Chapter 18

DATA RECOVERY FROM WINDOWS CE BASED HANDHELD DEVICES

Antonio Savoldi and Paolo Gubian

Abstract Data hiding creates serious problems for digital forensic practitioners attempting to recover evidence. It is possible to conceal large amounts of sensitive data in handheld devices in a manner that prevents their recovery using standard forensic tools. This paper describes a technique for recovering data stored in the slack memory of Windows CE based devices. A case study involving data hiding in a Toshiba E740 PDA is discussed.

Keywords: Data recovery, handheld devices, Windows CE, Toshiba E740 PDA

1. Introduction

Personal digital assistants (PDAs) and cell phones are the most pervasive pieces of electronic equipment in modern society. These devices contain a wealth of information of evidentiary value – subscriber data, call data, contact lists, SMS and email messages, images, audio and video files, as well as sensitive data concealed by exploiting weaknesses in the operating system and/or hardware. Data can be hidden in a variety of ways, usually for illicit purposes. Two common techniques involve hiding data in images, audio or video files using steganography and allocating sensitive data in the slack memory of electronic devices [11, 12].

Covert channels [14] are frequently used for the surreptitious transfer of sensitive data in a manner that violates the security policy of a computer system. In the case of a storage channel, data is transferred from one party to another by writing to shared storage; a timing channel signals sensitive data by modulating temporal system resources. Due to their popularity and functionality, PDAs and cell phones are attractive devices for implementing storage channels. It is common to find such communications devices with 256 MB RAM and 128 MB flash ROM, var-

Please use the following format when citing this chapter:

Savoldi, A. and Gubian, P., 2008, in IFIP International Federation for Information Processing, Volume 285; *Advances in Digital Forensics IV*; Indrajit Ray, Sujeet Shenoi; (Boston: Springer), pp. 219–230.

ious built-in wireless capabilities (Wi-Fi, Bluetooth, IrDa, GSM, UMTS, HSDPA) along with a high resolution camera and a GPS receiver. Large amounts of data can be hidden in these handheld devices in a manner that prevents their recovery using standard forensic tools.

This paper describes techniques for data concealment and recovery from devices running Windows CE (WinCE) [9], one of the most popular operating systems for handheld devices. A case study involving data hidden in the slack memory of a Toshiba E740 PDA is presented. Also, guidelines are provided to assist digital forensic practitioners in identifying and recovering hidden data in WinCE devices.

2. Background

This section describes Windows CE and the Toshiba E740 PDA used in our case study.

2.1 Windows CE Operating System

Windows CE [9], often referred to as WinCE, is a modular operating system, which serves as the foundation for several classes of embedded devices. It is supported by Intel Xscale processors and compatibles, and MIPS, ARM and Hitachi SH processors. WinCE is optimized for devices with minimal storage and small scale factors (small-scale digital devices); its kernel requires less than 1 MB of memory. WinCE devices are often configured without any disk storage and may be configured as closed systems, with the operating system burned on a flash ROM. WinCE is compliant with the definition of a real-time operating system with deterministic interrupt latency. It supports 256 priority levels and uses priority inheritance to deal with priority inversion. Furthermore, WinCE is a multitasking operating system, where the fundamental unit of execution is a "thread." Since the first edition of WinCE (called Pegasus) was released in 1996, the operating system has evolved to support platforms other than handheld devices. The basic WinCE core is used in AutoPC, PocketPC 2000/2002, Mobile 2003, Mobile 2003 SE, Mobile 5.0/6.0, Smartphone 2002/2003 and many other embedded systems and industrial devices.

The WinCE kernel uses a paged virtual memory system to manage and allocate program memory. The virtual memory system provides contiguous blocks of memory, between 1 KB and 4 KB within 64 KB regions, so that applications do not have to deal with memory allocation. In a WinCE device, the operating system and the applications bundled with the operating system are stored in ROM. The entire operating system is mapped to a binary ROM image divided logically into two types of modules. The first type corresponds to executable in place (XIP) modules; these modules save RAM space and reduce the time needed to start applications. The second type includes compressed modules, which are decompressed by the operating system and paged into RAM before execution.

In WinCE devices, the RAM is divided into two regions, "object store" and "program memory." The object store resembles a permanent, virtual RAM disk. Data in the object store is retained when the system is suspended or when a soft reset operation is performed. Normally, devices have a backup power supply for the RAM to preserve data when the main power supply is interrupted. When operations resume, the system searches for a previously-created object store in RAM and uses it (if one is found). Devices without battery-backed RAM may use a special flag in the registry to preserve data during multiple boot processes.

The remaining portion of the RAM on a WinCE device is designated for program memory. This space holds various stacks and heaps belonging to executing applications.

WinCE has a virtual memory address space of 4 GB. The operating system is able to manage at most 32 processes by assigning a "slot" corresponding to 32 MB of virtual address space to each process. This is partly due to the fact that Windows CE keeps the address spaces of all processes available at all times, even when the processes are not running. Thus, the lower portion of the address space is split into 32 MB slots. The address space is divided as follows (note that 32 MB corresponds to `0x02000000` in hexadecimal code):

- Slot 0 is assigned the memory locations in the range `0x00000000` to `0x01FFFFFF`.

- Slot 1 is assigned the memory locations in the range `0x02000000` to `0x03FFFFFF`.

- Slot 31 (last slot) ends at memory location `0x41FFFFFF`.

- Memory locations in the range `0x42000000` to `0x7FFFFFFF` mostly correspond to the "shared area" used for `VirtualAlloc` functions and memory-mapped files.

- Memory locations above `0x80000000` are reserved for the kernel. The kernel and the DLLs that load into the kernel (e.g., installable interrupt service routine (ISR) DLLs) execute from this memory space.

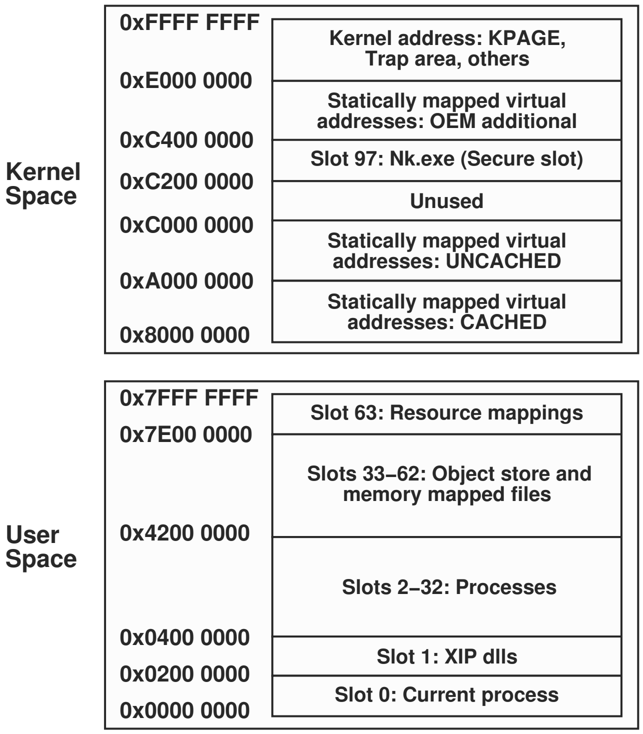
Kernel Space
0xFFFF FFFF
Kernel address: KPAGE, Trap area, others
0xE000 0000
Statically mapped virtual addresses: OEM additional
0xC400 0000
Slot 97: Nk.exe (Secure slot)
0xC200 0000
Unused
0xC000 0000
Statically mapped virtual addresses: UNCACHED
0xA000 0000
Statically mapped virtual addresses: CACHED
0x8000 0000
User Space
0x7FFF FFFF
Slot 63: Resource mappings
0x7E00 0000
Slots 33–62: Object store and memory mapped files
0x4200 0000
Slots 2–32: Processes
0x0400 0000
Slot 1: XIP dlls
0x0200 0000
Slot 0: Current process
0x0000 0000

Figure 1. Virtual address space managed by WinCE.

Figure 1 shows the layout of the virtual memory managed by WinCE. Note that the kernel and user space each have 2 GB of addressable memory.

The Remote Application Program Interface (RAPI) protocol [8] is often used by tools to extract the ROM and RAM contents of WinCE devices. The RAPI library enables applications running on a desktop computer to perform actions on a remote WinCE device; these include manipulating the file system on the remote device (e.g., creating and deleting files and directories). RAPI interfaces can be used to create and modify databases, either in the object store or in mounted database volumes. RAPI applications can also query and modify registry keys as well as launch applications and invoke methods on the remote device.

2.2 WinCE Test Device

A Toshiba E740 PDA equipped with the PocketPC 2002 OS (a WinCE derivative) was used in our investigation of data hiding and recovery at the firmware level. It has an Intel PXA240 (400 Hz) processor, 64 MB of SDRAM (main memory) and 32 MB of CMOS flash memory (ROM), which holds the operating system. The device also has built-in Wi-Fi and IrDa transceivers. Two slots for secure digital and compact flash cards are available for memory expansion. Later in this paper we will demonstrate the ease with which data can be hidden in the ROM and RAM in a manner that precludes its recovery using commercial digital forensic tools.

2.3 Data Extraction Techniques

The two main classes of data extraction techniques are logical extraction and physical extraction. A logical extraction technique focuses only on the visible content at the file system level, i.e., data pertaining to files, databases and registry along with other file system data. Device Seizure [10] is a popular logical data extraction tool for PDAs and cell phones (although it can access some physical data from certain devices).

A physical extraction technique, on the other hand, is attractive because it can recover all the data stored in an electronic device. In most cases, however, only the flash ROM and the RAM content are recovered using a special operating mode of the device (e.g., Palm OS debugger mode) or by communicating with the operating system (e.g. using the RAPI protocol [8]).

According to Breeuwsma and co-workers [2], three techniques may be used to obtain a complete copy of flash memory: (i) using "flasher" tools, (ii) using JTAG test access ports, and (iii) using forensic de-soldering.

Flasher tools are designed to copy the memory of certain families of electronic devices. They employ APIs that interact with the addressable memory. Generally, these tools originate from manufacturers, who use them for debugging purposes, or they come from the hacker community, which creates the tools to modify the functionality of handheld devices. An important advantage of this technique is that flash memory can be imaged without de-soldering the chip. However, many flasher tools do not make complete forensic copies of flash memory, mostly because of the limited functionality of the API provided by the embedded device. Furthermore, it is important to acknowledge Locard's Exchange Principle [3] in that a data extraction process executing in device memory can potentially affect the integrity of the memory. Gershteyn and co-workers [4] have used flasher tools to recover hidden data from BIOS chips. Savoldi and Gubian [11] have used similar tools to extract data from SIM/USIM cards.

The second physical extraction method involves the use of JTAG test access ports of embedded devices. JTAG ports in most devices are designed for debugging purposes, but they can also be used to access the flash memory [1]. The JTAG extraction technique is complex and time consuming; however, it is possible to guarantee that no data is written to memory during the data recovery phase.

The third physical extraction technique is to de-solder the memory chip and use a chip programmer or reader to extract the data. This method is expensive, time consuming and the most invasive; however, it can be used to recover data from damaged devices.

3. Data Extraction Methodology

This section discusses the use of open source tools based on the RAPI protocol [8] for acquiring the binary ROM image and major portions of the RAM of a WinCE device. The software-based approach falls in the category of using flasher tools. It can be used to extract data in a non-invasive manner from a variety of WinCE devices.

Our experiments employed a set of open source tools [6] based on the RAPI and ActiveSync protocols. Two tools, `pmemdump` and `pmemmap`, are particularly useful.

The `pmemdump` tool is very effective at extracting ROM and RAM data. To use the tool, it is necessary to copy a DLL library to the device file system. The following options are provided by `pmemdump`:

```
Usage: pmemdump [ -m | -p procname | -h prochandle] start length
            [ filename ]
    numbers can be specified as 0x1234abcd
    -1 -2 -4 : dump as bytes/words/dwords
    -w NUM   : specify number of words per line
    -s SIZE  : step with SIZE through memory
    -a       : ascdump iso hexdump
    -f       : full -- do not summarize identical lines
    -c       : print raw memory to stdout
    -x       : print only hex
    -xx      : print only fixed length ascii dumps
    -v       : verbose
    -n NAME  : view memory in the context of process NAME
    -h NUM   : view memory in the context of process with handle NUM
    -m       : directly access memory -- not using ReadProcessMemory
    -p       : access physical memory instead of virtual memory
    if -p, -h and -m are not specified, memory is read from the
    context of rapisrv.exe
```

By specifying the virtual starting address (in hexadecimal notation) with the length of the memory block, it is possible to obtain, for example, the entire ROM image (32 MB). This is saved in the file `rom_pda.bin` as follows:

```
      pmemdump.exe 0x80000000 0x02000000 rom_pda.bin
```

Table 1. Complete dump of the system pagetable.

Virtual Address	Physical Address	Size	KB
v160f9000-160fa000	pa3f77000-a3f78000	1000_{16}	4
v1649f000-164a0000	pa3f60000-a3f61000	1000_{16}	4
v1686f000-16870000	pa3d1f000-a3d20000	1000_{16}	4
v17f66000-17f67000	pa3d54000-a3d55000	1000_{16}	4
v80000000-80400000	**pa0100000-a0500000**	**400000_{16}**	**4096**
v80400000-82000000	**p00400000-02000000**	**$1c00000_{16}$**	**28672**
v88200000-88300000	p48000000-48100000	100000_{16}	1024
v88300000-88400000	p44000000-44100000	100000_{16}	1024
v88400000-89800000	p40000000-41400000	1400000_{16}	20480
v8b400000-8b500000	p28000000-28100000	100000_{16}	1024
v8b500000-8b600000	p20000000-20100000	100000_{16}	1024
v8b600000-8b700000	p38000000-38100000	100000_{16}	1024
v8b700000-8b800000	p30000000-30100000	100000_{16}	1024
v8c000000-8d000000	p0c000000-0d000000	1000000_{16}	16384
v90000000-90100000	pa0000000-a0100000	100000_{16}	1024
v90100000-90500000	p00000000-00400000	400000_{16}	4096
v90500000-94000000	**pa0500000-a4000000**	**$3b00000_{16}$**	**61440**
v98000000-9c000000	p2c000000-30000000	4000000_{16}	65536
v9c000000-a0000000	p3c000000-40000000	4000000_{16}	65536
va0000000-a0400000	pa0100000-a0500000	400000_{16}	4096
va0400000-a2000000	p00400000-02000000	$1c00000_{16}$	29696
va8200000-a8300000	p48000000-48100000	100000_{16}	1024
va8300000-a8400000	p44000000-44100000	100000_{16}	1024
va8400000-a9800000	p40000000-41400000	1400000_{16}	20480
vab400000-ab500000	p28000000-28100000	100000_{16}	1024
vab500000-ab600000	p20000000-20100000	100000_{16}	1024
vab600000-ab700000	p38000000-38100000	100000_{16}	1024
vab700000-ab800000	p30000000-30100000	100000_{16}	1024
vac000000-ad000000	p0c000000-0d000000	1000000_{16}	16384
vb0000000-b0100000	pa0000000-a0100000	100000_{16}	1024
vb0100000-b0500000	p00000000-00400000	400000_{16}	4096
vb0500000-b4000000	pa0500000-a4000000	$3b00000_{16}$	61440
vb8000000-bc000000	p2c000000-30000000	4000000_{16}	65536
vbc000000-c0000000	p3c000000-40000000	4000000_{16}	65536
vfffd0000-fffd1000	pa05a0000-a05a1000	1000_{16}	4
vfffd1000-fffd2000	pa05a0000-a05a1000	1000_{16}	4
vfffd2000-fffd3000	pa05a0000-a05a1000	1000_{16}	4
vfffd3000-fffd4000	pa05a0000-a05a1000	1000_{16}	4
vfffd4000-fffd5000	pa05a0000-a05a1000	1000_{16}	4
vfffd5000-fffd6000	pa05a0000-a05a1000	1000_{16}	4
vfffd6000-fffd7000	pa05a0000-a05a1000	1000_{16}	4
vfffd7000-fffd8000	pa05a0000-a05a1000	1000_{16}	4
vffff0000-ffff1000	pa05a8000-a05a9000	1000_{16}	4
vffff2000-ffff3000	pa05a8000-a05a9000	1000_{16}	4
vffff4000-ffff5000	pa05a8000-a05a9000	1000_{16}	4
vffff6000-ffff7000	pa05a8000-a05a9000	1000_{16}	4
vffffc000-ffffd000	pa05a9000-a05aa000	1000_{16}	4

The pmemmap tool can be used to sample the entire 4 GB of virtual memory as follows (each step of 16 MB takes 16 bytes):

```
pmemmap.exe -s 0x01000000 0 0xfff00000
```

An important task is to locate the starting and ending addresses of the ROM and RAM memory blocks. These addresses can be identified by analyzing the content of the system pagetable (Table 1), which was obtained using the `pmemmap` tool.

The entire binary ROM image is obtained by starting with the virtual address 0x80000000 and specifying a length of 32 MB (0x02000000). This can be verified by summing up the two physical block sizes identified with the virtual and physical addresses as shown below. Note that only one virtual memory block is present, which is mapped to two physical ROM blocks; the two physical blocks together constitute the 32 MB ROM block.

```
v80000000-80400000 -- pa0100000-a0500000  4096 KB
v80400000-82000000 -- p00400000-02000000 28672 KB
```

Extracting the RAM contents is important, especially as the RAM contains all the installed programs along with sensitive user data. Unfortunately, as will be explained below, it is not possible to obtain a complete forensically-sound copy of the RAM. Also, according to Locard's Exchange Principle, the integrity of the RAM memory image cannot be guaranteed because the acquisition process executes in the same memory from where data is being extracted. The pagetable shows a 60 MB block, which contains the object file store (32 MB) along with a substantial portion of the program memory (except for the kernel area). The portion of the pagetable presented below shows six virtual blocks that refer to three physical blocks.

```
v90500000-94000000 -- pa0500000-a4000000   60 MB

v98000000-9c000000 -- p2c000000-30000000   64 MB
v9c000000-a0000000 -- p3c000000-40000000   64 MB

vb0500000-b4000000 -- pa0500000-a4000000   60 MB
vb8000000-bc000000 -- p2c000000-30000000   64 MB
vbc000000-c0000000 -- p3c000000-40000000   64 MB
```

Our experiments indicate that only the 60 MB block is related to the main RAM. Therefore, it is possible to carve the signatures of all the known programs that are present in memory to verify the correctness of the extracted RAM block.

The main drawback of this data recovery technique is the possible lack of integrity of the extracted program memory. This is because the stack and heap portions of the memory are modified as the data extraction process executes. However, the memory portion related to the object store should not change because it is not influenced by the extraction process. Thus, the most important portions of the RAM can be successfully extracted if some integrity loss is acceptable. In any case, the integrity of the extracted data can be verified by analyzing the RAM contents and carving all the signatures related to user objects (programs, sensitive data, etc.).

An important point worth noting is that it is not necessary to scan the entire 4 GB virtual address space, which can take more than two hours. It is much more efficient to analyze the pagetable and focus on the memory blocks that have forensic value; this requires no more than 20 minutes to obtain the entire ROM and RAM contents. We believe that this methodology is applicable to the full range of WinCE devices.

4. Experimental Results

This section presents the results of the case study involving data hiding in WinCE devices. It shows how data can be hidden in the slack portion of the binary ROM of a Toshiba E740 PDA in a manner that prevents its recovery using standard digital forensic tools.

4.1 Binary ROM Image

The Toshiba E740 PDA has a regular binary ROM image of 32 MB. The ROM has a section allocated to the boot loader; the remaining portion of the memory holds the operating system kernel. Inspection of the ROM image released by Toshiba reveals that about 40% of the binary image is empty – this corresponds to about 12 MB of slack space.

Two principal techniques may be used to hide data in the slack portion of the binary ROM image. One approach is to use a flasher tool that has been modified using reverse engineering techniques. The second, simpler approach is use a compact flash card.

Generally, tools for upgrading the operating system are released by the manufacturer. They incorporate a checksum mechanism to verify the integrity of the official binary ROM image and, consequently, to permit its upload. In order to upload a modified version of the binary ROM image, it is necessary to remove this control in the original executable file using reverse engineering techniques. Other checksum tests may be implemented at the boot loader level to verify that a trusted ROM image is loaded into the PDA. It is also necessary to defeat these protection schemes in order to upload arbitrary ROM images.

In the case of the Toshiba E740 PDA, we have developed a technique for re-flashing the device without modifying the executable file or the boot loader. Specifically, it is possible to initiate the re-flashing process by uploading a ROM image on a compact flash card and performing a soft reset with the card inserted in the PDA. This bypasses all the integrity controls, enabling a modified ROM image to be installed in the device.

The binary ROM image is a sequence of contiguous blocks, some of which may be empty; these empty blocks can be used to hide sensitive data. To simplify memory allocation, we used only the empty blocks with size greater than 1 KB to hide data. Since about 40% of the ROM image is empty, approximately 12 MB is available to hide data. Of course, it is necessary to first identify all the empty and usable blocks and locate their starting and ending addresses.

4.2 Hiding Data

Standard strategies used for allocating pages in main memory (e.g., first fit, best fit and worst fit techniques [5, 13]) may be used to hide data within the slack portion of the ROM. We recommend the following data hiding strategy to accommodate the fact that empty blocks in the ROM are of varying size.

- A script (e.g., written in Perl) is used to identify all the empty blocks with size above a certain threshold (e.g., 1 KB). Each block has a starting and an ending address. In addition, a unique number is assigned to each block in order to apply a steganographic scheme. The total slack space, S_{tot}, is represented as:

$$S_{tot} = \{(n_1, s_1, e_1), (n_2, s_2, e_2), ..., (n_K, s_K, e_K)\} \tag{1}$$

  where n_k is the number assigned to the k^{th} empty block for steganographic purposes, and s_k and e_k are its starting and ending addresses, respectively.

- A file, F, is created with size less than or equal to $Dim(S_{tot})$ ($F \mid Dim(F) \leq Dim(S_{tot})$), where $Dim()$ is the space occupied by a specific block of data. Next, an allocation policy is chosen based on a block sequence specified according to Equation 1. Thus, the file F is mapped as follows:

$$F^1 = \{(n_1, s_1, e_1), ..., (n_p, s_p, e_p)\} \tag{2}$$

  where $\binom{K}{p} = \frac{K!}{p!(K-p)!}$ possibilities exist for selecting the p blocks from the K possible blocks. The F^1 file is the result of allocation using an arbitrary sequence of blocks in the set $\{1, ..., K\}$:

$$Dim(F^1) = \sum_{i=1}^{p} b(i) \;\mid\; Dim(F^1) \geq Dim(F). \tag{3}$$

Note that F^1 differs from F in the last block used, which can be greater than the last chunk of the file. The sequence of used blocks forms the steganographic key for recovering the original file.

4.3 Recovering Hidden Data

Every certified binary ROM image has a unique MD5/SHA1 signature that may be used to verify its integrity. An image with a different signature potentially contains hidden data.

The first step in recovering hidden data is to analyze the differences between the two images. Next, data carving techniques and a steganalysis approach are used to recover the hidden data. The procedure for recovering hidden data can be summarized as follows:

- Having verified that the extracted binary image is not original, analyze the differences between the certified ROM image and the extracted image.
- Apply data carving techniques [7] to obtain headers and fragments that might indicate the type of the data and the techniques used to hide it.
- If it is evident that scrambling techniques have been applied, attempt to identify the correct sequence of blocks used for data hiding.

Standard commercial tools such as Device Seizure [10] can be be used for logical data extraction. However, logical data extraction does not recover hidden data allocated within the slack portion of the ROM; only standard objects present at the file system level (e.g., user files, registry and installed programs) are visible and, consequently, recoverable. Unfortunately, Device Seizure was not very effective at physcial data extraction – it was unable to reveal any hidden data.

5. Conclusions

Large amounts of illicit data may be concealed in handheld devices in a manner that prevents their recovery using standard forensic tools. Due to the ubiquity of WinCE devices, digital forensic practitioners must be aware of techniques used for hiding data and for recovering this data. As verified in the case study involving a Toshiba E740 PDA, the methodology proposed for discovering and extracting hidden data from the slack portion of flash ROM and RAM is both sound and efficient. Moreover, the guidelines proposed for identifying and recovering hidden data hold for WinCE devices in general. Our future work will investigate data hiding and recovery techniques for embedded devices using other operating systems (e.g., Symbian OS and iPhone OS X).

References

[1] M. Breeuwsma, Forensic imaging of embedded systems using JTAG (boundary-scan), *Digital Investigation*, vol. 3(1), pp. 32–42, 2006.

[2] M. Breeuwsma, M. De Jongh, C. Klaver, R. van der Knijff and M. Roeloffs, Forensics data recovery from flash memory, *Small Scale Device Forensics Journal*, vol. 1(1), pp. 1–17, 2007.

[3] W. Chisum and B. Turvey, Evidence dynamics: Locard's exchange principle and crime reconstruction, *Journal of Behavioral Profiling*, vol. 1(1), 2000.

[4] P. Gershteyn, M. Davis and S. Shenoi, Forensic analysis of BIOS chips, in *Advances in Digital Forensics II*, M. Olivier and S. Shenoi (Eds.), Springer, New York, pp. 301–314, 2006.

[5] M. Gorman, *Understanding the Linux Virtual Memory Manager*, Prentice-Hall, Upper Saddle River, New Jersey, 2004.

[6] W. Hengeveld, RAPI tools (www.xs4all.nl/∼itsme/projects/xda/tools.html), 2003.

[7] K. Kendall and J. Kornblum, Foremost (version 1.5.3) (foremost.sourceforge.net).

[8] Microsoft Corporation, Remote API 2 (RAPI2), Redmond, Washington (msdn2.microsoft.com/en-us/library/aa920150.aspx).

[9] Microsoft Corporation, Windows CE overview, Redmond, Washington (msdn2.microsoft.com/en-us/library/ms899235.aspx).

[10] Paraben Corporation, Device Seizure v1.2, Orem, Utah (www.paraben-forensics.com/catalog).

[11] A. Savoldi and P. Gubian, Data hiding in SIM/USIM cards: A steganographic approach, *Proceedings of the Second International Workshop on Systematic Approaches to Digital Forensic Engineering*, pp. 86–100, 2007.

[12] A. Savoldi and P. Gubian, SIM and USIM file system: A forensics perspective, *Proceedings of the ACM Symposium on Applied Computing*, pp. 181–187, 2007.

[13] A. Silberschatz, P. Galvin and G. Gagne, *Operating System Concepts*, John Wiley and Sons, Hoboken, New Jersey, 2005.

[14] U.S. Department of Defense, Department of Defense Trusted Computer System Evaluation Criteria, Technical Report DOD 5200.28-STD, Washington, DC, 1985.