Live acquisition of main memory data from Android smartphones and smartwatches

Seung Jei Yang a,*, Jung Ho Choi a, Ki Bom Kim a, Rohit Bhatia b, Brendan Saltaformaggi c, Dongyan Xu b

a The Affiliated Institute of ETRI, 1559, Yuseong-daero, Yuseong-gu, Daejeon 34044, Republic of Korea
b Department of Computer Science and CERIAS, Purdue University, West Lafayette, IN 47907, USA
c School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332, USA

A R T I C L E   I N F O

Article history:
Received 7 April 2017
Received in revised form 28 August 2017
Accepted 13 September 2017
Available online 19 September 2017

Keywords:
Android forensics
Main memory
Physical acquisition
Firmware update mode
Smartphones
Smartwatches

A B S T R A C T

Recent research in Android device forensics has largely focused on evidence recovery from NAND flash memory. However, pervasive deployment of NAND flash encryption technologies and the increase in malware infections which reside only in main memory have motivated an urgent need for the forensic study of main memory. Existing Android main memory forensics techniques are hardly being adopted in practical forensic investigations because they often require solving several usability constraints, such as requiring root privilege escalation, custom kernel replacement, or screen lock bypass. Moreover, there are still no commercially available tools for acquiring the main memory data of smart devices. To address these problems, we have developed an automated tool, called AMD, which is capable of acquiring the entire content of main memory from a range of Android smartphones and smartwatches. In developing AMD, we analyzed the firmware update protocols of these devices by reverse engineering the Android bootloader. Based on this study, we have devised a method that allows access to main memory data through the firmware update protocols. Our experimental results show that AMD overcomes the usability constraints of previous main memory acquisition approaches and that the acquired main memory data of a smartphone or smartwatch can be accurately used in forensic investigations.

© 2017 Elsevier Ltd. All rights reserved.

Introduction

As of the second quarter of 2016, the Android operating system (OS) is the most widely used mobile OS, accounting for 87.6% share of the total smartphone OS market (Smartphone OS Market Share, 2016). In addition, development of various smartwatches based on the Android OS is expected to grow rapidly in the future (World Smartwatch Market, 2016). Although forensic research on smart devices has been actively conducted, most digital forensic commercial tools (Cellebrite UFED, 2017; Mobile Phone Examiner Plus, 2017; MSAB XRY, 2017; Oxygen Forensics, 2017) focus mainly on the acquisition and analysis of NAND flash data from smart devices. There is currently no support for forensic investigation of the main memory of smart devices. At the same time, the main memory of smart devices has increased in speed and capacity, and the prevalence of malware that resides only in main memory has also increased (Mobile Malware, 2017). Moreover, device manufacturers tend to use complex security functions such as Full-Disk Encryption (FDE) (Full-Disk Encryption, 2017), File-Based Encryption (FBE) (File-Based Encryption, 2017), Social Networking Service (SNS) conversation encryption (Facebook Messenger Secret Conversations, 2016; WeChat Message Cryptography, 2017; WhatsApp Security, 2017), KNOX (Samsung KNOX, 2017), and Secure Boot (Secure Boot, 2017). For these reasons, it will be difficult to rely only on NAND flash-based forensic technologies.

To overcome the above limitations, new advances in field-deployable main memory forensics are necessary, especially the technology to acquire main memory data from smart devices for use in forensic investigations. Currently, the technologies used for main memory data acquisition have several usability
limitations that make them difficult to apply in forensic inves-
tigations, and current research technologies on the subject are
hardly applied in practice because they do not solve the following
limitations.

First, a system restart may be required when acquiring main
memory data (Sylve et al., 2012; Taubmann et al., 2015). During
the process of replacing the kernel with a custom kernel or a custom
recovery image the system is restarted which causes a significant
loss of evidence.

Second, to overcome the limitation of kernel replacement, some
methods for acquiring main memory data work by injecting
executable code into a smart device (Yang et al., 2016). However, to
use this method, it is necessary to acquire root privilege in advance
(Rooting, 2016). In recent years, with the introduction of security
technologies such as Secure Boot and KNOX, root privilege escal-
ation has become just as difficult as acquiring main memory data.
Moreover, during root privilege escalation, the system may also
need to be restarted.

Third, an Android Debug Bridge-based protocol may be used to
acquire data by connecting to an Android smart device with a data
cable (Android Debug Bridge, 2017). However, if a screen lock (such
as a pattern lock, user passwords, etc.) is applied on a smart device
(and therefore USB debugging mode is disabled) access to main
memory and data acquisition are both impossible.

In this study, we have solved these limitations by providing
support for the physical acquisition of main memory data from
smart devices in forensic investigations. To this end, we have
investigated main memory acquisition based on the firmware up-
date protocols shared by smartphones and smartwatches. The
firmware update protocol is used to update to a new Android OS or
to patch software problems, and each manufacturer provides a
firmware update program to take advantage of the firmware update
protocol used in the Android bootloader (LG Software and tools
Download, 2017; Pantech Self-Upgrade, 2017; Samsung Kies,
2017; Samsung Odin, 2017).

We first analyzed the firmware update command by reverse
engineering the Android bootloader firmware update protocol. It is
important to note that the firmware update process runs only when
a smartphone enters a special mode called firmware update mode,
and it cannot run in normal operation mode. From analyzing these
firmware update programs, we have confirmed that it is possible to
switch from normal operation mode to firmware update mode
without a system restart through a software command. Moreover,
we have developed a way to access the main memory and obtain its
data during this firmware update process. In the case of Samsung
smartphones and smartwatches, we have found a vulnerability that
can access and acquire data in the main memory. In the case of LG
models, we have found that a main memory read command exists
in the manufacturer’s undocumented protocol, despite it not being
used in the firmware update process.

Based on these methods, we have developed AMD, a forensic
tool that provides automated acquisition of main memory data
from smartphones and smartwatches through their firmware up-
date protocols. The operation of AMD is fully automated, and AMD
presents a simple GUI interface for investigators to use (see Section
Android main memory dump tool). The following are the key
characteristics of the AMD acquisition method:

- It is possible to switch the device to firmware update mode for
  the acquisition of main memory data without a system restart
  through a software command.
- It is possible to acquire the main memory data through a
  firmware update command in firmware update mode without
  kernel replacement, without a device restart, and without root
  privilege.
- The proposed method does not use the Android ADB protocol
  because it acquires the data through the main memory read
  command. Therefore, it is affected by neither the USB debugging
  mode nor by a screen lock or user password.
- Our experiments show that smart device memory images ac-
  quired by AMD can be accurately used in forensic investigations.

Related work

There are currently no commercially available tools for
acquiring the main memory data of smart devices in forensic
studies. Because the Android system structure conforms to that of
the Linux operating system, most studies on main memory data
acquisition for Android have been performed by extending the
Linux-based data acquisition methods. The traditional Linux
methods for main memory acquisition can be performed by
software directly through the memory device file/dev/mem or by
using a Loadable Kernel Module (LKM) such as Fmem (How to
acquire memory from a running Linux system, 2017). However,
these acquisition methods cannot be applied to Android smart
devices because Android has recently blocked direct access to
/dev/mem and does not support ARM-based LKMs that target
Android.

Sylve et al. (2012) proposed the Linux Memory Extractor (LiME)
method (LiME, 2017). This method is the most widely used for
acquiring the main memory of Android smartphones, and it is an
LKM-based acquisition method like Fmem. However, to use the
LiME method, it is necessary to gain root privilege (persistent or
temporary root) to install the module. In some cases, it may also be
necessary to create a custom kernel to use LiME: In some Android
models, the default kernel does not allow inserting kernel modules
(the insmod/modprobe operation does not exist in the default
kernel). For such cases, a custom kernel is required. There are
techniques to compile a kernel without the original kernel source.
However, recently released Android models need the original
kernel source to compile a custom kernel because the kernel
configuration and toolchains must be exactly the same as the stock
firmware running on the device. Any change prevents the module
from being loaded due to a CRC checksum mismatch. Therefore,
the original kernel source code of the device is required for such
cases. Since a system restart may be required to obtain root privi-
leges and to use a custom kernel, there is a limitation in the
applicability of the LiME method to smart devices in forensic
investigations.

To overcome the limitations of custom kernel replacement,
recent research has proposed the AMExtractor tool for obtaining
main memory data, which converts the data acquisition function
into injectable code (AMExtractor, 2016; Yang et al., 2016). This
method has the advantage of not requiring a kernel module or the
original kernel source since it does not need to replace the cur-
current kernel with a custom kernel. However, to acquire main
memory data, this method requires a root privilege escalation in
advance. Unfortunately, because it is difficult to acquire root
privilege (due to the strengthening of security functions and the
fact that the system may need to be restarted during privilege
escalation), this method is hardly being adopted in forensic
investigations.

Similarly, the Bare Metal Application (BMA) method for
obtaining main memory data (Taubmann et al., 2015) leverages
recovery mode by uploading the acquisition code to the kernel area
of the Android recovery image. This method has the advantage of requiring neither a kernel module nor root privilege escalation. However, because it is necessary to create a custom recovery image, this method requires a recovery image file for the OS version of the device and a process for flashing the created custom recovery image. Since the system is restarted in this process, this method can result in significant loss of evidence. Recently, security technologies such as KNOX and Secure Boot have been strengthened, making it difficult to acquire data based on a custom recovery image.

The TrustDump method, a TrustZone-based memory acquisition mechanism, has been also been proposed [Sun et al., 2014]. This method can acquire the main memory data and CPU register values of a mobile device even when the operating system has crashed or has been compromised. However, this method is only supported on Freescale i.MX53 QSB, an embedded development board — making it an impractical tool for acquisition on commodity smart devices during forensic investigations.

Yang et al. [2015] proposed a NAND flash acquisition method based on analyzing the firmware update protocols of Android smartphones. They developed a physical acquisition method for Android smartphones by leveraging the NAND flash read command in the bootloader. However, this approach is hardly applicable to main memory acquisition. Since the NAND flash read commands are used during the firmware update process, they can be easily isolated. Thus, main memory data acquisition is impossible with this method because only a part of the main memory is read and the main memory read command is not used during the firmware update process. In this work we take an entirely new approach: we first performed reverse engineering to identify an unknown protocol that can access and dump main memory data during the firmware update process. This protocol is leveraged in our development of the AMD tool.

There are relevant studies on the acquisition of memory by analyzing Samsung SBOOT (Exploiting Android S-Boot, 2017; S BOOT dump, 2017). These approaches have been shown to be both robust and reliable, but they remain limited to only device models using the Samsung Exynos chipset — a small subset of the models that AMD supports.

Additionally, there are various studies to acquire NAND flash data from Android smartphones. Logical acquisition methods acquire user data stored on a smartphone via ADB Backup (Android Backup Extractor, 2014) or Content Provider (Hoog, 2011). Several rooting-based acquisition studies were introduced in scholarly works (Hoog, 2009; Lessard and Kessler, 2010). An acquisition method based on changing the custom recovery image has also been studied (Vidas et al., 2011; Son et al., 2013). The analysis of social networking applications (Mutawa et al., 2012), a prototype enterprise monitoring system for Android smartphones (Grover, 2013), and Kindle forensics (Hannay, 2011; Iqbal et al., 2013) were also studied. However, these methods only recover NAND flash memory and are not applicable to main memory acquisition, which involves different challenges such as requiring root privilege escalation, custom kernel replacement, or screen lock bypass.

In addition to physical dumping methods for acquiring the entire main memory data in Android, it is also possible to connect a smart device to a PC and then use ADB and Google Dalvik Debug Monitor Service (DDMS) tools to acquire and analyze the heap memory of Android apps (DDMS, 2017; HPROF, 2017). However, these methods are performed using the ADB protocol and, therefore, USB debugging mode must be enabled. Further, because this mode is disabled by default, it is impossible to apply these methods when a pattern lock or a user password are set. Moreover, the information that can be obtained with this method is very limited compared to that obtained with physical acquisition (i.e., the entire content of physical memory) methods.

There are hardware-based acquisition methods for acquiring main memory data using JTAG (Willassen, 2005; Guri et al., 2015). The acquisition is done using the JTAG debug interface that exists on the PCB of a smart device. Representative examples of hardware-based tools include Trace32 (Trace32, 2017), Riff Box (Riff box, 2017), and the ORT tool (ORT tool, 2017). These tools are mainly used by manufacturers to develop smart devices or to repair smart devices with hardware/firmware problems. However, not all smart devices support the JTAG interface, and there is a risk of damage when a smart device is disassembled; therefore, the applicability of these techniques in forensic investigations is quite limited.

Once collected, the Volatility tool is often used to analyze the physical image of Android’s main memory (Android volatility, 2016). It has supported the analysis of Android main memory since version 2.3. This tool is compatible with the image format dumped by the LiME method, which acquires separate system RAM areas of an Android device’s main memory; therefore, most acquisition tools create an image in the LiME format and use Volatility to validate the integrity of the acquired image. In our experiments, we use Volatility and a number of other metrics to verify the correctness of AMD’s memory images.

### Android main memory acquisition in firmware update mode

Android’s main memory is a module that corresponds to the main memory of a computer. When a user executes a program, the program fetches the related data from the NAND flash memory, uploads this data to main memory, executes its operations on that data, and then stores any persistent data back to the NAND flash memory. Therefore, the main memory contains evidence of program executions and actions that the user has performed. One key feature of evidence in main memory is that any encrypted data stored to NAND flash memory will remain decrypted in the main memory. Decrypted data such as the FDE key and user password can be obtained from main memory. Moreover, manipulated information in the kernel area, such as a hidden process, can be obtained from main memory data. However, since main memory is volatile, it only keeps data when the system is powered on, it loses the data if power is lost or if the system is restarted. Therefore, the physical acquisition of the main memory data should be performed as fast as possible in the live state, without a system restart of the smart devices in forensic studies.

**Main memory acquisition from Samsung devices**

In normal boot mode, there is no way to access the main memory. We analyzed the firmware update program, firmware update protocol, and firmware update process of Samsung to look for a method to access the main memory. The firmware update protocol is the only protocol that can access the memory directly by software, and we analyzed the commands that are used in the firmware update process and also analyzed how to access the main memory.

The firmware update mode and protocol are called Odin mode and Odin protocol, respectively. To acquire the main memory of smart devices in forensic studies, the device should be switched to Odin mode without a system restart. For this, we analyzed the
Odin protocol and found that software commands exist that can switch the device to Odin mode without a system restart. If the AT command “AT+FU?” and the string “ODIN” are sent in sequence from a PC to the Samsung smart device while connecting the device to the PC with a data cable, the smart device will return the string “LOKE” to the PC in the ACK message. If this process is successful, the Samsung smart device will switch to Odin mode. After this mode is entered, the firmware update process is performed.

Fig. 1 shows the firmware update process after the device switches to Odin mode. The 0x64 command is sent to the smart device to initialize the ODIN protocol communication process. After the initialization, it proceeds to partition the configuration for firmware update by using the 0x65 command/subcommands. Samsung manages every partition via the Partition Information Table (PIT) structure. PIT is an integral element of all firmware programs that contain the map of storage allocations for different system partitions. The firmware update process is executed according to the partition information through the 0x66 command/subcommands. When the firmware update process is completed, the Odin protocol is terminated via the 0x67 command/subcommands and the system is restarted to complete the firmware update process. During this process, we found a method for acquiring the main memory data in the process of reading the PIT partition information from a memory buffer (0x65).

**PIT read command: 0x65**

The PIT information is uploaded to the pit_buffer of the main memory according to the 0x65 command/subcommands, and it is read or written, and then finally written to the NAND flash memory. Table 1 shows the subcommands of 0x65.

<table>
<thead>
<tr>
<th>Command</th>
<th>Subcommands</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x65</td>
<td>0x00</td>
<td>—</td>
<td>Start writing the partition information</td>
</tr>
<tr>
<td>0x65</td>
<td>0x01</td>
<td>—</td>
<td>Start reading the partition information</td>
</tr>
<tr>
<td>0x65</td>
<td>0x02</td>
<td>Block number y</td>
<td>Read or Write the PIT from the main memory</td>
</tr>
<tr>
<td>0x65</td>
<td>0x03</td>
<td>—</td>
<td>Finish the read operation/write to flash memory</td>
</tr>
</tbody>
</table>

This command reads the partition information by dividing it into blocks of 500 bytes starting from the pit_buffer address in the main memory. For the Galaxy S2 model, the PIT size (pit_size) is 4096 bytes, and for models after Galaxy S2, the PIT size (pit_size) is 8192 bytes. In both cases, the size of the block that can be read from the main memory at once is 500 bytes. Therefore, in the case of the Galaxy S2 device, the area from 0 to 8 blocks after the pit_buffer address contains the PIT information. Similarly, for models after the Galaxy S2, the area from 0 to 16 blocks after the pit_buffer address contains the PIT information. Fig. 2 shows the process of reading the partition information from a Galaxy S2 model (CMD: 0x65; SUBCMD: 0x02).

**PIT read algorithm**

The IDA Pro tool (Hex-Rays, 2016) was used to decompile the PIT reading logic within the Samsung bootloader. Listing 1 shows the 0x65 command logic obtained from decompiling the Samsung bootloader. The reverse-engineered code of the PIT read command (CMD: 0x65; SUBCMD: 0x02) in Listing 1 is also represented in pseudocode steps in Table 2.

When the Odin program requests to read y blocks from the pit_buffer through the PIT read command, the send_address, that is,
the start address of each block, is calculated with Equation (1) using the address of \textit{pit\_buffer}. The variable \(y\) indicates the blocks number in the PIT section, and its value is also used to compute the \textit{send\_address} of the block. Equation (2) uses \textit{pit\_size} (4096 bytes for Galaxy S2 and 8192 bytes for models after Galaxy S2) to calculate how much memory should be read from \textit{send\_address}. The value is stored in \textit{send\_size}. Note that for the last block read this size will be less than 500 bytes. For Galaxy S2, the eighth block is 96 bytes, and for models after Galaxy S2, the 16th block is 192 bytes. If \textit{send\_size} from Equation (2) is greater than 500 bytes, then \textit{send\_size} is set to 500 according to Equation (3). Then, through Equation (4), \textit{send\_size} bytes are read from main memory at \textit{send\_address}. During the correct usage of this process, the partition information is read from the main memory.

\textit{Main memory acquisition}

As a result of analyzing the PIT read code in Section \textit{PIT read algorithm}, it was confirmed that a desired block of main memory can be read according to the value of \textit{send\_address} that is computed. Even if a block value outside of the \textit{pit\_buffer} area is specified as a parameter in the PIT read command, the main memory data of the corresponding area can be acquired. Fig. 3 shows the values of \textit{send\_address} and \textit{send\_size} according to the value of \(y\) (the block number) for the entire main memory of the Galaxy S2 model. The addresses are calculated via the PIT read algorithm in Table 2.

In this way, the main memory can be divided into A, B, and C areas. Area A indicates the area above the PIT area in main memory, area B is the PIT area, and area C represents the addresses below the PIT area in the main memory. To obtain the main memory data above the PIT area (area A), it is necessary to know the block number \(y\) of the physical main memory starting address. For this, we can compute the starting block number using the start address of the \textit{pit\_buffer} (the top of area B) and the physical start address of the main memory. For example, in the case of a Samsung device, the physical start address of the main memory is 0x40000000.

However, the start address of the \textit{pit\_buffer} is set when the Odin protocol is initialized, it differs from device to device, and changes at every boot — so it cannot be known \textit{a priori}. By further analyzing the protocol, we have found how to calculate the start address of the \textit{pit\_buffer} via the following two steps:

\begin{verbatim}
unsigned __int8 __fastcall process_packet_101_set_partition(unsigned __int8 *result) {
    int v_len; // ST0C_48?
    int v2; // r39
    int v3; // r8015
    int v4; // r1015
    int v5; // r2015
    int v_cmd; // [sp+8h] [bp-14h]@1
    int v7; // [sp+10h] [bp-Ch]@9
    signed int v8; // [sp+14h] [bp-8h]@3
    {
        case 0:
            pit_loaded = 0;
            result = (unsigned __int8 *)upload_ack(v_cmd, 0);
            break;
        case 1:
            pit_loaded = 1;
            v8 = v_cmd;
            read_part_info_dit.pit_buffer();
            if { check_pit_integrity(dit.pit_buffer) != 1 }
                v8 = -1;
            result = (unsigned __int8 *)upload_ack(v8, 4096);
            break;
        case 2:
            if { pit_loaded ) // PIT read command (CMD:0x65, SUBCMD: 0x02)
            {
                if { pit_loaded == 1 }
                {
                    v2 = -500 * v7 + 4096;
                    if { v2 >= 500 }
                        v2 = 500;
                        result = (unsigned __int8 *)upload(dit.pit_buffer + 500 * v7, v2);
                }
                else
                {
                    upload_ack(v_cmd, 0);
                    download(char_4EE00000[18874368], v_len);
                    FillChar(dit.pit_buffer, 0, 4096);
                    t_copy_data(dit.pit_buffer, (int)char_4EE00000[18874368], v_len);
                    result = (unsigned __int8 *)upload_ack(v_cmd, 0);
                }
                break;
    }
}
\end{verbatim}

\textit{Listing 1. Code for reverse engineering of the 0x65 command in the bootloader.}
Step 1 First, we must acquire the `pit_buffer` information. The start address of the `pit_buffer` is calculated using the last 16 bytes of the –1 block, which is the block immediately preceding the `pit_buffer`. Luckily, we can acquire the contents of –1 block through the PIT read command.

Step 2 Using the –1 block, we can calculate the `pit_buffer` start address. Fig. 4 shows the process of calculating the start address of the `pit_buffer` for the Galaxy S3 device. The four 4-byte values stored in the 16 bytes of the –1 block correspond to the values of PIT_Size, PIT_Alloc, PIT_Prev, and PIT_Next, respectively. Among them, the address of `pit_buffer` can be obtained using PIT_Size and PIT_Next. The value of PIT_Next means the next address at the end of the PIT, and PIT_Size means the size of the PIT. Therefore, by subtracting PIT_Size from PIT_Next, the algorithm can calculate the start address of the `pit_buffer`.

\[
\text{pit_buffer start address} = \text{PIT\_Next} - \text{PIT\_Size}
\]

The structure of the –1 block is different across the Samsung models. The `pit_buffer` start addresses of the Galaxy S4 and Galaxy Note 3 models are set to the –1 block, unlike the Galaxy S3 model. Fig. 5 shows the `pit_buffer` start address of the Galaxy S4 that we tested.

When the \( y \) value of the start block in the main memory is obtained, the main memory data above the PIT area can be dumped. The values of `send_address` and `send_size` are calculated according to the algorithm in Table 2 even if the \( y \) block value of area A has a negative value. As shown in Fig. 3, even if a negative \( y \) block is requested by the PIT read command, the data acquisition is performed via 500 byte blocks from the calculated `send_address`.

The main memory data of the PIT area (area B) can be acquired from the zeroth block directly via the PIT read command. To acquire the main memory data below the PIT area (area C), both `send_address` and `send_size` are calculated according to the algorithm shown in Table 2, and therefore `send_size` becomes a negative value. As shown in Fig. 3, `send_size` has a negative value of –404. There is no part of the PIT read algorithm that handles the case where `send_size` is negative. Therefore, a value of –404 in `send_size` is recognized as an unsigned value, and as a result, a value of –404 is calculated as 4,294,966,892 and a command to dump that size is executed. In this case, the algorithm will read from the specified block to the end of the main memory if a command to read block 9 is sent. After reading to the end of the main memory, the command performs no more read operations. Therefore, the acquisition of the main memory data below the PIT area is performed by a single read operation.

---

**Table 2**

<table>
<thead>
<tr>
<th>Step</th>
<th>Equation</th>
<th>Pseudocode</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>( \text{send_address} = \text{pit_buffer} + 500*\text{y} )</td>
<td><code>send_address = pit_buffer + 500*y;</code></td>
</tr>
<tr>
<td>(2)</td>
<td>( \text{send_size} = \text{pit_size} - 500*\text{y} )</td>
<td><code>send_size = pit_size - 500*y;</code></td>
</tr>
<tr>
<td>(3)</td>
<td>if <code>send_size</code> &gt; 500 then <code>send_size</code> = 500</td>
<td><code>if send_size &gt; 500 then send_size = 500;</code></td>
</tr>
<tr>
<td>(4)</td>
<td>upload <code>(send_address, send_size)</code></td>
<td><code>upload (send_address, send_size);</code></td>
</tr>
</tbody>
</table>
Fig. 3. Values of send_address and send_size according to y block values (Galaxy S2).

Fig. 4. The process of calculating the pit_buffer start address (Galaxy S3).

Fig. 5. The pit_buffer start address (Galaxy S4).
As a result, in the case of a Samsung smart device, access to both the entire main memory area and data acquisition is possible using the PIT read command. Fig. 6 shows an example of the main memory data acquisition from the Galaxy S3 model.

Main memory acquisition in LG devices

The LG firmware update program and firmware update protocol were reverse engineered and then analyzed for main memory access. In the case of LG, the firmware update mode and protocol are called download mode and download protocol, respectively. The download protocol was analyzed by decompiling the LG bootloader (SBL3).

The commands used in download mode are LG’s own implementation and follow the frame structure of the High-Level Data Link Control (HDLC) protocol (High-Level Data Link Control, 2017), which starts with the HDLC flag (0x7E); followed by a 1-byte command, variable-sized data, and 2 bytes of CRC-16; and ends with the HDLC flag (0x7E). If download commands are sent to a smart device, the ACK (0x02) or NAK (0x03) reply packet is sent.

In the case of the LG model, the device should be switched to download mode without a system restart. In the analysis of the download protocol, we found software commands that could switch the device to download mode without a system restart. If the “0x3A” command is sent from a PC to a smart device while connecting the LG smart device to the PC with a data cable, the smart device switches to download mode.

As a result of reverse engineering the LG bootloader using IDA Pro, we were able to analyze LG download commands that are not available to the public. These include commands implemented by LG for software debugging purposes as well as commands used for firmware updates. From our analysis of the SBL3 bootloader, we found that there is a main memory read command that can access the main memory and acquire its data. These commands are expected to be used for memory debugging purposes. Fig. 7 shows the main memory read command which was found by decompiling the SBL3 bootloader in the LG Optimus G pro model. After the device switches to download mode, we can acquire the entire main memory data using the main memory information acquisition commands and main memory read commands. Table 3 shows the commands used for the main memory data acquisition in LG.

Main memory information command: 0x10

After the device enters the download mode, the command for main memory information acquisition (0x10) is sent to the smart
device to obtain information about main memory. The obtained information includes the start address and the size of the main memory connected to the External Bus Interface (EBI) (External Bus Interface, 2015). It also contains debug information that can be used for showing the CPU register values and for memory debugging. Fig. 8 shows the process of acquiring the main memory information of the Optimus G Pro model.

**Main memory read command: 0x12, 0x14**

Based on this information, we can obtain the main memory data using the main memory read command. If the size of the main memory is 2 GB (from the obtained main memory information), the main memory data can be obtained using the 0x12 read command. The main memory information also reports that the start address of the memory is 0x80000000. When the read commands are sent to the smart device, the dump data (4 KB segments per command) are sent to the PC. Similarly, if the size of main memory is reported to be 1 GB, the main memory data can be obtained using the 0x14 read command. In this case, the main memory information reports that the start address of the memory is 0x40000000. Fig. 9 shows the format of the main memory read command and the process of acquiring the main memory data of the Optimus G Pro model.

**Main memory image**

The acquisition method proposed in this study can physically acquire the entire main memory, and the acquired physical image can be used for forensic analysis. For this, we used the Volatility tool to analyze the dumped images acquired by AMD. Volatility supports the analysis of a main memory image acquired through the LiME method. Therefore, we implemented a function in AMD to convert the acquired data to the LiME image format. LiME creates an acquisition file according to the system RAM areas identified in the /proc/iomem structure. The format of the main memory file acquired by LiME consists of a header that stores the physical address information of each system RAM area. Fig. 10 shows the LiME header structure. The entire main memory area obtained by AMD is converted to the LiME-based image format by attaching LiME headers according to the system RAM areas in /proc/iomem.

**Android main memory dump tool**

We developed the Android Main memory Dump (AMD) tool to acquire the main memory data of smart devices in forensic investigations, as described in Section Android main memory acquisition in firmware update mode. This tool was developed using the C++ language. After a smart device is connected to a PC using a data cable, the tool provides a simple GUI environment that allows a user to perform main memory acquisition by clicking on the buttons. The tool was developed so that even a user who is not a professional forensic expert can easily and quickly acquire main memory data for use in forensic investigations. Fig. 11 shows the process of acquiring the main memory data of a Galaxy Gear, a Samsung smartwatch, using the AMD main memory data acquisition tool.

**Main memory data acquisition**

The current AMD prototype supports physical memory data acquisition for Samsung smartphones and smartwatches and LG smartphones. As of writing, we have verified that AMD can correctly acquire a main memory image from the 50 device models listed in Table 4. In the future, we hope that our ongoing research will continue to expand this list.

Before connecting a smart device to a PC using a USB data cable, the USB driver must be installed in advance. After connecting the USB data cable, the user should click the button “Switch to FU mode.” This sends download-mode-switching commands to the smart device which switches the device into download mode. After the switch to download mode, the tool obtains the main memory information using the main memory information commands and outputs the information to the Partition Information window.

When the user clicks the “Full Dump” button, the entire physical acquisition process is performed from the main memory start address to the end address using the Samsung PIT read command or the LG main memory read command. Once the acquisition

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3A</td>
<td>Switch to download mode</td>
</tr>
<tr>
<td>0x10</td>
<td>Get main memory information</td>
</tr>
<tr>
<td>0x14</td>
<td>Read main memory (1 GB)</td>
</tr>
<tr>
<td>0x12</td>
<td>Read main memory (2 GB)</td>
</tr>
<tr>
<td>0x0A</td>
<td>Reset system</td>
</tr>
</tbody>
</table>

**Table 3** Commands used for the main memory data acquisition in LG.

![Image of PC to Phone and Phone to PC diagrams]
process is completed, the Dump Log window shows the investigator information, acquisition tool information, and acquired image information, as shown in Fig. 11. The acquired image information includes the acquisition start time, acquisition image storage location, and acquisition end time, and the MD5 hash value of the imaged file. If the "Stop Dump" button is clicked during the acquisition process, the process is stopped. After acquisition completes, the system can be restarted by clicking the "Reboot" button. Note that AMD makes no persistent changes to the device, but a reboot is required to continue using the device in normal operation mode.

Conversion to the LiME image format

The image file acquired with the AMD tool has a raw data format that represents the physical acquisition of the entire main memory area. To more easily analyze the image file acquired by the AMD tool, when the user can click the button "Convert to LiME format", the entire main memory image file acquired by AMD is converted to the LiME image format.

Experiments

The requirements for the physical acquisition of the main memory data of a smart device in forensic investigations are as follows. First, the main memory data acquisition process should be performed without a system restart for the smart device. Second, it must support a rapid evidence collection and an integrity check of the acquired image. Third, the analysis of the acquired image must be provided using standard forensic analysis tools. Fourth, it should be possible to extract evidence of the user’s actions from the acquired image. With these requirements, we compared the AMD acquisition method proposed in this study with existing research methods, namely, the LiME, AMExtractor, and BMA methods. Table 5 shows the experimental results obtained. The experimental results below were obtained by averaging across 5 acquisitions per device. From the 50 device models (shown in Table 4) that we have verified are supported by AMD, we selected 12 representative models for these experiments. The 12 representative models are: SHV-E210S, SHV-E210K, SHV-E330S, SHV-E330K, SM-N900S, SM-N9005, LG-F180S, LG-F180L, LG-F240S, LG-F240K, LG-E960, SM-V700.

Live acquisition

We performed live main memory acquisition experiments using the existing methods and AMD, as shown in Table 5. LiME requires root privilege to acquire main memory data. A custom kernel and original kernel source code are only necessary for cases where the default kernel does not allow inserting kernel modules and the kernel configuration and toolchains must be exactly the same as the stock firmware running on the device, respectively. During our evaluation, we tested a number of LG devices (Optimus G, Optimus G Pro, Nexus 4) which require flashing a custom kernel and the original kernel source to use LiME. Moreover, because LiME uses the ADB protocol, it is impossible to acquire the data from a smart device with a screen lock (USB debugging mode disabled). The AMExtractor method also requires root privilege and uses the ADB protocol to upload the dump code to the smart device. It cannot access the main memory of a smart device that has a screen lock. The BMA method requires the original recovery image and a process to flash a custom recovery image, which needs to restart the system and therefore limits its application in forensic investigations. AMD can acquire data through the main memory read command — after switching the device to firmware update mode through a software command without a system restart. In this way, AMD solves the limitations of existing methods such as obtaining a root privilege, replacing with a custom image, and bypassing the screen lock. Even in the case of a smart device with a screen lock, since the acquisition process is performed by switching the device to firmware update mode, the proposed method can still acquire a main memory image.

Evaluation of AMD

We verified the integrity of the images acquired with the AMD tool. In these experiments, main memory images were acquired with LiME first and then AMD. Both LiME and AMD recovered a
The total of 3,870,712 pages. The memory contents dumped by LiME and AMD were nearly the same. Both techniques collected the same number of pages from the device and 99.4% (Identical pages are 3,847,487) of those pages were identical. However, several pages are not identical. This occurs because some code and data areas of the kernel are changed while switching to the firmware update mode. By comparison, the loading of the LiME module also modifies small portions of kernel code and data during module insertion, but LiME’s modifications are not reflected here.

In order to show the performance of the AMD tool, the acquisition times of 12 Android models using the AMD tool are shown in Table 6.

Volatility analysis

In addition to comparing the contents of the pages in LiME and AMD memory images, we also verified that the evidence recovered by Volatility plug-ins from AMD memory images is correct. Before collecting an AMD memory image, we took note of the following information which would serve as the ground truth for the Volatility plugins’ results: 1) the process list entries for the linux_pslist and linux_psscan plugins, 2) the output of procfs for the linux_iomem plugin, and 3) the memory maps of the processes for the linux_proc_maps plugin. We then collected a memory image with AMD and verified that the results of those plugins correctly matched our ground truth. The results were as follows: linux_pslist reported 194 entries, linux_iomem found 108 entries, linux_psscan located 2,596 entries (which we manually verified were all the live processes as well as valid previously-killed processes), and linux_proc_maps recovered 26,968 entries. These results confirm that AMD correctly images the data in main memory.

Table 4
Models that we have verified are supported by the AMD tool.

<table>
<thead>
<tr>
<th>Model</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Galaxy S2</td>
<td>SHW-M250S</td>
</tr>
<tr>
<td>Galaxy S3</td>
<td>SHV-E210S</td>
</tr>
<tr>
<td>Galaxy S4</td>
<td>SHV-E330S</td>
</tr>
<tr>
<td>Galaxy Note 2</td>
<td>SHV-E250S</td>
</tr>
<tr>
<td>Galaxy Note 3</td>
<td>SM-N900S</td>
</tr>
<tr>
<td>Galaxy Grand</td>
<td>SHV-E270S</td>
</tr>
<tr>
<td>Galaxy Pop</td>
<td>SHV-E220S</td>
</tr>
<tr>
<td>Galaxy Note 8</td>
<td>SHV-M500W</td>
</tr>
<tr>
<td>Galaxy Note 10</td>
<td>SM-P9600</td>
</tr>
<tr>
<td>Galaxy Gear</td>
<td>SM-V700</td>
</tr>
<tr>
<td>Optimus G</td>
<td>LG-F180S</td>
</tr>
<tr>
<td>Optimus G Pro</td>
<td>LG-F220K</td>
</tr>
<tr>
<td>Optimus Gx</td>
<td>LG-F310L</td>
</tr>
<tr>
<td>Optimus Gx Pro</td>
<td>LG-F240S</td>
</tr>
<tr>
<td>Optimus Pad 8.3</td>
<td>LG-V500</td>
</tr>
<tr>
<td>Optimus Vu1</td>
<td>LG-F100S</td>
</tr>
<tr>
<td>Optimus Vu2</td>
<td>LG-F200S</td>
</tr>
<tr>
<td>Optimus LTE1</td>
<td>LG-J62000</td>
</tr>
<tr>
<td>Optimus LTE2</td>
<td>LG-F160S</td>
</tr>
<tr>
<td>Optimus LTE3</td>
<td>LG-F200S</td>
</tr>
<tr>
<td>Nexus 4</td>
<td>LG-E960</td>
</tr>
</tbody>
</table>
Acquisition of user information

Recently, FDE, FBE, and SNS conversation encryption technologies have been applied to protect NAND flash data. In this case, even if the NAND flash data are acquired, the data encryption cannot be deciphered by existing NAND flash forensic technologies. To solve this problem, it is necessary to extract important user information such as the FDE master key, user conversation messages, etc., from the main memory where they remain decrypted. However, since the main memory is a volatile memory, its data are only retained when the system is powered on and are lost if the power is lost or if the system is restarted. AMD can extract evidence left by the user as is because it can obtain the entire main memory data from a smart device without a system restart. The decrypted FDE master key, user passwords, and decrypted SNS conversation message from the image obtained by the proposed method can be then extracted.

Fig. 12 shows a portion of an image acquired with the AMD tool. This shows that the user’s data (such as FDE keys and SNS conversation messages) remain intact in the image acquired by the AMD method. In the case of recovering the FDE master key, after encrypting the Android NAND flash, we dumped the main memory with the AMD tool. We extracted the FDE master key (shown in Fig. 12) from the acquired image by searching for the cipher identification string “AES-CBC-ESSIV:SHA256”. After acquiring the

Table 5
Experimental results.

<table>
<thead>
<tr>
<th>Requirements</th>
<th>AMD</th>
<th>LiME</th>
<th>AMExtractor</th>
<th>BMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root privilege</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Original kernel source</td>
<td>No</td>
<td>Some Models(^a)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Process for flashing of custom image</td>
<td>No</td>
<td>Some Models(^b)</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Prior disabling of screen lock or user password</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

\(a\) Older Android models may not require original kernel source because only modern Android firmware enforces CRC checks.

\(b\) Creating and flashing a custom kernel is necessary when the default kernel does not support inserting kernel modules.

Table 6
Acquisition time of main memory data by using AMD tool.

<table>
<thead>
<tr>
<th>Model</th>
<th>Name</th>
<th>OS Version</th>
<th>Size</th>
<th>Acquisition time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Galaxy S3</td>
<td>SHV-E210S</td>
<td>4.1.2</td>
<td>2 GB</td>
<td>23 min</td>
</tr>
<tr>
<td>Galaxy S4</td>
<td>SHV-E210K</td>
<td>4.4.4</td>
<td>2 GB</td>
<td>23 min</td>
</tr>
<tr>
<td>Galaxy Note 3</td>
<td>SM-N900S</td>
<td>4.3</td>
<td>3 GB</td>
<td>52 min</td>
</tr>
<tr>
<td></td>
<td>SM-N900S</td>
<td>4.4.2</td>
<td>3 GB</td>
<td>52 min</td>
</tr>
<tr>
<td>Optimus G</td>
<td>LG-F180S</td>
<td>4.4.4</td>
<td>2 GB</td>
<td>13 min</td>
</tr>
<tr>
<td>Optimus G Pro</td>
<td>LG-F240S</td>
<td>5.0.1</td>
<td>2 GB</td>
<td>12 min</td>
</tr>
<tr>
<td>LG-F240K</td>
<td>5.0</td>
<td>2 GB</td>
<td>12 min</td>
<td></td>
</tr>
<tr>
<td>Lexus 4</td>
<td>LG-E960</td>
<td>5.1.1</td>
<td>2 GB</td>
<td>13 min</td>
</tr>
<tr>
<td>Galaxy gear</td>
<td>SM-V700</td>
<td>4.1</td>
<td>512 MB</td>
<td>26 min</td>
</tr>
</tbody>
</table>

Acquisition of user information

Recently, FDE, FBE, and SNS conversation encryption technologies have been applied to protect NAND flash data. In this case, even if the NAND flash data are acquired, the data encryption cannot be deciphered by existing NAND flash forensic technologies. To solve this problem, it is necessary to extract important user information such as the FDE master key, user conversation messages, etc., from the main memory where they remain decrypted. However, since the main memory is a volatile memory, its data are only retained when the system is powered on and are lost if the power is lost or if the system is restarted. AMD can extract evidence left by the user as is because it can obtain the entire main memory data from a smart device without a system restart. The decrypted FDE master key, user passwords, and decrypted SNS conversation message from the image obtained by the proposed method can be then extracted.

Fig. 12 shows a portion of an image acquired with the AMD tool. This shows that the user’s data (such as FDE keys and SNS conversation messages) remain intact in the image acquired by the AMD method. In the case of recovering the FDE master key, after encrypting the Android NAND flash, we dumped the main memory with the AMD tool. We extracted the FDE master key (shown in Fig. 12) from the acquired image by searching for the cipher identification string “AES-CBC-ESSIV:SHA256”. After acquiring the
NAND flash memory, it could directly be decrypted with the extracted FDE master key. We then confirmed that the decrypted NAND flash memory image could be correctly analyzed by the Cellebrite forensic tool.

In the case of the Facebook user conversation messages, we searched for the messages that had been sent to the smartphone within the memory image acquired by the AMD method. We then confirmed that the conversation messages remain intact within the acquired image. This result highlights the fact that the AMD method acquires the main memory data without restarting the system.

The FDE master key and Facebook conversation messages are encrypted when stored to NAND flash memory. So there are only brute-force encryption-breaking methods to recover their decrypted values. The existing NAND flash forensic technologies cannot solve these issues. However, since they are decrypted in main memory, the data can be acquired with AMD.

Conclusion

Existing research on acquiring Android main memory data are hardly applied in practice because they do not solve the constraints of root privilege escalation, custom kernel replacement, and screen lock bypass. To solve this problem, we proposed AMD, a tool for acquiring the main memory data of smart devices, which can easily be deployed in forensic investigations. In designing AMD, we reverse engineered the firmware update program, firmware update protocol, and firmware update process provided by the manufacturers. From the results of this analysis, we found that physical acquisition of all data in main memory is possible after the device is switched to firmware update mode without a system restart. Based on these findings, we developed AMD to support the acquisition of the main memory from Android smartphones and smartwatches.

Comparison to existing acquisition methods showed that the proposed method can acquire main memory data without a system restart, root privilege escalation, custom kernel, and screen lock bypass. We confirmed that, from the obtained image, the evidence stored in the main memory that the user had left can be extracted. We also verified the integrity of the acquired image and confirmed that it can be further analyzed through the Volatility tool.

References