applications. However, their analysis concentrated primarily on serialized Java objects contained in the heap and not on the way heap objects are managed, leading to a fragmentary forensics result.
Ntantogian et al. [11] discover common patterns for usernames and passwords of mobile apps. However they still use the string matching method to search the entire memory image, which is coarse-grained, and have not found the new pattern that we found.
Cohen [12], Ligh [13] and Block [14] et al. have conducted detailed researches on the fine-grained volatile memory analysis technology, especially the process heap data structure and its management methods, which make great contributions to volatile memory forensics research work. However, it is a pity that these researches are aimed at Windows, MacOS or Linux. Since the memory allocator used by Android is different from the memory allocator used by other systems, the forensic analysis method is completely different in the detailed implementation.
3. Principle analysis

In this section, we first introduce the result of our analysis on the Android process address space layout and system memory allocation mechanism from a volatile memory forensic perspective, and then we present the bridge principle of semantic gap between the kernel and Jemalloc.

3.1. The Android process address space layout

Android adopts Linux kernel and inherits the process management mechanism of Linux. Each Linux/Android process corresponds to a process controller structure, called task_struct. The mm pointer of the task_struct points to the memory descriptor structure of the process, managing the virtual address space of the process, called mm_struct. The pgd pointer of the mm_struct points to global page table of the process, and the mapping from virtual address to physical address is implemented through the multi-level page table. The address space layout of Linux/Android process is shown in figure 1.

The Linux/Android process address space contains the following data regions:

- Code segment stores the program code and some statically allocated read-only constant variables, such as character constants.
- Bss segment stores statically allocated uninitialized global variables.
- Data segment stores statically allocated initialized global variables.
- Brk segment stores dynamically allocated data smaller than 128KB.
- Mmap segment stores dynamically allocated data larger than 128KB.
- Stack segment stores dynamically temporary created local variables.

Without considering the shared memory, there are two ways to dynamically allocate memory for Linux processes: brk and mmap. Brk allocates small contiguous chunks of memory and increases the value of brk pointer of mm_struct. Mmap allocates discontinuous large chunks of memory and concatenates all mmap regions in a process into a double linked list. However, for Android processes, we find that the value of brk pointer of mm_struct is equal to the value of start_brk pointer, that means the Android processes only use mmap to allocate memory. In other words, the Android heap is the mmap area.

![Figure 1. Linux/Android process address space layout.](image-url)
Android forensics investigators can locate the task_struct of process according to the process name or PID by traversing the tasks list. The mm_struct and the virtual address of various data areas are further obtained according to the offset among various data structures as shown in figure 1. Finally, the physical address of each data area are obtained through the page table transformation.

The mobile device user behaviors data, such as instant messages, browser history, and GPS geographic location, which are concerned by investigators, are almost all dynamically generated and stored in the heap of processes. However, the inner structure of the heap is still a black box for investigators. Heap contain not only various kinds of data, but also management and control information of data structures. Indiscriminately regarding heap as data sources of evidence, is not enough to support the accuracy of digital evidence. Therefore, the research of heap data analysis method is particularly important, and further analysis of the Jemalloc, the default Android heap manager, is required.

3.2. The Android process address space layout

Since version 5.0, Android has adopted Jemalloc as the default memory allocator. Jemalloc is originally a new generation memory allocator developed by Jason Evans for FreeBSD with powerful multi-core and multi-threaded allocation capability. Jemalloc manages the memory data through the data structures of Arena, Chunk, Run and Region. The relationships of these structures are presented in figure 2.

![Figure 2. The relationships of data structures in Jemalloc.](image)

Arena is the top-level structure in Jemalloc, and the data structure used to describe the state of Arena is arena_s. Jemalloc divides the heap memory into several separate areas, each of which is managed and maintained through Arena structure. In Android platform, the number of Arena is limited to no more than two by the makefile of Jemalloc. Arena divides the heap memory area managed by itself into several Chunks, and manages Runs and different types of Regions through these Chunks.

Chunk is the secondary level structure in Jemalloc, size of which defaults to 4MB and must be aligned to the 4MB boundary. The data structure for the Jemalloc to record the state of chunk is arena_chunk_s, but this structure is just a header for the whole chunk, containing only some statistics information and a dynamic array map. Chunk and its internal Runs are made up of many pages, and the allocation state, properties, offsets, and other tag information of these pages are recorded in the map array of Chunk. Therefore, in addition to recording page state, the map array also determines the base address retrieval of Runs.

Run is the heap memory allocation main body of Jemalloc, size of which must be aligned to the page size (default to 4KB). The data structure for the Jemalloc to record the state of chunk is arena_run_s, and similar to arena_chunk_s, this structure is also just a header for the whole Run. Run is internally divided into Regions of the same size and records the state of these Regions by the bitmap of its header.
Region is the basic heap memory data storage unit in Jemalloc. As shown in table 1, according to the storage space size, Jemalloc divides the Region into three types: small, large and huge. Small Region and large Region are managed by Run. However, huge Region is directly managed by Arena, which is equivalent to Chunk.

**Table 1.** Spacings and sizes of different types regions.

| Types   | Spacing | Sizes                  |
|---------|---------|------------------------|
| small   | N/A     | [8B]                   |
|         | 16B     | [16B, 32B, 48B, ..., 128B] |
|         | 64B     | [192B, 256B, 320B, ..., 512B] |
|         | 256B    | [768B, 1024B, 1280B, ..., 3840B] |
| large   | 4KB     | [4KB, 8KB, 12KB, ..., 4072KB] |
| huge    | 4MB     | [4MB, 8MB, 12MB, ...]   |

3.3. The bridge principle of the semantic gap between kernel and Jemalloc

From mobile forensics investigators’ perspectives, we have to locate the physical address of every data structure both in the kernel and Jemalloc, so that we can extract the accurate data from the volatile memory. However, due to the address space of Android process is layout by the kernel and Jemalloc resides in the system software level which is above the kernel level, there is a semantic gap between the kernel and Jemalloc. In addition to that, although the physical addresses of kernel level data structures can be resolved step by step through a system.map file, the physical addresses of Jemalloc data structures cannot be located independently because the address of Arenas, a global vector group which contains the top-level data structure in Jemalloc, is not exported. Hence, the key to solve this problem is to accurately locate the physical address of Arena structure in Jemalloc.

As shown in figure 2, there is a pointer named arena in Chunk points to the Arena that the Chunk belongs to, and this provides a new idea for Arena structure positioning. Because the size of Chunk defaults to an integer multiple of 4MB and the bound of that is aligned with 4MB. So each 4MB starting address in the Android process address space is likely to be a Chunk header structure.

Therefore, as shown in figure 3, taking advantage of the boundary auto alignment feature of Chunk structure, we can bridge the semantic gap between the kernel and Jemalloc, and accurately locate the physical address of each data structure in Jemalloc.

**Figure 3.** The bridge of the semantic gaps between the kernel and Jemalloc.

4. Implementation

Through the analysis on the bridge principle of the semantic gap between the kernel and Jemalloc in previous section, based on the memory acquisition tool Lime and analysis framework Volatility, we purpose a Fine-grained Analysis method for Android volatile Memory, called FAMAM.
The overall framework of FAMAM is shown in figure 4, including three modules: memory acquisition module, data locating module and data analysis module.

4.1. Memory acquisition module
FAMAM uses Lime to acquire the physical memory image of Android device, this process can be divided into the following steps:
- Download the device kernel source from the official website according to the device model, modify the kernel source (hijacking system calls, etc.), and set the Root permission back door.
- Turn on the LKM loadable option in the kernel configuration, recompile the kernel, and generate a new boot.img.
- Unlock the device (if needed), enter the fastboot mode, and brush in the new boot.img to enable the device to load the LKM. Brk segment stores dynamically allocated data smaller than 128KB.
- Install Lime on the device, configure the device kernel source path, compile and generate lime.ko kernel module.
- Get the ADB shell with a Root permission using the preset back door.
- Load the lime.ko into the device with a optional transmission mode (SDcard or TCP) and export the device physical memory image which can be analyzed by Volatility.

4.2. Data locating module
The locating of Android process heap data requires not only the kernel data structures, but also data structures of Jemalloc memory allocator. Therefore, FAMAM needs to combine several methods to locate the process heap data structure, including System.map, Linux kernel source and Jemalloc source.
- System.map is a symbol table of Linux kernel, recording the address of symbols such as functions and global variables of kernel. With the help of system.map, the task_struct address of init_task, the first process of Linux kernel, can be found.
- Linux kernel source code defines kernel data structures such as process control block task_struct, memory control block mm_struct and virtual memory management unit vm_area_struct, which can be used to accurately calculate the offsets among various kernel data structures. By tracking tasks process list of init_task, we can not only get the page table address of any process and implement the transformation from virtual address to physical address, but also locate all data regions of any process.
- Jemalloc source code defines memory heap data structures such as Arena, Chunk, Run and Region, which we need to calculate the offset between various heap data structures. Particularly noteworthy is that for each mmap area of the target process, doing an alignment operation of the 4MB boundary can get an address of the region belongs to the Chunk, further confirm the address point to a chunk_header structure, the locating of the chunk structure can be realized.

Based on the above analysis, the pseudo-code of Android process heap data locating algorithm is shown in table 2.
Table 2. The pseudo-code of data locating algorithm.

```c
/* Gets the start address of init_task from system.map */
struct task_struct *p = read_system_map(init_task);
/* Gets the start address of target process according to PID */
struct task_struct *entry = NULL;
list_for_each_entry(entry, &(p->tasks), tasks) {
    if (PID == entry->pid)
        entry = p;
}
/* Locate and export the target process mmap areas */
if (entry) {
    struct mm_struct *mm = entry->mm;
    struct vm_area_struct *vma = mm->mmap;
    while (vma) {
        export(vma->vm_start, vma->vm_end);
        vma = vma->next;
    }
}
/* Locate the chunk structure in the exported mmap areas */
for_all_exported_areas {
    struct arena_chunk_s *chunk_ptr = chunk_addr2base(start_addr);
    if (chunk_ptr is a chunk_header)
        export this area as a chunk;
}
/* Locate the data area in the exported chunk structure */
for_all_exported_chunks {
    if (chunk contains a huge region) //huge region
        acquire the data in the huge region;
    else {
        //large or small region
        struct arena_run_s *runs[] = get_runs_in_chunk(maps);
        for_all_runs_in_chunk {
            regions[] = get_regions_in_run(bitmap_info);
            acquire the data in the regions;
        }
    }
}
```

4.3. Data analysis module

Application data in memory is usually stored in plain text, so that regular expression matching can be used to mark and analyze application data in memory with specific format or special meaning. As the heap data locating algorithm purposed in this paper can directly locate the application data within the Region structure, the data used to record the data structure relations outside the Region structure will not be taken as the application data. In this case, the regular expression matching method can be used to obtain a higher accuracy.

Zhou[8], Zhang[15] and Wu[16] et al. have studied the extraction and analysis techniques of WeChat application data from different perspectives. Based on their analysis results of WeChat, through further experimental verification, we confirm that the process of WeChat is com.tencent.mm and the chat text of that stored in memory is in this format:

<Unix timestamp><Username><Content><Type>

- Unix timestamp field represents the timestamp of text sent or received, which is a hexadecimal number of 6 bytes in length.
• Username field represents the username of the text sent or received and can be obtained via WeChat unencrypted database IndexMicroMsg.db.
• Content field represents the specific content of the text.
• Type field represents the received and sent types, 0x7B00 for received and 0x1BFF for sent.

5. Test and evaluations
This section tests and evaluates FAMAM in terms of functionality and robustness, and then presents a new storage pattern of application authentication information such as username and passwords. Functional testing is used to test whether FAMAM can acquire user data for the Android application processes. Robustness tests are used to test the ability of FAMAM to maintain access to user data for Android application processes under special situations.

The test environment is as follows: the test device is Samsung Galaxy Note3, device model is SM-N9008V, baseband version is N9008VZMUDOH2, memory size is 3G, kernel version is 3.4.0, Jemalloc version is 3.6.0, Lime version is 1.8, Volatility version is 2.6, WeChat version is 6.6.5, the forensic analysis platform is A Lenovo PC (Lenovo-A6810f) with Ubuntu 14.04 OS.

5.1. Functional test
The chat text of WeChat is selected as the test object to evaluate the functional validity of FAMAM. The test steps are as follows:
• Initialize the test device (restore factory settings).
• Perform steps in Section 4.1 to prepare for the acquisition of the physical memory image of text device.
• Install WeChat on the test device, login the test account, randomly send 20 chat text messages, and receive 20 chat text messages, and then randomly delete 5 messages sent and 5 messages received.
• Perform the acquisition step to export the acquired physical memory image to the forensic analysis platform.
• Use the plugins implemented in this paper to analyze the physical memory image.

Based on the analysis result of the regular expression format of WeChat chat text, FAMAM is used to locate and analyze the WeChat chat text in the physical memory. The result is shown in figure 5. It can be seen that the plugins have successfully acquired WeChat process heap data structure information and chat texts.

Comparing the acquired text chat records with the 40 texts set in the text steps, it is found that they are all same. The result indicates that the function of the FAMAM method proposed in this paper is effective.

There are some researchers have studied the forensics and analysis methods on WeChat. Zhou[8] performed a dump and coarse-grained analysis of Android volatile memory on WeChat. Zhang[15] and Wu[16] studied the logical acquisition and analysis methods on WeChat. In the test environment used in this paper, FAMAM is compared with their acquisition methods, and the results are shown in table 3.

![Figure 5. The analysis results of WeChat processes.](attachment:image)
Table 3. Results of comparison test.

| Method | Sent | Received | Total |
|--------|------|----------|-------|
|        | undeleted | deleted | undeleted | deleted | undeleted | deleted |
| Zhou   | 15/15 | 5/5 | 0/15 | 0/5 | 15/30 | 5/10 |
| Zhang  | 15/15 | 0/5 | 15/15 | 0/5 | 30/30 | 0/10 |
| Wu     | 15/15 | 0/5 | 15/15 | 0/5 | 30/30 | 0/10 |
| FAMAM  | 15/15 | 5/5 | 15/15 | 5/5 | 30/30 | 10/10 |

It can be seen from table 3 that compared with other WeChat data forensics and analysis methods, the FAMAM proposed in this paper can completely and effectively acquire the undeleted and deleted chat text records received and sent of WeChat, which performs better in term of data acquisition ability. The logical acquisition method of Zhang[15] and Wu[16] is implemented by analyzing WeChat database, but WeChat database need a root privilege to access while enables encryption security and deletion policies. Even if the key of WeChat database can be obtained, only undeleted data can be acquired. Similar to FAMAM, the acquisition method of Zhou[8] is also implemented based on memory forensics, but it simply takes the whole memory as a large block of data and does not analysis the data structure of memory, which is coarse-grained, so that it only can find out the text format of sent message.

5.2. Robustness test

In this paper, the ability of FAMAM to acquire user data of Android applications in the context of some special scenarios are tested, so as to evaluate the robustness of FAMAM. We still use WeChat as the test application, chat text as the test data. The test steps are consistent with those in the functional test except for perform the acquisition step after let the device enter special scenarios. These special scenarios include application background running, application account relogin, application restart, device reboot and device shutdown. The test results are shown in table 4.

It can be seen from the test results that FAMAM can acquire undeleted data and deleted data when the application process is background running. However, FAMAM cannot acquire deleted data when the account relogins, the application restarts, the device reboots, and the device shutdowns.

Table 4. Results of comparison test.

| Scenario                | Sent | Received | Total |
|-------------------------|------|----------|-------|
|                         | undeleted | deleted | undeleted | deleted | undeleted | deleted |
| background running      | 15/15 | 5/5 | 15/15 | 5/5 | 30/30 | 10/10 |
| Account relogin         | 15/15 | 0/5 | 15/15 | 0/5 | 30/30 | 0/10 |
| App restart             | 15/15 | 0/5 | 15/15 | 0/5 | 30/30 | 0/10 |
| Device reboot           | 15/15 | 0/5 | 15/15 | 0/5 | 30/30 | 0/10 |
| Device shutdown         | 15/15 | 0/5 | 15/15 | 0/5 | 30/30 | 0/10 |

6. New discovered patterns

As discovered by Ntantogian[11], the common pattern for application username and password is shown as table 5.

Table 5. The common pattern for username and password discovered by Ntantogian.

| Username | Password |
|----------|----------|
| j_username= | j_password= |
| username= | password= |
| userid> | password: |
| login i:type= | pass i:type= |
Unfortunately, we are failed to use these patterns to find username and password of Wechat. As shown in figure 6, although we find a pattern for username (username=), we do not find the actual username behind the pattern. This indicates that the Wechat has taken security measures to protect the username and password stored in memory.

Figure 6. The search result of using a common pattern for Wechat username.

However, we notice that there is a 8 bytes magic symbol "(0x)80 91 45 6F" appears behind the common username pattern. We use FAMAM to quickly locate the heap data address space of Wechat processes, and find that this magic symbol appears at other addresses in the data space, as shown in figure 7 and figure 8. More importantly that, we find the actual username and password of our test account behind this magic symbol.

Figure 7. The search result of using a common pattern for Wechat username.

Figure 8: The new pattern for WeChat password discovered by FAMAM.

This interesting finding indicates that the patterns for username and password of Wechat are as follows:

• The pattern for magic symbol: <common pattern>=<magic symbol>(8 bytes).
• The pattern for username: <magic symbol>(8 bytes) <junk>(16 bytes) <username>.
• The pattern for password: <magic symbol>(8 bytes) <junk>(8 bytes) <password>.

7. Conclusion

In order to solve the inaccuracy of forensics analysis caused by coarser-grained analysis of Android volatile memory, this paper proposes a Fine-grained Analysis Method for Android volatile Memory, called FAMAM. We analyze the address space layout of Android processes and the heap data management mechanism of Jemalloc memory allocator. FAMAM takes advantage of Lime, the famous Linux memory extract tool, to extract the memory image of Android mobile devices. According to the analysis results of the Android process address space layout and Jemalloc heap data management mechanism, the localization of Android process heap data is realized. The heap data of Android process is parsed using the method of regular expression matching. The functional test shows that FAMAM can accurately acquire the undeleted and deleted chat texts of Wechat. Compared with several other analysis methods for Wechat, FAMAM has a more accurate analysis capability. The robustness test proves that FAMAM still has the data acquisition ability under some special scenarios. More importantly, we successfully use FAMAM to discover new storage patterns for username and password of Wechat.
At the same time, FAMAM has the same limitations as Lime because it uses Lime to extract memory images: it requires a Root privilege access to device and the enable of LKM loading. For the vast majority of Android devices, it need to recompile kernel to do so. Future work will focus on reducing the limitations of this approach.

Acknowledgement

This work is supported by project 61802432 National Natural Science Foundation of China.

References

[1] WeChat. Tencent company. http://weixin.qq.com
[2] Full Database Encryption for SQLite. https://www.zetetic.net/sqlcipher/
[3] Ligh M H, Case A, Levy J, et al. The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory[ C]// Wiley Publishing, 2014.
[4] Jason Evans malloc (Jemalloc). https://wikipedia.org/wiki/C_dynamic_memory_allocation#jemalloc.
[5] Sylve J, Case A, Marziale L, et al. Acquisition and analysis of volatile memory from android devices[J]. Digital Investigation, 2012, 8(3): 175-84.
[6] Stüttgen J, Cohen M. Robust Linux memory acquisition with minimal target impact[J]. Digital Investigation, 2014, 11: S112-S119.
[7] The Volatility Foundation. http://www.volatilityfoundation.org.
[8] Zhou F, Yang Y, Ding Z, et al. Dump and analysis of Android volatile memory on Wechat[C]// IEEE International Conference on Communications. IEEE, 2015:7151-56.
[9] Macht H. Live memory forensics on android with volatility[J]. Friedrich-Alexander University Erlangen-Nuremberg, 2013.
[10] Leppert S. Android memory dump analysis[J]. Student Research Paper, Chair of Computer Science, 2012, 1.
[11] Ntantogian, C., Apostolopoulos, D., Marinakis, G. Evaluating the privacy of Android mobile applications under forensic analysis[J]. Computers & Security, 2014, 42(4):66-76.
[12] Cohen M. Forensic analysis of windows user space applications through heap allocations[C]// IEEE Symposium on Computers and Communication. IEEE, 2015: 237-44.
[13] Ligh M H, Case A, Levy J, et al. The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory[C]// Wiley Publishing, 2014.
[14] Block F, Dewald A. Linux memory forensics: Dissecting the user space process heap[J]. Digital Investigation, 2017, 22:S66-S75.
[15] Zhang L, Yu F, Ji Q. The Forensic Analysis of WeChat Message[C]// IEEE International Conference on Instrumentation & Measurement. IEEE, 2016:500-03.
[16] Wu S, Zhang Y, Wang X, et al. Forensic analysis of WeChat on Android smartphones[J]. Digital investigation, 2017, 21: 3-10.