When Hardware Meets Software: A Bulletproof Solution to Forensic Memory Acquisition

Alessandro Reina
Dipartimento di Informatica
Università degli Studi di Milano

Aristide Fattori
Dipartimento di Informatica
Università degli Studi di Milano

Fabio Pagani
Dipartimento di Informatica
Università degli Studi di Milano

Lorenzo Cavallaro
Information Security Group
Royal Holloway, University of London

Danilo Bruschi
Dipartimento di Informatica
Università degli Studi di Milano

ABSTRACT

The acquisition of volatile memory of running systems has become a prominent and essential procedure in digital forensic analysis and incident responses. In fact, unencrypted passwords, cryptographic material, text fragments and latest-generation malware may easily be protected as encrypted blobs on persistent storage, while living seamlessly in the volatile memory of a running system. Likewise, systems' run-time information, such as open network connections, open files and running processes, are by definition live entities that can only be observed by examining the volatile memory of a running system. In this context, tampering of volatile data while an acquisition is in progress or during transfer to an external trusted entity is an ongoing issue as it may irremediably invalidate the collected evidence.

To overcome such issues, we present SMMDumper, a novel technique to perform *atomic* acquisitions of volatile memory of running systems. SMMDumper is implemented as an x86 firmware, which leverages the System Management Mode of Intel CPUs to create a *complete* and *reliable* snapshot of the state of the system that, with a minimal hardware support, is resilient to malware attacks. To the best of our knowledge, SMMDumper is the first technique that is able to atomically acquire the whole volatile memory, overcoming the SMM-imposed 4GB barrier while providing integrity guarantees and running on commodity systems.

Experimental results show that the time SMMDumper requires to acquire and transfer 6GB of physical memory of a running system is reasonable to allow for a real-world adoption in digital forensic analyses and incident responses.

Categories and Subject Descriptors

D.4 [**Operating System**]: Security and Protection — System Program and Utilities — Invasive software (e.g., viruses, worms, Trojan horses)

General Terms

Security

Keywords

Forensic, System Management Mode, Live Memory Acquisition

1. INTRODUCTION

Memory acquisition and analysis are well-known and long-term studied digital investigation activities. Although historically focused on information stored and likely hidden on persistent media (e.g., hard disks), digital investigation efforts have nowadays embraced the realm of volatile storage too. For instance, unencrypted passwords, cryptographic material, text fragments and latest-generation malware may easily be protected as encrypted blobs on persistent storage, while live seamlessly in the volatile memory of a running system. In addition, systems' run-time information, such as open network connections, open files and running processes, are by definition live entities that can only be observed on a live, running, system.

Given the importance of the problem and its related challenges, a number of solutions have been proposed to date. Overall, such approaches differ mostly by the degree in which the basic forensic requirements of *atomicity* (i.e., volatile acquisitions must occur within an uninterrupted time-frame), *reliability* (i.e., only trustworthy and consistent acquisitions must be retained), and *availability* (i.e., solutions must be device-agnostic)—informally described in [10]—are satisfied.

Roughly speaking, most solutions are either software- or hardware-based [19, 10]. The former are typically embedded into the operating system kernel, offering isolation (and thus protection) only against typical userspace attacks. Conversely, hardware-based solutions seem to be resilient against kernel-level threats too (for instance, Carrier and Grand rely on a DMA-based physical copy of the volatile memory built on top of a specific PCI device [2]).

Although perceived in principle to be more secure than their software counterparts, hardware-based solutions too may suffer from serious weaknesses, undermining their overall effectiveness. For instance, Rutkowska showed quite re-

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

cently how the privileges of a kernel-level malware can successfully manipulate specific hardware registers to provide device-dependent (i.e., CPU or DMA) split views of the contents of (volatile) memory [16]. As such, it is currently safe to assume that *pure* hardware-based state-of-the-art techniques are as robust (or weak) as their software-based counterparts, leaving memory acquisitions an open challenge.

To address such issues, one of the directions the research community is exploring revolves around the possibility of building dependable memory acquisition techniques on top of system management mode (SMM), a particular execution state of modern x86 CPUs.

Modern x86 CPUs enter SMM via system management interrupts (SMIs). Switching to SMM causes the CPU to save the current system execution state in the system management RAM (SMRAM), a private address space specifically set up for the purpose. Afterward, the execution flow passes to the SMI handler, which is resident in SMRAM. One important feature of the SMRAM is that it can be made inaccessible from other CPU operating modes. Therefore, it can act as a trusted storage, sealed from any device or even the CPU (while not in SMM). Although powerful, SMM has a critical issue: according to the Intel specifications, SMM can only access up to 4GB of physical memory, even on IA-32 CPUs that support physical address extension (PAE) [7], limiting the overall memory acquisition capability of such approaches.

Wang *et al.* propose an hardware-based approach to periodically trigger SMIs (and thus entering SMM) to read the RAM content and send it off to a remote server [21]. Although interesting, that approach has a number of limitations. First, it requires additional hardware (i.e., a PCI card) to be installed on the system *prior* to any live acquisition attempt. Second, no attempt to bypass the 4GB SMM-imposed memory upper bounds is made, greatly limiting the acquisition process, especially on modern hardware equipped nowadays with more than 4GB of physical memory. Finally, Wang *et al.*'s approach does not provide any integrity guarantee on the overall acquisition procedure.

To overcome such issues, we present SMMDumper, a novel technique to perform *atomic* acquisitions of volatile memory of running systems. SMMDumper is implemented as an x86 firmware, which leverages the SMM of Intel CPUs to create a *complete* and *reliable* snapshot of the state of the system that, with a minimal hardware support, is resilient to malware attacks.

Our solution is based on a *collector* and a *triggering* modules. The former is resident in SMRAM and is responsible for transmitting to a trusted host the entire processor state and the memory content, overcoming the SMM-imposed 4GB barrier when PAE is enabled, while providing integrity guarantees and running on commodity systems. Conversely, the triggering module takes care of activating the collector via SMIs. Ideally, a sound and bulletproof implementation of the triggering component should rely on hardware-based activation mechanisms (for instance, a common scenario may see a specific keystroke directly connected to the SMI CPU pin). These artifacts would isolate the protected software component, making it inaccessible from user- or kernel-space. Although such mechanisms have already been proposed in the literature (e.g., [9]), we have opted for a software-only proof-of-concept implementation of the triggering module, for simplicity (a software-emulated hardware triggering mechanism does not affect the effectiveness of SMMDumper).

Roughly speaking, our software-based SMI triggering solution works as follows. We modify the Redirection Table of the I/O APIC to trigger an SMI upon the pressure of an appropriate keystroke combination. Once the SMI is triggered, the CPU switches to SMM, the current system state is saved and our SMI handler (i.e., the collector module) is executed. Although as outlined above our current SMI triggering implementation is vulnerable to kernel-level threats (e.g., SMI invocation avoidance via Redirection Table reconfiguration to reroute keyboard interrupts), SMMDumper's underlining idea remains sound and, moreover, its triggering module implementation can easily be extended to rely on hardware-based mechanisms (e.g., [9]).

In summary, we make the following contributions:

1. We devise a novel firmware-based technique to create a complete and reliable snapshot of the state of the system that, with a minimal hardware support, is resilient to any malware attack.

2. We devise an SMM-based mechanism that enables us to access any physical memory extending over 4GB.

3. We devise a mechanism to digitally sign, while in SMM, the entire RAM contents (extension over 4GB included).

4. We implement a QEMU-based [1] prototype, which enables us to show the usefulness and correctness of our solution as well as perform a performance evaluation.

2. RELATED WORK

Live memory acquisition is an interesting and challenging computer forensics topic, which has largely gained the attention of the research and industry community. In the following, we provide a brief overview of the state-of-the-art, pointing out its main characteristics and limitations.

2.1 Hardware-based Approaches

In principle, every PC hardware bus can be leveraged to gain access to the host physical memory via direct memory access (DMA). For instance, solutions relying on PCI [2], PCMCIA and FireWire [12] buses have been proposed in literature. Such techniques have the advantages of not causing any change in the state of a running OS and being unaffected by most of attacks-hiding techniques. Unfortunately, PCI devices require a prior installation on the system and this greatly reduces their usability. FireWire-based solutions address this issue by allowing analysts to hot-plug them in the target system; thus they can be carried in the toolkit of an incident response team to be installed just after an incident occurred.

Both techniques have limitations. First, they cannot access the processor state (i.e., registers). Second, a knowledgeable attacker, or an advanced malware, can perform a scan of the PCI bus and detect the presence of such ad-hoc devices and consequently decide to stop any malicious activity and wipe every fingerprinting left behind. Lastly, as shown by Rutkowska in [16], DMA devices can be tricked to provide split views of memory contents, thus making the output of such devices unreliable.

2.2 Software-based Approaches

Software-based approaches vary greatly both in complexity and reliability. Such solutions often rely on the OS internals of the host whose memory must be collected. As an example, the simplest way to dump the memory via software is to rely on special virtual devices, if present, like `/dev/mem` on Linux or `\Device\PhysicalMemory` on different Windows systems. Such devices, in fact, allow user space programs to read the whole physical memory of the running system. This possibility leads to trivial memory collection operations, for example through simple UNIX utilities such as `dd` and `netcat`. The main drawback of all these solutions is that they need to be loaded into memory in order to run, thus modifying the original state of the target machine. This causes the captured data to be inconsistent and not reliable. An alternative software solution, even if not always viable, is to crash the system that needs to be analyzed. Indeed, Windows automatically trigger a dump of the physical memory on the hard disk whenever a crash occurs.

Other software memory collection solutions can be used when dealing with virtualized environments. Indeed, virtualization opened up many new possibilities for forensic analyses. The execution of the virtualized system, commonly referred to as the *"guest"* operating system, can be completely frozen for an arbitrary amount of time, allowing for easy atomic collection operations.

One of the most advanced solutions in such a case has been proposed in HyperSleuth [11], where the authors describe a framework that leverages standard hardware support for virtualization to gather memory contents without interrupting the target services. The only drawbacks of this approach are that some small changes in the memory of the target are induced by the installation procedure and that a powerful attacker in the same network of the target could interfere with the packets containing the memory dump.

As previously mentioned, in the last years some authors proposed solutions which exploit the characteristics of System Management Mode. Among them, the closest to our approach is that presented in [21], where a mechanism for RAM collection leveraging SMM is presented. Such a mechanism however has been mainly devised for malware detection and thus does not address some critical issues required by digital investigations. More precisely, these includes the following problems: the integrity of the copy obtained, its adherence to the original content and the possibility of dumping the Physical Address Extension (if present). Furthermore, it requires the installation of a dedicated PCI network card. Our approach, as we will see, also requires a network card to operate, but it leverages the one already installed on the system and not a custom piece of hardware installed *ad-hoc* on the target.

In a subsequent paper [23], an extension of Hypercheck to comply with digital investigations has been proposed. However, no proof of concept has been provided and most of the limitations mentioned above have not been addressed.

3. SYSTEM MANAGEMENT MODE

SMM is a special mode of operation of Intel CPUs, introduced in the i386 processors, designed to handle system-wide functionalities, such as power management and hardware control.

The processors enters SMM in response to a System Management Interrupt (SMI), which has a higher priority compared with other interrupts, and it is signaled through the SMI# pin on the processor or through the APIC bus. When SMM is invoked, the CPU saves the current state of the processor, switches the System Management RAM (SMRAM) address space and begins to execute the code present in it (i.e., the SMI handler).

SMM code is not intended for general purpose applications, but it is limited to system firmware only. Indeed, the main benefit provided by SMM is the execution of the code in an isolated processor environment that operates transparently to the operating system. SMM is defined to be a real-mode environment with 32-bits data access when operand- and address-size override prefixes[1] are used. Otherwise, operand's and address's size is restricted to 16-bits.

The only way to exit from the SMM operation mode is by means of the `rsm` instruction that is available only in the SMM. The `rsm` instruction takes care of restoring the saved state of the processor, and returns the control to the interrupted program. When the processor is in SMM all hardware interrupts, but software-invoked interrupts and exceptions, are disabled.

During boot time, it is a duty of the BIOS firmware to intialize SMRAM, copy the SMI handlers to it, and, as a form of protection, lock SMRAM to disallow any further writing accesses to this area. In fact, SMM is the mode of operation with the greatest level of privilege, informally named ring -2, and has to be as safe as possible from malicious users. The solution presented in this paper works properly only if the SMRAM has been locked at boot time.

3.1 Threat Model

Before describing the technique we devised, we must delineate the threat model in which we expect SMMDumper to be used. Our model assumes the availability of an hardware method to trigger an SMI and trasfer control to SMMDumper, such as the one briefly depicted in Section 1. At the end of the section, however, we discuss what are the consequences of being unable to adopt an hardware solution.

The machine, which memory we must acquire, has allegedly been compromised and these conditions may hold:

- The attacker (or malware) has root access to the compromised system.
- The attacker has compromised other machines in the same network of the target.
- The attacker can perform network attacks (e.g., intercepting and modifying packets).

On the other hand, SMMDumper is not able to deal with the following situations:

- The attacker has access to the smartcard containing the private key used to sign collected memory.
- The attacker exploits vulnerabilities that allow *write* access to SMM memory of the target system [25].
- The attacker has physical access to the target system and is able to power it down or mount a DMA attack that wipes the memory before it is collected [24].

[1] Intel and AT&T syntaxes, respectively provide `A32`, `O32` and `addr32`, `data32` as explict address- (`0x67`) and operand-size (`0x66`) override prefixes.

We are well-aware that an attacker could write a malware able to thwart the execution of the memory dump, whether we use our fallback software-based solution to trigger an SMI. Specifically, the malware must have administration privileges in order to launch the attack, tamper our SMI triggering solution and subsequently prevent the launch of the dump of the system memory. To achieve its goal, the malware modifies I/O APIC Redirection Table by setting the delivery mode of the `IRQ1` to `Fixed` and the `Interrupt Vector` field to the malicious interrupt vector in the `IDT`. By doing so, the malware disables the ability of an user to trigger an SMI by pressing a specific keystroke that launches the memory dump. With respect to others techniques to trigger an SMI [22], the technique that we adopt, presented in Section 4.1, is more challenging to be silently disabled by a malware. This is due to the fact that it requires the modification of an offset of the well-known address of the I/O APIC Redirection Table. This behavior can be considered "suspicious" at least and may ring an alarm bell for common anti-virus software. This, of course, is far from being an optimal solution: as we already stated, the best solution would be a dedicated hardware method. Furthermore, to the best of our knowledge, every technique proposed so far that leverages SMM and needs to trigger an SMI on an allegedly compromised system, suffers from the same problem [22].

4. SMM-BASED MEMORY DUMP

This section describes in detail the design and implementation of SMMDumper, the SMM-based infrastructure we have devised to perform a consistent and unforgeable dump of the volatile memory of a running operating system.

As briefly outlined in Section 1, SMMDumper can be logically divided in two components: a triggering module and a memory collector module. The former component is responsible for invoking SMIs and thus entering SMM. Ideally, this component should be implemented in hardware to provide strong guarantees and resiliency against malware threats. Instead, we have opted for a software-based implementation to allow our solutions to be used on commodity hardware. The memory collector module represents the main component of SMMDumper. It is in charge of reading the physical memory of the target host and transmit it over the network. It is a BIOS extension loaded in SMRAM at boot time and unauthorized modifications of its content are prevented by having the BIOS locking write access to that specific region.

A global overview of SMMDumper architecture can be observed in Figure 1. Intuitively, a forensic analyst invokes SMMDumper by initiating a predefined keystroke sequence (1). This sequence is immediately intercepted by the triggering module, which switches the system CPU to SMM. The memory collector module (2) starts subsequently, initiating the host physical memory dump over the network (3). As we will see shortly, before the acquisition process actually starts, SMMDumper waits for detecting the presence of a commodity cryptographic device, which must be plugged into the system *after* entering SMM. This device is responsible for creating on-chip digital signatures and to provide strong integrity guarantees of the transmitted data (4).

Running code at system management mode privilege opens a number of challenges that need to be properly addressed to achieve the goals mentioned at the beginning of this section. In particular, we must (i) trigger system management interrupts to switch to SMM, (ii) be able to access *all* the physical memory of the target system, even when more than 4GB of physical memory is installed on 32-bit hosts, (iii) guarantee the integrity of the collected data on the host as well as while in transit to a generic—but trusted—device. Meeting such challenges clearly guarantees the atomicity (i and ii), reliability (iii), and availability (iii) forensic requirements illustrated in Section 1.

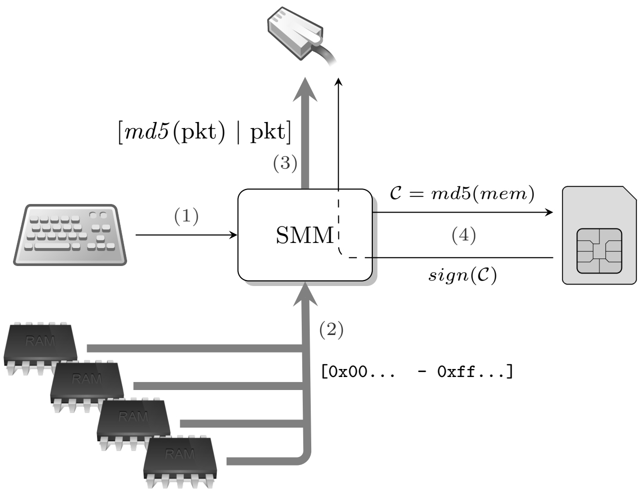

Figure 1: Overview of the system

4.1 System Management Interrupts

Intel CPUs enter SMM by invoking a System Management Interrupt (SMI). SMIs can be triggered through either an external SMM interrupt pin (`SMI#`) or the Advanced Programmable Interrupt Controller (APIC). Even though only one SMI pin is physically hard-wired to the CPU, different events—generally specified by the I/O Controller Hub (ICH)—can trigger SMIs. Modern chipsets, such as the Intel ICH10 [5], have approximately 40 different ways to trigger an SMI, such as power management, USB, Total Cost Of Ownership (TCO), writing to the Advanced Power Management Control port register, periodic timer expiration and SMBus events. In addition, some motherboards are equipped with dedicated hardware that can be legitimately exploited to raise SMIs. For example, an SMM interrupt switch installed on the motherboard to allow users to suspend the system when turned on (power-save mode). Nonetheless, triggering an SMI requires a proper software register configuration.

Switching to SMM to start a whole-system memory dump requires raising an SMI whenever a specific keystroke sequence is detected. To this end, our approach builds on [3] to implement a fully-functional SMM-based keylogger. In particular, everything revolves around the Intel Advanced Programmable Interrupt Controller (APIC), which overlooks the communication between the CPU and external devices. The APIC is divided into I/O and Local APIC. They are located on the chipset and integrated onto the CPU, respectively, and communicate over a dedicated APIC bus. The I/O APIC receives external interrupt events from the system hardware and its associated I/O devices and, depending on the configuration of its Redirection Table, routes them to the Local APIC as interrupt messages. The Local APIC delivers the interrupts received from the I/O APIC to the CPU

```
  data32
  movl $0xdeadbeef , %eax
start:
  data32
  addr32
  movl (%eax), %ebx
  ;; Do something with %ebx
  data32
  addl $0x4, %eax
  data32
  cmpl $0xdeadceef , %eax
  jl   start
```

Figure 2: Accessing physical memory from SMM

it belongs to (after consulting the Local Vector Table, which specifies how interrupts are delivered to the CPU and their priorities). Finally, the Interrupt Descriptor Table (IDT) is indexed with the vector number by the CPU to select the proper Interrupt Service Routine (ISR) handler to invoke.

Overall, the Redirection Table plays a crucial role in the above-sketched process, as it specifies the interrupt vector and delivery mode of each interrupt pin. In particular, the delivery mode is fundamental to accomplish the goal of triggering an SMI when an arbitrary, custom and predefined, keystroke sequence is observed. To this end, we set the delivery mode of the `IRQ1`[2] to SMI and the vector information to 0s to properly forward that IRQ line to our SMI handler.

Our SMM ISR handler extracts the keyboard scancode from the keyboard controller buffer, reading from the I/O port `0x60`. Any scancode mismatching the predefined keystroke sequence is properly re-injected to the keyboard controller buffer by writing the keyboard controller command `0xd2` to the I/O port `0x64` [18].

Modifying the I/O APIC Redirection Table to deliver an SMI when the `IRQ1` IRQ line is asserted requires to forward the interrupt to the CPU. This is achieved by sending Interprocessor Interrupts (IPIs) from software by properly configuring the Local APIC Interrupt Command Register (`ICR`). To this end, we set the `ICR Destination Field` to `self` and the `Delivery Mode` to `fixed`. Writing to the least-significant doubleword of the `ICR` causes an IPI message to be sent out and the interrupt to effectively be delivered to the CPU as soon as the `rsm` instruction is executed.

4.2 Accessing Physical Memory

As noted elsewhere, SMM is similar to real mode. Therefore, the size of operands and addresses of the instructions executed in SMM are limited to 16 bits, restricting the addressable memory to 1MB. However, override prefixes are generally used to access up to 4GB of the addressable memory space [8], as briefly sketched in Figure 2. In particular, the snippet of code iteratively reads 4KB of memory starting from the address `0xdeadbeef`. It is worth noting that running code in SMM disables paging, allowing for a direct access to *physical*—rather than virtual—memory. While this has the clear benefit of allowing a straightforward memory access without worrying about virtual-to-physical address translations (and viceversa), it has drawbacks too. According to the Intel specifications [8], SMM can only access up to 4GB of physical memory, even on CPUs that support Physical Address Extension (PAE) or long mode with IA-32e. To overcome such a limitation, one may think about enabling paging in SMM to populate a custom Page Table to map physical-to-virtual pages and access them through virtual addresses. Unfortunately, enabling paging requires to switch the CPU to protected mode (the de-facto default mode of operation of Intel-like CPUs nowadays). Within SMM, this can only be achieved by executing the `rsm` assembly instruction, which causes an exit from SMM too [7]. We defer the solution SMMDumper adopts to Section 4.4.

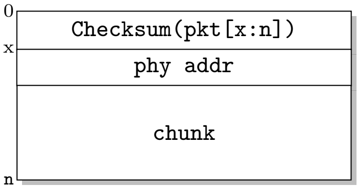

Figure 3: Packet format

4.3 Data Integrity and Transmission

Direct physical memory access alone does not meet all the forensics requirements sketched in Section 1. Ideally, SMMDumper could just read memory one byte at a time and send it off over the network. However, this would hardly represent a reliable solution. Running SMM code is also similar to running in hypervisor mode: there is no operating system service we can rely on and it is like being executed directly on the bare metal. Therefore, transmission errors may just occur and there are no in-place mechanisms to address such issues (e.g., data loss or integrity corruptions).

To overcome such limitations and meet the reliability forensic requirement outlined in Section 1, we have designed a simple, yet effective, communication protocol. Intuitively, SMMDumper divides the physical memory into chunks of fixed size (1KB in our current implementation). Each chunk is then embedded in a packet structured as shown in Figure 3. The base address of the memory chunk contained in the packet (i.e., `phy addr`) is metadata and represents a unique label that is used by the receiver to correctly handle out-of-order or missing chunks. Checksum over the whole packet payload (metadata and the physical memory chunk), instead, offers the opportunity to detect integrity violations.

4.3.1 Data Signing

Clearly, the simple checksum-based scheme outlined above protects only against transient network transmission errors (similar to what a TCP segment checksum does), but it easily fails against an attacker that purposely modifies data on-the-fly and recomputes the checksum to reflect such changes. To address this threat, SMMDumper computes an incremental checksum $\mathcal{C}$ of the whole physical memory, as individual packets are sent off[3]. Subsequently, when the transfer of the whole physical memory is completed, SMMDumper signs $\mathcal{C}$

[2] We intercept PS/2 and USB keyboard when its state is set as legacy mode.

[3] While details of the actual network transmission and retransmission due to loss or incorrect data are described next, let us just assume here that all the packets have been correctly transmitted to the receiver and that the whole system physical memory has thus been dumped.

and sends the resulting ciphertext blob to the receiver, which verifies the signature and compares $\mathcal{C}$ against a freshly computed checksum over all the received packets. A valid signature guarantees the integrity of the received signed checksum and matching checksums guarantee the integrity of all the received packets. To be able to incrementally compute $\mathcal{C}$, an appropriate algorithm needs to be chosen. As we will see in Section 5, SMMDumper uses the MD5 cryptographic hash function.

The question of where to store the private key used for signing $\mathcal{C}$ still remains. Any attempt to embed the key into SMMDumper may be threatened by sophisticated attacks aimed at reading its code or data. To overcome such threats, we have decided to offload all the cryptographic operations (key management and signing) to an external hot-pluggable smart card device $\mathcal{D}$. In particular, as soon as SMMDumper starts executing (that is, as soon as the system enters into SMM), it waits for $\mathcal{D}$ to be plugged into the system to carry out the tasks outlined above. Not only this procedure guarantees the integrity of the collected memory, but it also provides validity as the signing key can only be accessed by whoever has access to $\mathcal{D}$ and can start the live forensic memory dump.

It is worth noting that hot-pluggable memory collection devices may induce tangible side-effects in the target memory as a result of their installation or initialization in the system, as pointed out in Section 2.1. However, SMMDumper requires the analyst to plug the smart card device once the system has entered SMM, where no operating system service is in execution and any potential side-effect is under the control of SMMDumper. In addition, different smart cards allow for the use of different signing identities, which can be easily produced and distributed to forensic analysts. By using a personal smart card, the forensic analyst implicitly certifies that a given live memory acquisition is associated to that specific, personal, smart card.

Assuming that errors do not generally occur, the schema just outlined is both effective and efficient as it generally requires only one encryption operation (signing) per a whole memory dump. In fact, signatures of individual packets would negatively impact on the overall overhead. On the other hand, if an individual packet is corrupted, the whole dump must be taken again. Although the errors-are-seldom-events assumption seems reasonable, SMMDumper can be easily extended to suite the analysts needs. For instance, signed checksums could be sent out for every 500MB of data, trading-off packet retransmissions and overall overhead.

4.3.2 Network Transmission

Once packets are ready, SMMDumper needs to transfer them from the target machine to an external trusted host or device, for future analyses. We have opted for the former solution. Conversely, the latter would, for instance, require to store packets on an USB storage plugged on the target machine and this raises two main concerns. First, interacting with USB devices from SMM is everything but simple and the code needed to interact with the USB controller would likely be bloated and prone to errors. Second, the destination USB device needs to be physically plugged on a port on the target system while, leveraging a network connection, the collected data can be sent to a remote host, tipically on the same local network but, *potentially*, even elsewhere on the Internet.

SMMDumper implements a basic network driver that is able to communicate through I/O operations with the Network Interface Card of the target machine. As noted elsewhere, operating in SMM does not allow to rely on any operating system-provided service, such as networking. Therefore, SMMDumper is also equipped with the code responsible to forge UDP packets sketched in Figure 3. Once again, the choice of the UDP protocol rather than the more reliable TCP is driven by the willingness of keeping the code as simple as possible and ease the burden of a fully-functional (and complex) implementation. As described next, this is not a limitation *per-se* as we do have enough metadata to recover from arbitrary transmission errors.

One could argue that transferring data over the network is less secure than using a removable device as the attack surface increases. Indeed, an attacker that compromises a machine in the same network of the target could use different techniques (e.g., ARP spoofing [20]) to intercept or prevent the reception of the packets sent by SMMDumper. However, it is impossible for an attacker to arbitrarily modify the content of a packet without being detected as the overall checksum sent at the end of the transfer is signed with a private key that, in our threat model, is inaccessible to the attacker. An attacker could still perform a Denial of Service attack, by dropping or damaging packets, but we argue that it is easy to identify this kind of attacks and its source inside a local area network and, consequentially, to exclude it from the network and then request a retransmission of the blocked packets through the protocol explained in the next paragraph.

4.3.3 Retransmission of Lost Data

As soon as SMMDumper finishes sending the memory to the remote host, it switches from send mode to listen mode, where it accepts requests to retransmit certain chunks of memory. The remote hosts uses metadata contained in the packets to check if everything was transmitted correctly, identifying in the process missing or corrupted chunks. Every lost packet is then asked back to the SMMDumper that recreates the packet and tries again to transfer it. An attacker, of course, could try to mangle with this mechanism, for example by modifying sent requests or forging fake ones. These are not really problematic attacks as the remote host knows which packets it has requested and can simply discard fake ones.

4.4 Accessing more than 4GB of Memory

There may be some situations in which SMMDumper is required to access more than 4GB of physical memory: if the CPU on the target machine supports Physical Address Extension (PAE), Page Size Extension (PSE-36) or IA-32e mode (namely 64bit support). PSE-36 is very similar to PAE, it just changes some internal structure of the page tables. Thus, in this paper we address only PAE among these two alternatives, as we believe that modifications needed by SMMDumper to handle PSE-36 would be trivial. Unfortunately, handling IA-32e mode is not straightforward and it is part of our ongoing research effort.

4.4.1 Handling PAE on IA-32

Physical Address Extension is a paging mechanism that is supported by an extension of physical addresses from 32 bits to MAXPHYADDR bits, where MAXPHYADDR is 36 bits on IA-

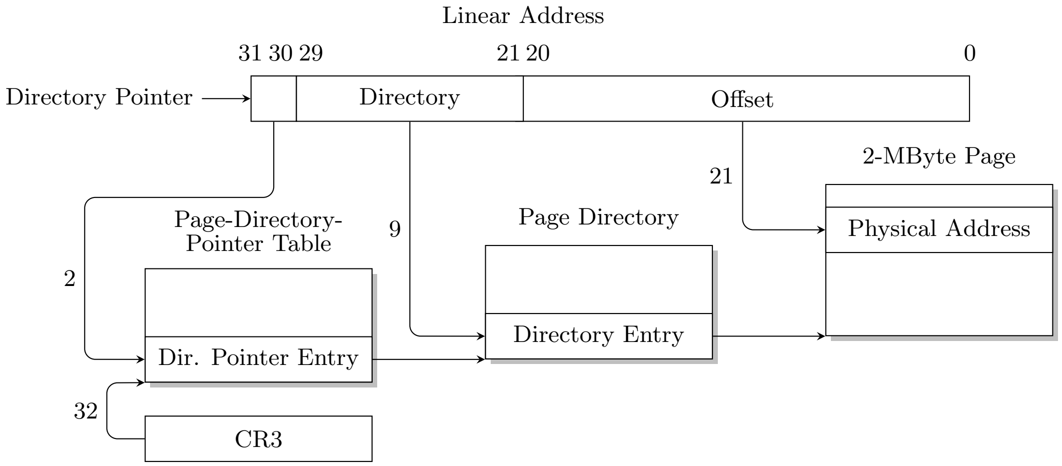

Figure 4: Address translation with PAE (2MB pages)

32 CPUs. Since our approach deals with IA-32 CPUs, from now on we will consider `MAXPHYADDR` to be limited to 36 bits. Theorically, this allows to use up to 64GB of memory. Unfortunately, as we said before, when we are in System Management Mode, we are limited to 32 bit addressing and, thus, we cannot use 36 bit addresses and registers. Furthermore, we cannot enable paging and switch to virtual addressing as this is also forbidden when in SMM. Some authors [25, 4] state that, surprisingly, it is instead possible to switch from SMM to protected mode, or even IA-32e. However, we did not find any specification that explicitly allows this switch and our experiments confirmed that it is not possible indeed. To the best of our knowledge, some CPUs may allow it but as we aim to reach a good degree of compatibility we do not rely anyhow on this possibility.

The rationale behind our solution is simple: once we finish dumping the first 4GB of physical memory, with the method explained in the previous Sections, we change the paging structure used by the target system so that we can control the mapping between virtual and physical memory, both from SMM and protected mode. Then, we inject some code in the mapped pages and we modify `EIP` field in the State Save Map so that it points to the code that we inject. Once everything is set up, we issue a `rsm` instruction and go back to protected mode. During the switch, `EIP` is restored from the State Save Map and thus the execution of the system resumes from the custom code we injected *before* the switch.

Setting-up Paging.

To better understand how we setup page tables before returning in protected mode, we reported the address translation mechanism with PAE enabled in Figure 4 as explained in [7]. For the sake of simplicity, we only analyze (and use) paging with 2MB pages. When PAE is enabled on a IA-32 CPU, the size of virtual addresses remains 32 bit while the size of physical addresses is extended to 36 bit. When translating a Virtual Address (VA) into a physical one, the MMU uses the structures depicted in Figure 4. The first two bits of the virtual address points to an entry of the Page Directory Pointer Table (whose physical address is stored in the register `CR3`). The MMU then checks this entry (PDPTE) to see if it is actually mapped (present bit) and if the required access is allowed; then, it follows the physical address contained in the entry, that points to the second structure: the Page Directory. Bits 21:29 of VA determine the Page Directory Entry (PDE) the MMU must use. The MMU performs the same check on the PDE and then it uses the base address contained in the PDE and bits 0:20 of VA to calculate the physical address corresponding to VA.

As can be observed, the whole address translation mechanism depends on the `CR3` register. When in SMM, however, we cannot modify the `CR3` register value stored in the State Saved Map. However, even if PAE is enabled, this register is *not* extended to 36 bits so we can access the memory pointed by the register, thus modifying the paging structure at our likings, even if we are restricted to 32-bit addressing. As can be seen in Figure 4, the `CR3` register points to the first Page Directory Pointer Table Entry (PDPTE). Part of the bits are used to control the Page Directory mapped by this PDPTE (e.g., if it is present, if caching is enabled) while the remaining part contains its physical address. Every PDPTE is 64 bit long, thus, to set it up correctly, we have to access it in two "rounds", using `CR3` for the lower 32 bits and `CR3+4` for 32 higher bits. SMMDumper edits the first PDPTE so that it points to a Page Directory located at physical address `0x0`. This implies that now the first Page Directory Entry (PDE) is located at `0x0`. SMMDumper configures this entry to be present, writable, accessible from every control privilege level and large (2MB instead of 4KB). The page base address of the first PDE is then set to `0x0`. This may seem wrong at a first glance, since we already use `0x0` as base address of the Page Directory. On the contrary, this is not only correct but also very convenient. Indeed, it means that we will be able to use *virtual* address `0x0` to access a 2MB physical page whose first 4KB correspond exactly to the Page Directory itself. This is possible because we edited the first PDPTE and the first PDE of the system, thus the MMU, when translating virtual address `0x0`, will walk these two entries, as we explained before, and will translate the virtual address into the physical address `0x0`. Furthermore, being able to access the Page Directory once switched to protected mode, by using virtual range `[0x00000000-0x00001000]`, will allow us to directly edit paging structures (i.e., map new physical pages into virtual ones) without having to reference physical addresses.

Returning to Protected Mode.

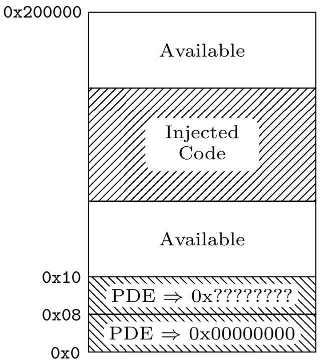

Figure 5: Layout of physical/virtual page 0x0

Before returning the execution control to the system, further setup is required. First of all, we must insert the code that will be executed in protected mode. Obviously, it must be carefully tailored to avoid overlapping the memory areas we reserved for PDEs. This code is stored in the SMRAM along with the code needed to perform the dump of the first 4GB and then copied into a section of the first mapped page. This page has been mapped as explained in the previous paragraph, and its layout can be observed in Figure 5. As we can see, the first 16 bytes are reserved for PDEs: the first PDE maps physical page 0 (i.e., itself) on virtual page 0, thus allowing access to paging structure and to injected code even when SMMDumper will go back to protected mode. The PDE at address 0x00000008, on the other hand, is prepared but still unused: SMMDumper will modify it to map physical memory pages that still need to be sent out (i.e., above 4GB). The injected code, depicted in Figure 6, performs the following tasks:

```
1  va       = 0x00200000
2  p_pde    = 0x00000008
3  phy_addr = 0x100000000 /* 36-bit */
4  while phy_addr < MAX_MEMORY :
5    /* Setup PDE */
6    p_pde->page_base_addr = phy_addr
7    p_pde->p  = 1 /* Present */
8    p_pre->us = 1 /* User/Super */
9    /* Now 0x00200000 points to phy_addr */
10   offset = 0
11   while offset < PAGE_SIZE:
12     packet =  str(phy_addr+offset)
13     packet += va[offset:offset+CHUNK_SIZE]
14     packet += MD5(packet[0:len(packet)])
15     /* Send pkt */
16     /* Update overall checksum */
17     offset += CHUNK_SIZE
18   phy_addr += PAGE_SIZE
```

Figure 6: Dump of the upper physical memory

- modify PDE at virtual address 0x00000008, so that it will map a physical 2MB page that needs to be dumped. Since this PDE is the second one contained in the Page Directory under our control, this physical page will be mapped at virtual address 0x00200000 (lines 6 - 8);
- read a chunk of memory from *virtual* page starting at 0x00200000, create a packet with checksum and send it over the network (lines 12 - 15). Alternatively, if we want to avoid code duplication, it is possible to just store memory chunks at an address that is readable from SMM (e.g., any area tagged as *Available* in Figure 5) and trigger a SMI. The code executed in SMM will then take care of signing and sending operations just like it did for the first 4GB;
- loop through the inner loop (lines 11-17) until a full 2MB page has been dumped, then repeat the procedure (lines 4-18) until the whole memory higher than 4GB has been dumped.

Before finally returning back to protected mode we have to make sure that (I) the execution flow will not go out of our code and that (II) our code has enough privileges to operate without triggering faults. To avoid (I) we clear the Interrupt Flag in the EFLAGS register stored in the Save System Map so that interrupts will not cause a transfer of the execution flow to the system interrupt handlers. Non-Maskable Interrupts (NMI), can still happen even if EFLAGS.IF is 0, so we disable them by interacting *directly* with the APIC. Disabling NMI is not a problem as the code that we will execute in protected mode does not rely on them anyhow. Problem (II) is mainly due to the fact that, from SMM, we cannot modify the CPL that will be set upon a resume. Fortunately, the only operations for which our code will need some custom privileges are I/O operations (both for interacting with the NIC or to trigger a SMI and get control back to code in SMM). This privileges can be granted by altering the Input Output Privilege Level (IOPL) bits in EFLAGS just like we do for IF.

4.5 Portability

The approach adopted by SMMDumper is *completely* OS-independent and can be easily applied as a patch to already existing BIOS or installed in new ones. The most restricting requisites of SMMDumper are mainly hardware:

1. The presence on the target system of a port (USB, serial) to interact with the smartcard hardware.
2. A network interface card to send out packets.

Developing drivers for many different network interface cards may be painful, as every piece of hardware has its specifications and peculiarities. However, most BIOS already include primitives to interact with on-board NICs (e.g., PXE [6] functionality required to interact with the network). BIOS manufacturers willing to support SMMDumper can easily modify it to include additional drivers of well-known on-board NICs.

5. EXPERIMENTAL EVALUATION

To verify the soundness of the proposed approach, we implemented a prototype based on coreboot [13], so that it can be easily installed on many production systems as a BIOS update. The collector module of SMMDumper has been entirely coded in Assembly and it counts slightly less than 500 LoC, of which ≈ 47% is for the MD5 implementation. To perform tests, we employ QEMU 1.0.1 [1], equipped with a single Intel 3GHz processor, 6GB of RAM and a Realtek RTL8139 100Mbps network card.

To interact with the NIC, we implemented a small driver that is able to run in SMM to send and receive UDP packets. The code required to interact with this piece of hardware is quite simple: to transmit a packet, the driver writes the physical address and the size of the packet to the appropriate control registers of the device. Then, it polls the status register of the device until the transmission is completed. We use polling instead of interrupts because interrupts are disabled as soon as we switch to SMM. Packets reception, that is needed for the retransmission protocol, is implemented in the same way. It is worth noting that we allocate the packet transmission buffer in a region of memory outside the SMRAM because devices cannot access SMRAM due to hardware restrictions [17]. Before using this region, we backup it into the SMRAM and restore it later, in order to send it.

As a checksum algorithm to grant both packet and overall memory integrity we use `MD5`. As we briefly outlined in Section 4.3.1, our choice of `MD5` is convenient, as it allows us to *incrementally* calculate the overall checksum of the memory as we read it to create packets. To univocally sign the overall checksum, SMMDumper leverages an external hardware device that is able to read a private key from a smart card and use it to encrypt data. To perform experiments, we created a simulated smart card reader that can be hotplugged into our development environment by means of a Serial Port (RS232). According to our threat model, since the communication channel is established once SMMDumper is already executing, no attacks can interfere with this channel.

To evaluate the performance of SMMDumper and the average time needed to transfer the acquired volatile data over the local network, we performed a number of transmission tests (all the results are averages with negligible standard deviation). With a chunk size of 1024B of memory, each transmitted UDP packet carries a total payload size of 1024(`chunk`) + 16(`MD5`) + 8(`phy addr`) bytes. Using that value as our reference payload size, we measured the time needed to transfer 6GB of memory to be $\approx 820''$, $\approx$ 10% of which is overhead due to the calculation of the `MD5` checksum over a single packet. The overall overhead caused by the metadata inserted in every packet is 144MB. It is worth noting that choosing a chunk size bigger than 1024B would introduce error-prone implementation intricacies in exchange for a relatively small gain. By maximizing the chunk size up to the MTU, indeed, we would incur in cross-page readings that, especially when dealing with PAE, may be tricky to deal with. Assuming a chunk size of $(MTU - TCP\ header - 16 - 8) = 1436$B, the time needed to dump 6GB and the overhead of metadata would respectively drop to $\approx 814''$ and $\approx$ 103MB. This roughly corresponds to an interesting downgrade of 29% of the overhead caused by metadata. However, when calculated on the whole traffic, the overall improvement is *only* of 0.66%. For this reason, our current implementation relies on a 1024B chunk size and trade a negligible performance boost for a more linear algorithm.

Finally, we must verify if SMMDumper is able to guarantee the properties that we illustrated in Section 1. Firstly, to evaluate if *atomicity* is satisfied, we instrumented our development environment in order to take a snapshot of the system memory before starting to execute our SMM ISR handler. Then, we compared such a dump with the one produced by SMMDumper. Results showed that our technique allows to gather an accurate and consistent memory dump: as we expected, no changes occur on the host after we trigger an SMI by pressing a specific sequence of keystrokes. However, we are aware that some changes may occur meanwhile SMMDumper reads memory mapped I/O regions which are reserved for devices. These changes however do not affect the *consistency* of our dump. Indeed, they cannot invalidate the overall checksum since it is incrementally calculated, i.e., we just read these areas once, both for sending them out and to calculate the hash. However, changes may happen between the time the user press the keystroke and the time at which the actual chunk is sent out over the network. Moreover, since we mapped SMRAM `[0x000a0000-0x000affff]` in a section of memory that overlaps the area used by video RAM `[0x000a0000-0x000bffff]`, the memory readings in that address range result in a dump of the SMI ISR handler and the SMRAM State Save Map, instead of the original content of the memory area. We assert that these changes do not violate *consistency* and *atomicity* properties as the information they contain are strictly related to physical devices and are not relevant for the analysis.

Secondly, we verified if the requirement of *reliability* is satisfied. In this part of the evaluation, we performed a man-in-the-middle attack between the target machine running SMMDumper and the receiver host [14]. The attack allowed us to accurately tamper with the packets, modifying the payload and re-calculating the `MD5` checksum to hide evidences of our modifications. The receiving host, at the end of the procedure, detected our evil modifications leveraging the overall encrypted checksum that SMMDumper sent at the end of the dump.

In summary, our experiments show that SMMDumper is able to provide an *atomic* and *reliable* snapshot of the target system, in a timely fashion.

5.1 Future Work

The actual SMMDumper implementation does not address Multi-Processor (MP) systems. Nevertheless, SMMDumper can be easily extended for handling it. In MP systems, SMI is propagated to all processors. As described in [15], processors enter SMM at slightly different times, because SMI could be serviced in between of CPU instructions. Therefore, SMM ISR handler waits for all processors to enter SMM and then, using a semaphore, only one processor executes the memory dump while the others wait it to complete.

Our threat model does not implicitly include the presence of malicious hardware-assisted hypervisors on the target system. However, when we do not have to dump more than 4GB of RAM, our solution is resilient to this kind of attacks. On the other hand, as we explained in Section 4.4, the fallback solution to dump higher memory must exit SMM with an `rsm` instruction. An hardware-assisted hypervisor could intercept the `rsm` [8] and get hold of the execution control before SMMDumper, thus disabling the acquisition of memory higher than 4GB.

Furthermore, we are currently working on a solution to tackle the problem of dumping even more than 64GB of RAM (i.e., on CPUs that support IA-32e mode). Preliminary results show that it is indeed possible to dump more than this amount of RAM. Furthermore, to the best of our knowledge, we believe that our future solution will be resilient to malicious hardware-based hypervisor attacks.

6. CONCLUSION

In this paper, we presented an SMM-based volatile memory acquisition technique that overcomes many of the limitations affecting state-of-the-art solutions. In particular, we have shown how SMMDumper is able to atomically perform a live memory acquisition, while guaranteeing the on-system and in-transit integrity of the acquired information.

While the firmware-based implementation of our proof-of-concept may be undermined by sophisticated kernel-level malware, the design of SMMDumper remains sound, arguing that the introduction of a naive and inexpensive hardware modification by vendors, such as an interrupt line directly connected to the processor SMI pin, would make SMMDumper completely bulletproof and resilient to any form of attacks.

Our experimental evaluation shows that SMMDumper is effective and efficient, allowing for its real-world deployment in digital forensic analyses and incident responses.

7. REFERENCES

[1] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[2] B. D. Carrier and J. Grand. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1(1):50–60, 2004.

[3] S. Embleton, S. Sparks, and C. Zou. SMM rootkits: a new breed of OS independent malware. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks*, page 11. ACM, 2008.

[4] Intel. Intel Software Network. `http://software.intel.com/en-us/forums/showthread.php?t=63946`.

[5] Intel. *Intel I/O Controller Hub 10 (ICH10) Family*, 2008.

[6] Intel Corporation. Preboot Execution Environment (PXE) Specification, 1999.

[7] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3A*, May 2012.

[8] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B*, May 2012.

[9] F. Z. Kun Sun, Jiang Wang and A. Stavrou. SecureSwitch: BIOS-Assisted Isolation and Switch between Trusted and Untrusted Commodity OSes. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, California, USA, February 2012.

[10] E. Libster and J. Kornblum. A Proposal for an Integrated Memory Acquisition Mechanism. *ACM SIGOPS Operating Systems Review*, 42(3):14–20, 2008.

[11] L. Martignoni, A. Fattori, R. Paleari, and L. Cavallaro. Live and Trustworthy Forensic Analysis of Commodity Production Systems. In *Proceedings of the 13th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2010.

[12] A. Martin. FireWire memory dump of a Windows XP computer: a forensic approach. `http://www.friendsglobal.com/papers/FireWire%20Memory%20Dump%20of%20Windows%20XP.pdf`.

[13] R. Minnich. coreboot. `http://www.coreboot.org/`.

[14] A. Ornaghi and M. Valleri. Man in the middle attacks demos. *Blackhat [Online Document]*, 2003.

[15] Phoenix. BIOS Undercover: Writing A Software SMI Handler, 2008. `http://blogs.phoenix.com/phoenix_technologies_bios/2008/12/bios-undercover-writing-a-software-smi-handler.html`.

[16] J. Rutkowska. Beyond the CPU: Defeating hardware based RAM acquisition tools, 2007. `http://invisiblethings.org/papers/cheating-hardware-memory-acquisition-updated.ppt`.

[17] T. Schluessler and P. Rajagopal. OS Independent Run-Time System Integrity Services. *Research Paper, IT Innovation and Research, Intel Corporation*, 2005.

[18] S. Sokolov. 8042 keyboard controller. `http://stanislavs.org/helppc/8042.html`.

[19] T. Vidas. *Acquisition and Forensic Analysis of Volatile Data Stores*. PhD thesis, University of Nebraska at Omaha, 2006.

[20] R. Wagner. Address resolution protocol spoofing and man-in-the-middle attacks. *The SANS Institute*, 2001.

[21] J. Wang, A. Stavrou, and A. Ghosh. Hypercheck: A hardware-assisted integrity monitor. In *Proceedings of the 13th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2010.

[22] J. Wang, K. Sun, and A. Stavrou. An Analysis of System Management Mode (SMM)-based Integrity Checking Systems and Evasion Attacks. Technical report, George Mason University, 2011.

[23] J. Wang, F. Zhang, K. Sun, and A. Stavrou. Firmware-assisted Memory Acquisition and Analysis Tools for Digital Forensics. In *Proceedings of the 6th International Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE)*, Oakland, California, USA, May 2011.

[24] R. Wojtczuk. Subverting the Xen hypervisor. *Black Hat USA*, 2008, 2008.

[25] R. Wojtczuk and J. Rutkowska. Attacking SMM memory via Intel CPU cache poisoning. *Invisible Things Lab*, 2009.