Now You See Me: Hide and Seek in Physical Address Space

Ning Zhang
Virginia Polytechnic Institute and State University
Blacksburg, VA, USA
ningzhang@vt.edu

Kun Sun
College of William and Mary
Williamsburg, VA, USA
ksun@wm.edu

Wenjing Lou
Virginia Polytechnic Institute and State University
Blacksburg, VA, USA
wjlou@vt.edu

Y. Thomas Hou
Virginia Polytechnic Institute and State University
Blacksburg, VA, USA
thou@vt.edu

Sushil Jajodia
George Mason University
Fairfax, VA, USA
jajodia@gmu.edu

ABSTRACT

With the growing complexity of computing systems, memory based forensic techniques are becoming instrumental in digital investigations. Digital forensic examiners can unravel what happened on a system by acquiring and inspecting in-memory data. Meanwhile, attackers have developed numerous anti-forensic mechanisms to defeat existing memory forensic techniques by manipulation of system software such as OS kernel. To counter anti-forensic techniques, some recent researches suggest that memory acquisition process can be trusted if the acquisition module has not been tampered with and all the operations are performed without relying on any untrusted software including the operating system.

However, in this paper, we show that it is possible for malware to bypass the current state-of-art trusted memory acquisition module by manipulating the physical address space layout, which is shared between physical memory and I/O devices on x86 platforms. This fundamental design on x86 platform enables an attacker to build an OS agnostic anti-forensic system. Base on this finding, we propose Hidden in I/O Space (HIveS) which manipulates CPU registers to alter such physical address layout. The system uses a novel *I/O Shadowing* technique to lock a memory region named *HIveS memory* into I/O address space, so all operation requests to the HIveS memory will be redirected to the I/O bus instead of the memory controller. To access the HIveS memory, the attacker unlocks the memory by mapping it back into the memory address space. Two novel techniques, *Blackbox Write* and *TLB Camouflage*, are developed to further protect the unlocked HIveS memory against memory forensics while allowing attackers to access it. A HIveS prototype is built and tested against a set of memory acquisition tools for both Windows and Linux running on x86 platform. Lastly, we propose potential countermeasures to detect and mitigate HIveS.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: [Software Architectures - Information hiding]; K.6.5 [**Management of Computing and Information Systems**]: [Security and Protection - Invasive software]

General Terms

Design, System

Keywords

Digital Forensics; Memory Acquisition; System Security; Rootkits

1. INTRODUCTION

Digital forensics is the science on collecting and presenting digital evidence. With the ever increasing use of computing systems in our daily life, computers and networks have become not only the personal portal to instant information, but also a platform that criminals exploit to commit crimes. Digital forensics is now one of the services sought at the very beginning of all types of investigation - criminal, civil, and corporate [7, 13].

Disk forensic methods and tools have matured in the past two decades, offering comprehensive capabilities to extract and visualize artifacts from nonvolatile storage images. With the prevalence of memory hiding techniques and the need to evade disk forensic, adversaries are starting to hide the presence of malicious code and data only in the memory [8, 16, 15]. To tackle this problem, forensic examiners are increasingly relying on live memory forensics to uncover the malicious contents in the memory [7].

There are two general memory acquisition approaches: *software based* approach and *hardware based* approach. Software based solutions rely on a trusted memory acquisition module in the operating system to acquire the memory through the processor [35, 21]. Hardware based solutions often utilize dedicated I/O devices, such as network interface card, to capture physical memory image via direct memory access (DMA) [10, 28, 39] with the processor totally bypassed. Some hardware based approaches use the remanence of physical memory to extract sensitive data from memory module in systems that are powered off for a short time [11, 18].

To counter live memory forensics, attackers have developed a number of anti-forensic techniques to sabotage the memory acquisition process [19]. Current anti-forensic techniques against software based memory acquisition rely on manipulating the software used in the memory acquisition process. Some examples include modifying the acquisition module or the OS kernel [20, 34, 15], hooking operating system APIs [32], or installing a thin hypervisor on the fly [29]. Based on this observation, Stüttgen et al. recently suggested that the memory acquisition process can be trusted with two conditions. The first one is that the acquisition module has not been tampered with, and the second one states all the operations are performed without relying on the operating system or any other untrusted software [35]. However, in this paper, we show that this assumption is not true by presenting Hidden in I/O Space (HIveS), an operating system (OS) agnostic anti-forensic mechanism, that is capable of evading the most updated software based memory forensics tools.

Physical address space on x86 platform is shared between physical memory and I/O devices. Memory access to a physical address gets directed to either the memory controller or the I/O bus based on where it is located in the address space layout. This physical address layout is also what memory forensics tools use to understand where the physical memory regions are located. Memory forensic tools obtain this layout information by interacting with operating system or BIOS, and they assume this layout is correct and updated. We show that this condition can be easily violated by presenting HIveS. HIveS alters the machine physical address layout while the system is in operational state by modifying registers in the processor. With this mispresented address layout, HIveS can conceal a memory region called *HIveS memory* from being observed and acquired by memory forensics tools.

The basic idea is to map (or lock) the HIveS memory into the I/O space, so that any operation on the physical memory address will be redirected to the I/O bus instead of the memory controller. When the HIveS memory is locked, its memory contents cannot be accessed by any processor, including the one(s) controlled by the attacker. When the attacker wants to access the HIveS memory, she would first unlock the memory region by mapping it back into the memory address space. To protect the unlocked HIveS memory against memory forensics, we propose two novel techniques, *Blackbox Write* and *TLB Camouflage*. Blackbox Write enables only write access to the HIveS memory by creating asymmetric read and write destinations between the memory space and the I/O space. TLB Camouflage exploits TLB cache incoherency among multi-core processors to ensure exclusive read and write access for a single processor core to the HIveS memory.

HIveS is operating system agnostic, since it only changes the system hardware configurations. We build a prototype of HIveS on an x86 desktop with an AMD FX processor running both Windows and Linux. Since HIveS conceals the presence of malware without changing any system software including BIOS, hypervisor or OS kernel, it can effectively defeat the most updated software based memory acquisition tools on both Windows and Linux. Furthermore, we extend HIveS with a number of existing anti-forensic techniques, such as *RAM-less encryption* and *Cache based I/O storage*, to defeat hardware based memory acquisition approaches.

We propose several countermeasures for detecting and mitigating HIveS. One seemingly simple solution is to directly inspect the CPU registers that may have been manipulated by HIveS. However, since legitimate peripheral device drivers may also change the same set of CPU registers, it remains a challenge to distinguish normal configurations from malicious usages, and maybe impossible without crashing the system on some hardware platforms.

To summarize, we make the following contributions.

- We present HIveS, a system that exploits hardware features in x86 platform to subvert the foundation of memory acquisition. HIveS is an OS agnostic anti-forensic mechanism that can defeat memory forensic techniques by concealing the HIveS memory in the I/O space.
- We develop two novel techniques to enable covert operations on the unlocked HIveS memory against memory forensics. Blackbox Write grants only the write privilege to the HIveS memory, and TLB Camouflage can grant malicious users exclusive read and write access to the HIveS memory.
- A prototype of HIveS is built on the x86 platform to demonstrate its capability on concealing the HIveS memory against a number of most updated memory forensics tools on both Windows and Linux.
- We propose potential countermeasures to detect and mitigate HIveS. As an arms race, we show that HIveS can be enhanced to further evade hardware based memory acquisition solutions.

The remainder of the paper is organized as follows. Section 2 describes some background knowledge on x86 memory address space. We present the HIveS framework in Section 3 and discuss its extensions in Section 4. A prototype implementation is detailed in Section 5. We propose potential countermeasures in Section 6. Section 8 discusses the related works. Finally, we conclude the paper in Section 9.

2. BACKGROUND

The entire range of memory addresses accessible by x86 processors is often referred to as *physical address space*. Contrary to popular believes, the length of such address space usually does not equal to the amount of actual physical memory installed on the platform. This is because some of the address is mapped to the bus for I/O devices, instead of dynamic random access memory (DRAM). A typical memory layout of systems with AMD processors is shown in Figure 1, where the shaded areas are backed by DRAM, and the areas without shade are backed by I/O devices. This memory layout is used by the Memory Map Unit (MMU) to route memory requests from the processor to either DRAM or memory-mapped I/O (MMIO).

The memory setting of an x86 system is initialized by the BIOS at hardware reset and parsed by the operating system during the system bootstrap [9]. The layout is configured via several configuration registers in the North Bridge (NB) and the processor. DRAM Base/Limit register pair is among the earliest ones configured by the BIOS. They define the ranges of physical address space

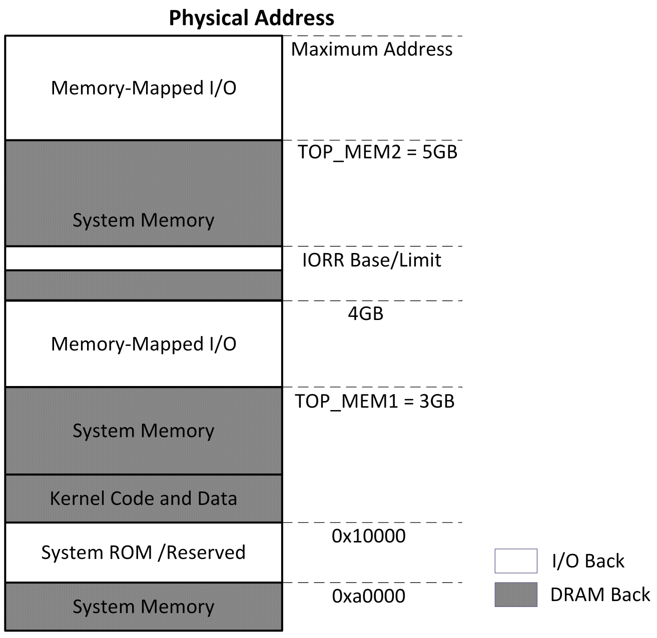

Figure 1: Physical Address Layout on AMD Architecture

mapped to DRAM in the north bridge. Any access to these areas will be forwarded to the DRAM Controller (DCT). These registers are configured by the BIOS with the result of system memory probing during hardware initialization. Therefore they are designed to be lock-once (i.e., write-once). The values cannot be changed until the next system reset.

The next set of registers that shapes the memory layout consists of two Mode Specific Registers (MSR) Top Of Memory (TOM) registers. AMD processors allow system software to use TOM registers to specify where memory accesses are directed for a given address range [2]. There are two TOM registers, *TOP_MEM1 (TOM1)* and *TOP_MEM2 (TOM2)*. Figure 1 shows that the address range from 0 to TOM1 as well as the address range from 4GB to TOM2 are set as system memory on this AMD system. Access requests within these two ranges are directed to the DRAM, while requests outside these two ranges are directed to the I/O space. The purpose of these two registers is to offer the operating system software the ability to carve out large memory space to organize DRAM and I/O devices. Even though they can be changed even when the system is operational, unlike the DRAM Base/Limit register, it is rare to change the memory address allocation after the system starts up. This is because the DRAM boundaries, governed by DRAM Base/Limit registers, have already been determined. Lastly, systems usually stop functioning if these registers are changed, since the OS kernel was not expecting the change of hardware configuration while the system is running.

The last set of registers that shapes the layout is also MSR in the processor. They are the Input Output Remap Registers (IORR). These set of registers can create a special mapping beyond the base setting to direct specific read/write access of any address space between the I/O space and the DRAM space. This set of registers are designed to enable system software to shadow ROM device in memory to improve the system performance.

3. HIVES FRAMEWORK

In the ongoing battle between attackers and digital forensics examiners, memory acquisition is becoming an important technique for evidence collection. From the perspective of an attacker, we design HIveS, an anti-forensic system. It is capable of evading acquisition by software based memory forensics tools on a designated range of physical memory chosen by the attacker. We call this range of memory *HIveS memory*. It can be used by attackers to store malicious code or sensitive data.

A high level block diagram of the HIveS system is shown in Figure 2. For simplicity, we show a generic x86 multi-core architecture with one processor consisting of two cores. Each processor core has its own cache and TLB.

When a processor core needs to access the DRAM memory, it sends a request to the north bridge. The MUX inside the north bridge is responsible of forwarding the memory request to either the DRAM controller or the south bridge based on the physical address layout. This layout was initialized by the BIOS, then further defined by the operating system using model-specific registers (MSRs) including the top of memory (TOM) registers and the I/O range registers (IORRs). When the physical address is mapped to the I/O space, the request is forwarded to the south bridge. When the physical address falls in the DRAM range, the memory request goes through the DRAM controller to the physical memory.

HIveS has two states, *locked* and *unlocked*. When it is in the locked state, the HIveS memory is completely inaccessible to any processor core. This is because all access attempts are forwarded to the I/O space once HIveS is locked. While the system remains in this state, even the malicious core (e.g., Core 1 in Figure 2) cannot access the HIveS memory. When the attacker needs to access the HIveS memory, she can set HIveS to unlocked state, where only the malicious core can access the HIveS memory, and memory requests from all other cores are redirected to another DRAM region. Lastly, since HIveS relies only on hardware configurations to conceal the HIveS memory, it is OS agnostic. Moreover, it leaves no trace in memory. Unlike some of the current rootkits that modifies kernel data structures or operating system APIs, HIveS cannot be detected by checking the integrity of the OS.

3.1 Inaccessibility in the Locked State

Considering the use case of a password stealing rootkit, whose goal is to steal passwords and store them quietly in some place before an opportunity to exfiltrate, there is no need for the rootkit to read from or write to the memory where the stolen passwords are stored until it is ready to transmit. HIveS is designed to an anti-forensic tool, so we develop a novel *I/O Shadowing* technique to block all processor cores from accessing the HIveS memory. The basic idea of I/O shadowing is to dynamically manipulate the configuration of a memory range so that even if it is backed by the DRAM in the physical address space, any read/write request will be redirected to the I/O space. The real contents in the DRAM memory are shadowed by the memory-mapped I/O (MMIO).

Among the various controls that shapes the memory layout, there are two MSRs that can be controlled by the system software when the system is operational. They are TOM and IORR.

Though TOM registers can be modified after the system boots up, any modification of the TOM registers can greatly affect the

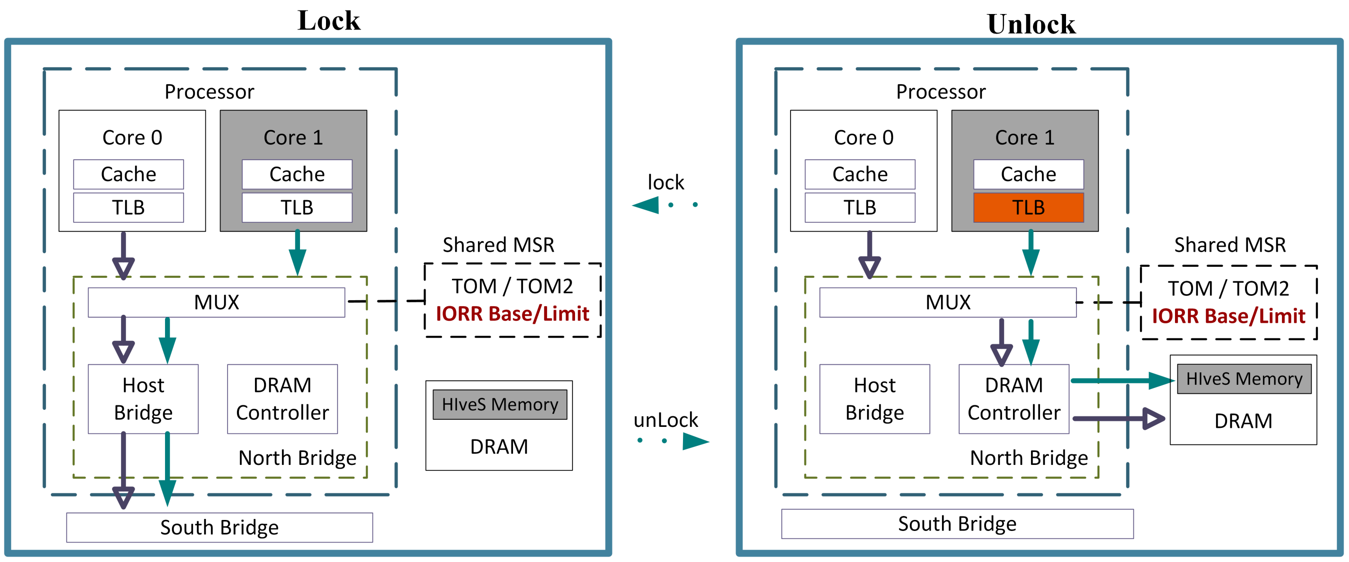

Figure 2: Architecture of HIveS

system stability, since the OS kernel uses the TOM registers in many default system settings. Furthermore, TOM modifications can only change the boundary between the default I/O area and the DRAM area. Even if system instability was not an issue, the manipulation would be very limited.

We instead use I/O range registers (IORRs) to adaptively prevent all processor cores from accessing the HIveS memory. IORRs are variable-range memory type range registers (MTRRs). They can be used to specify if reads and writes in any physical address range should map to system memory or memory-mapped I/O (MMIO). In AMD architecture [2], up to two address ranges of varying sizes can be controlled using IORRs. Figure 1 shows an example that maps an area of system RAM between 4GB and 5GB into MMIO using one IORR.

Each IORR has a pair of registers, *IORR base register* and *IORR mask register*. The IORR mask register contains the length of the region and a valid bit indicating whether the IORR configuration pair is active. IORR base register contains the starting address of the IORR region, as well as two important flag bits, *WrMem* and *RdMem* [2]. When these two bits are set to 1, the north bridge directs read/write requests for this physical address range to system memory. When these bits are cleared to 0, all reads/write requests are directed to memory-mapped I/O.

The RdMem and WrMem bits in IORR are originally designed for shadowing ROMs of I/O devices in DRAM memory to improve system performance. The system can create a shadow region by setting $WrMem = 1$ and $RdMem = 0$ for a dedicated memory range and then copy the ROM from I/O device into DRAM memory. Once the copy operation is completed, the system changes the bit value to $WrMem = 0$ and $RdMem = 1$. Now the memory reads are directed to the faster copy in the DRAM memory instead of ROM of the device; write requests are still being directed to the ROM, but the ROM simply ignores any write request.

The I/O shadowing provided by IORRs can be misused to redirect processor requests of a valid system memory area to the I/O space. When both RdMem and WrMem bits are set to 0 in the IORR, all read and write requests to the HIveS memory will be redirected to the I/O space. With this configuration, the HIveS memory becomes inaccessible for all processor cores. Since both Windows and Linux operating systems make no assumptions on the default configurations and usages of IORRs, the modification of unused IORR registers has no impact on the OS reliability. In addition, IORR registers offers great adaptability in both the location and size of the HIveS memory.

3.2 Exclusive Access in the Unlocked State

The HIveS memory in the unlocked state is designed to allow an exclusive access from the processor core controlled by the attacker, while preventing acquisition by the processor cores that perform memory forensics. IORRs are registers shared by all processor cores, so any modification on one IORR register affects all the processor cores in the system. When an attackers needs to access the HIveS memory in a single core system, she can simply unlock HIveS memory by disabling the I/O shadowing, read or modify contents in the HIveS memory, and then lock it by enabling the I/O shadowing. However, it becomes a challenge to ensure an exclusive access to HIveS memory with parallel execution in a multi-core system, since the forensic examiner can be collecting memory with the other running core. We develop two new techniques, *Blackbox Write* and *TLB Camouflage*, to solve this problem.

3.2.1 Blackbox Write

When an attacker with an active keylogger uses HIveS memory to store the collected sensitive data, it will be writing to the HIveS memory most of the time and does not need to frequently read it back. On the other hand, forensic examiners are interested only on reading the memory contents. In order to preserve the integrity of the evidence, memory forensic tools always read the memory contents and never write to the memory.

Based on the above asymmetric operations between the attackers and the examiners, we develop *Blackbox Write* to redirect all

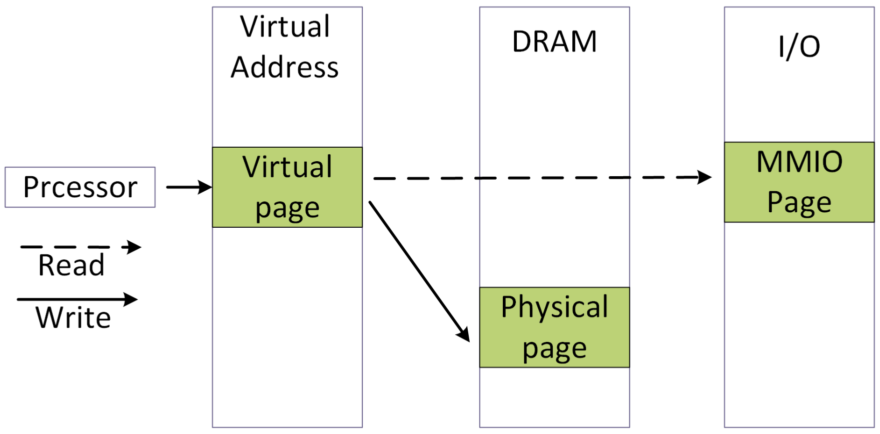

Figure 3: Blackbox Write

Algorithm 1: TLB Camouflage

```
begin
    allocate a new memory page;
    pause all other running cores;
    all cores flush TLBs;
    modify the new page PTE to point to the HIveS memory;
    malicious core read/write the virtual address;
    malicious core TLB entry loaded;
    modify the new page PTE back to regular address;
    resume all other cores;
```

memory read requests to the I/O space by setting $RdMem = 0$ in IORR and send all the memory write requests to the HIveS memory by setting $WrMem = 1$. With this setting, attackers can write new contents into the HIveS memory while preventing forensic examiners from reading and analyzing it. Because there is no real I/O device in the I/O hub to respond to the memory reads, a default value (e.g, $0xFF$ in AMD FX processor [4]) is returned instead. Note the examiner can also write into the HIveS memory, however actively modifying memory is a an act of compromising evidence, which is against the principle of digital forensics.

The attacker eventually needs to send the data in the HIveS memory to a remote machine. For instance, after a fix amount of user key strokes have been stealthily recorded, the keylogger can send the data to a remote server using network packets. Instead of unlocking processor's read access to the HIveS memory, the attacker can manage to read the HIveS memory by a peripheral device via DMA. To prevent random peripheral devices from reading the HIveS memory, HIveS can set the IOMMU to only allow a designated peripheral device to access the HIveS memory. Thus, a network interface adapter can read the key stoke logs from the HIveS memory via DMA and exfiltrate them.

3.2.2 *TLB Camouflage*

Blackbox write is an effective technique for malware that continuously stores sensitive data in a secret place with little need to read back, such as keyloggers. However, when the malware needs to unlock the HIveS memory for continuous read and write, it leaves a large time window for memory forensic tools to acquire the HIveS memory. We propose TLB Camouflage technique to mitigate this problem. Figure 4 shows the basic idea of TLB camouflage, where the unlocked HIveS memory can only be accessed by the malicious Core 1 that is controlled by the attacker, while the read and write requests from Core 2 for memory forensics are redirected to another memory space. TLB camouflage enables exclusive access to HIveS memory by creating an incoherent view of memory mapping between cores, allowing the HIveS memory contents be accessed only by the processor core that is running the malicious software.

Modern operating systems enable paging mechanism to translate virtual memory address into physical memory address before passing the memory access request to DRAM Controller (DCT) [9]. Translation-Lookaside Buffer (TLB), also known as page-translation caches, is designed to reduce the performance penalty during the time-consuming address translation process [2]. Only one memory access per virtual memory request is required when the translation for the demanding page is present in the TLB (a TLB hit). When there is no entry in the TLB for the demanding page, a TLB miss occurs. And the translation information for the page is copied from a page table entry (PTE) into the TLB (a TLB reload).

Each processor core has its own TLB [2, 3]. When the operating system changes a page mapping, the TLB won't be automatically updated to reflect the new virtual to physical address translation. TLB camouflage exploits this property to create a page translation incoherence among different processor cores.

A pseudo code of TLB Camouflage is shown in Algorithm 1. The idea is to create an incoherent cache entry in the TLB caches among the running processor cores. A new page is allocated in the kernel for the page translation manipulation, such that the rest of the system would not be affected. Then, all the other processor cores are paused. At this point, the malicious core can flush the TLB and to make sure that there is no preexisting translation stored for the our newly allocated page already. The PTE of the allocated page is then modified to point to HIveS memory, and several LDR instructions are then used to force a translation table walk and TLB reload. And the malicious core would have a TLB entry mapping to HIveS memory. Then the PTE is modified back to the original values, and the other cores are resumed. Technically, the TLB entry for the allocated page of malicious core is incoherent, and contains

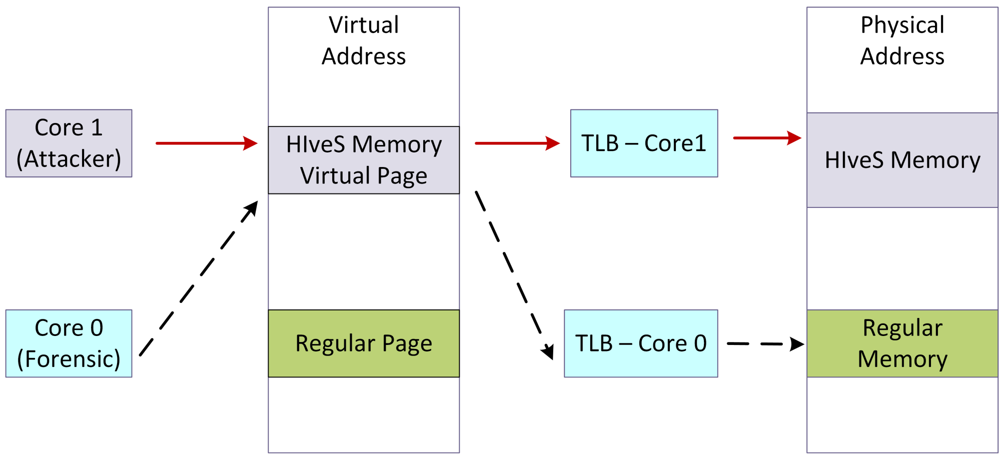

Figure 4: TLB Camouflage

a false mapping. And this is exactly what we need. In Figure 4, when Core 2 requests to access the virtual page of the HIveS memory, it will get the content in the regular memory. On the other hand, since the malicious Core 1 has an incoherrent TLB entry pointed to the HIveS memory address, it can access the HIveS memory if the TLB entry has not been flushed out.

TLB camouflage technique greatly increases the usability of HIveS memory, which can be used not only as a temporary storage with few interactions, but also as an interactive memory storage to support more malicious operations.

However, TLB Camouflage has some limitations. First, not all forensic tools rely on the existing kernel page tables to map virtual addresses to physical addresses [35]. In this case, TLB camouflage won't work since it depends on manipulating the kernel page tables. Second, the TLB entry should be sustained all the time; otherwise, the malicious core cannot access the HIveS memory either. Since TLB locking capability is not supported by the latest x86 architecture, malicious code has to freshen the TLB entry periodically in order to keep it active in TLB depending on the processor implementation.

3.3 HIveS Memory Access Property

When the HIveS memory is in the locked state by applying the I/O shadowing technique, none of processor cores can read or write the HIveS memory. Most of the time, the attacker does not need to access the HIveS memory at all, so it can lock the memory for a better protection. However, the attacker has to unlock the memory eventually to access it. When the attacker only needs to write to the HIveS memory, it can use the Blackbox Write technique. Moreover, if the attacker also needs to frequently read the memory contents, it can use the TLB Camouflage technique. Table 1 shows the different access privileges to the HIveS memory for both attackers and forensic examiners when applying different anti-forensic techniques.

HIveS is operating system agnostic, so the HIveS memory can be concealed on x86 platforms for both Windows and Linux. However, we need to develop a kernel module on Linux or a device driver on Windows with the root privilege to set the hardware registers. Contrary to current rootkits that modify kernel data structures or routines in the operating system, HIveS does not leave any trace in the memory or hard disk, so it cannot be detected by checking the integrity of the OS image in the memory and the hard disk.

4. HIVES EXTENSION

HIveS is mainly developed to defeat software based memory acquisition methods that rely on a trusted software module in the operating system to acquire the physical memory through the processor to memory interface. Both I/O Shadowing and Blackbox Write rely on modifying the IORR registers, and TLB Camouflage creates an incoherent page translation in TLB caches of multiple processor cores. All the modifications are made on the processor, and thus only affect processing of memory request originated from the processor. On the other hand, hardware based memory acquisition solutions can detect HIveS, since a dedicated I/O device can capture physical memory image via direct memory access, which totally bypasses the processor hardware configurations made by HIveS. Moreover, the Cold Boot technique [18] exploits the physical remanence property of memory chips to directly extract sensitive data from the chips. Cold boot technique resets the system and invalidates all configurations prior to system reset. To enhance the capability of HIveS against the hardware based forensics tools, we propose to retrofit a number of existing techniques in HIveS, including IOMMU, RAM-less encryption, and Cache based I/O storage.

4.1 Hiding from I/O Devices

We propose to use IOMMU to evade physical memory forensics by I/O devices via DMA. Similar to the translation from virtual memory address to physical memory address performed by MMU, IOMMU is a hardware device that translates device DMA addresses into proper physical memory addresses. Each I/O device is assigned a protected domain with a set of I/O page tables that define the corresponding memory addresses. During a DMA transfer, the IOMMU intercepts the access message from the I/O bus and checks its cache (IOTLB) for the I/O to memory address translation along with the access right. IOMMU is controlled with in-memory tables and memory-mapped registers. Once a DMA request passes IOMMU, it is then processed by the north bridge.

Table 1: Comparison of Access to HIveS Memory

	Attacker Read	Attacker Write	Regular Read	Regular Write
I/O Shadowing	no	no	no	no
Blackbox Write	no	yes	no	yes
TLB Camouflage	yes	yes	no	no

The north bridge then forwards the request either to the I/O hub or the DRAM controller base on the ranges defined by DRAM Base/Limit and MMIO Base/Limit registers. Therefore, HIveS can set the IOMMU to only allow a peripheral device to perform DMA into assigned regions, thus preventing a full system memory acquisition with DMA. When the IOMMU is not available on some old systems, the DMA can also be redirected by manipulating the north bridge using MMIO Base/Limit registers. The main idea is to modify the MMIO Base/Limit registers to bounce DMA reads back to the I/O hub. The details can be found in [30].

4.2 Hiding from Cold Boot

There are two solutions to evade Cold Boot based memory acquisition mechanisms: *RAM-less encryption* and *Cache based I/O storage*.

RAM-less encryption. The basic idea is that attacker encrypts all the memory contents in the HIveS memory with a secret key stored in CPU registers [24, 33]. Since operating systems do not use all the MTRR and IORR register pairs all the time, HIveS can encrypt the HIveS memory using AES and store the encryption key in unused MTRR or IORR registers. Thus, even if the physical memory is completely acquired through Cold Boot, the contents of HIveS are still being protected, because the encryption key in the CPU registers is lost forever due to the system reset.

Cache based I/O storage. The idea is to save a small HIveS memory only in the CPU cache [22, 26, 17] and then mask it with I/O Shadowing technique. When the memory address is set to cacheable in the page table entry and both RdMem and WrMem bits in the IORR base register are set to 1, any write to that location will trigger a cache line fill if the memory contents are not yet loaded in the cache. When the HIveS system is unlocked, the attacker can simply write data into memory as usual. When the HIveS system is locked, the HIveS memory is cached and masked by I/O shadowing. Therefore, neither I/O devices nor the processor can read out the HIveS memory in the cache via DMA. However, it remains a challenge to maintain the contents in the cache considering the limited cache control provided by the x86 architecture [2, 3].

5. IMPLEMENTATION AND EVALUATION

We build a prototype of HIveS on an x86 desktop with AMD FX processor. The motherboard is ASUS M4 A96 R2.0, running a AMD FX-8320 8-core processor with single bank DDR3 4GB memory. The 4GB memory is relatively small but it shortens the time for memory acquisition and it is large enough to demonstrate all the functionality of HIveS.

To illustrate the effectiveness of the HIveS memory, we implement a keylogger rootkit called *HIL* that uses HIveS memory to store the keystrokes so that the stolen information cannot be detected by memory forensics. We implement HIL prototypes on both Windows and Linux. On Ubuntu 13.04, we implement a Linux kernel module to support all the techniques in HIveS. On 64-bit Windows 7, we implement a kernel mode device driver as a keylogger and use WinDbg debugger to configure the IORR pair.

We implement I/O shadowing, Blackbox Write, and TLB Camouflage techniques and evaluate their effectiveness using a number of most updated software based memory forensic tools. We also implement RAM-less encryption and cache based I/O Storage techniques to demonstrate the capability of HIveS to evade Cold Boot based physical memory forensics.

5.1 I/O Shadowing

Since modification of MSR require privilege mode, we implemented most of the functionalities in a kernel module. User space programs can communicate with the kernel module through *procfs* export. For I/O shadowing, the kernel module is responsible for manipulating the IORR register to set the base and the size of the HIveS memory, as well as the WrMem and RdMem flag bits. With the physical address and HIveS running mode passed in through procfs, the module first masks off the lower 12 bit of the physical address, and inserts it into bits 12 to 47 in I/O Range Base register, *MSRC001_0016*, since the physical addressing in AMD x64 is 47 bit. The bits 3 and 4 of the register are RdMem bit and WrMem bit respectively. For I/O Shadowing, we clear both bit 3 and bit 4 to redirect both read and write requests into the I/O space.

The IORR base register should always be written first, since the IORR mask register, *MSRC001_0017*, contains a *valid* bit, which will immediately enable the IORR pair once this bit is set. Therefore, we cannot set the two IORR registers in reverse order; otherwise, the system will fail and hang itself. In the AMD FX system [2], the valid bit is bit 11 of the IORR mask register.

Although the detailed HIveS implementation is different on Linux and Windows, the workflow remains the same. We first load HIveS as a kernel module in the system. An 1MB area at physical address offset of *0x10c800000* is allocated to be the HIveS memory. With RdMem and WrMem both set, we fill the memory with repeating pattern of *0x12345678*. Once the pattern is written, we flush the cache to make sure that these patterns are written into the memory. Then we enable I/O shadowing to lock the HIveS memory by clearing both the WrMem and RdMem bits. At this point, all the contents in the HIveS memory should be protected against memory forensic tools.

We verify that none of the software based memory forensic tools that we tested is able to capture the HIveS memory protected by the I/O shadowing technique. Table 2 summarizes the tools that we use in our experiments. Tools such as the Linux base memory acquisition tool LiME [36] obtain the physical memory layout of the system by parsing kernel data structure. For each valid memory re-

Table 2: Verification against memory forensic tools

Tool	Tool Version	OS	Examine IORR	With HIveS	Without HIveS
UnitTest	1.0	Linux	No	No Detection	Identified
LiME	1.1	Linux	No	No Detection	Identified
MemDump	1.01	Linux	No	No Detection	Identified
DD	8.13	Linux	No	No Detection	Identified
WinPmem	2.3.1	Windows	No	No Detection	Identified
Mem Marshall	1.0	Windows	No	No Detection	Identified
Memoryze	3.0	Windows	No	No Detection	Identified
Dumpit	1.3.2	Windows	No	No Detection	Identified

gion, it calls $kmap$ function to map virtual pages to physical pages to read the content.

Table 2 shows that none of these memory forensic tools can detect the HIveS memory through searching the special repeating pattern *0x12345678* when the I/O shadowing is enabled. However, when the memory dumps are taken again after the I/O shadowing is disabled, we can identify the repeating pattern in the memory dumps. None of the tools can detect HIveS, since all the processor memory accesses can be manipulated by the IORR registers and all the software based tools utilize processor to access memory.

5.2 Blackbox Write and TLB Camouflage

Blackbox Write only provides write access to the HIveS memory and prevents any read access. We implement it by clearing the RdMem bit and setting the WrMem bit. To disable Blackbox Write, we simply clear the valid bit of the IORR pair. To verify its effectiveness, we set up the keylogger to work in the Blackbox Write mode. Instead of filling the HIveS memory with repeating pattern *0x12345678*. We run the keylogger, and manually type in "this is a HIveS blackbox write test!". When Blackbox Write is enabled, we dump the memory using the memory acquisition tools, including LiME, MemDump, and WinPmem, to capture the entire physical memory images. And we verify that the sentence we typed was not found in the acquired memory image. Immediately after the first round of memory dump, we disable Blackbox Write to allow both read and write access to the HIveS memory and perform memory dumping again. This time, we were able to find the logs of what we just typed.

TLB camouflage protects the HIveS memory by only allowing read and write access to a single processor core. After pausing all other cores, we flush the TLBs of all cores. Next, we disable all interrupts on the malicious core and then read the contents of the HIveS memory into a temporary memory space. The kernel module then goes in a busy loop accessing the memory location continuously to sustain the TLB entry in the malicious core's TLB. We confirm that only a single processor core can access the HIveS memory by dumping the memory images using different processor cores and searching the coded repeating pattern.

5.3 RAM-less Encryption

For RAM-less encryption, we use a secret key to XOR the plaintext instead of using the AES function, since the feasibility of RAM-less encryption has already been verified [24, 33] and our focus is on testing the stability of the MSRs for storing the secret key. In particular, we use the unused MTRR registers and IORR registers, which can be identified by checking the valid bit. On our AMD platform, there are eight MTRR pairs per core plus two shared core IORR registers. When the valid bit is cleared, the register is not used by the system. The bits provided by these registers are large enough to store a short encryption key.

5.4 Cache based I/O Storage

We perform a simple experiment to verify that the cache based I/O storage is able to keep the sensitive data in the cache only. Similarly, a repeating pattern *0x12345678* is written into the HIveS memory. Now the pattern should be stored in the cache. Next, we execute an *INVD* instruction, which invalidates all cache content without writing them back to the physical memory. If the pattern is indeed in the cache, after the execution of *INVD* instruction, such written pattern should no longer be observable. In our experiment, since the memory read back after *INVD* is not *0x12345678*, and therefore the modifications to the memory we wrote was truly stored in the cache. However, when the processor is busy, such contents stored in the cache is flushed out to the physical DRAM in a very short time.

6. HIVES LIMITATIONS AND COUNTERMEASURES

6.1 HIveS Limitations

Though the prototype shows promising potential on using HIveS to conceal malicious code and sensitive data in HIveS memory, the system has some limitations.

First, since the basic idea behind HIveS is the manipulation of physical address layout, system architecture with a fixed or reliable way to retrieve the physical address layout is not vulnerable to this attack. Furthermore, our implementation of HIveS relies on manipulating hardware registers in the AMD processor [2], therefore porting of the malware to other platforms requires careful design changes and examinations. As with most other advanced anti-forensic rootkits, HIveS requires kernel privilege to manipulate the system registers, and thus it is not available to user space malwares.

Second, HIveS achieves stealthiness by redirecting memory access on the hardware level. This inherently implies all software based accesses to the memory are redirected. Therefore HIveS cannot be used to store the current executing code. As a result, malware code that utilizes this storage, as presented in our prototype, is left in the memory and could be captured and analyzed by forensic examiners.

Lastly, HIveS focuses on defeating the software based memory acquisition approaches, so it has to be augmented with other anti-forensic mechanisms to defeat the hardware based memory acquisition approaches. Those mechanisms increase the complexity of HIveS and possibly make the targeted system unstable.

6.2 Countermeasures

HIveS is a system to subvert the organization of physical address layout. In order to defeat HIveS, it is important to get a reliable representation of the true address layout. Unfortunately, there is currently no architecturally supported method to verify the truthfulness of the layout. For the rest of the discussion, we focus on how to defeat our HIveS implementation on AMD platforms.

First, we know that the manipulation of IORR is essential in HIveS, and since IORR registers are only available on AMD processors, HIveS as a system does not work well in Intel family processor from HIveS, though several techniques we presented in HIveS may still apply.

Second, the use of IORR can be a good hint of the presence of HIveS memory in the system. It can be identified by simply inspecting the valid bit in the IORR mask register. A forensic examiner can also detect the existence of HIveS by measuring the timing for memory operations. When I/O shadowing is enabled, all memory access goes through the I/O bus and takes much longer to complete.

Note that legitimate I/O devices may also use the IORR to map physical memory address to the I/O space. For instance, AGP video driver in Linux kernel uses the IORR register in some cases. Since AMD provides two pairs of IORR registers, a forensic examiner can also examine the difference of the two pairs. Yet, such analysis could be quite system dependent. It is difficult to determine if the use of IORR registers is benign or malicious. The forensic examiner may assume the use of IORR is malicious and directly modify the value of this IORR register to reveal the memory contents. However, if the use of IORR is benign and system originated, such direct manipulation may crash the system. A more conservative approach might be to read out the contents from the address range without modifying any of the registers. Even though this might cause system instability as well, the probability of a system catastrophic failure caused by memory read is much smaller. If all the bytes read back are all identical values $0xFF$ or $0x00$, then most likely there is no real I/O device behind these I/O addresses.

Finally, HIveS can be detected by Cold Boot if we don't apply the HIveS extensions such as RAM-less encryption. Forensic examiners can first dump the registers, including all the MSRs and debugging registers from all processor cores. All system cache can then be flushed back into memory. Then the system is reset to extract memory content exploiting the memory remanence characteristics. This however changes many system configurations in the system as well as some memory contents, which violates the forensic principle of not altering the crime scene.

7. EXTENSION AND FUTURE WORK

While some of the limitations discussed above are unavoidable, such as dependence on architecture and operating system, some others can be overcome. We discuss possible extensions of HIveS in this section.

7.1 Eliminating Memory Traces

One of the limitations discussed above is that the storage can only be used to store data collected by the malware instead of protecting the entire malware. The forensic examiner might be able to analyze the malware memory to discover the manipulation of address layout. One key insight is that, the physical address layout manipulation performed by HIveS is very infrequent if not one time. Furthermore, the amount of code required to alter this layout is very small, most likely a single instruction or two. Taking our prototype as an example, the kernel module initialization routine can use one instruction to change the IORR register then immediately erasing the previous instruction by zeroing it. This leaves a very small time window, three instruction execution time, for the forensic examiner to capture the image and discover the use of IORR in the malware. The practical chance of catching such moment is close to zero.

7.2 Extending HIveS to Intel Platforms

The IORR registers HIveS exploited to alter the physical address layout is AMD specific unfortunately. To the best of our knowledge, there is no such MSR in the Intel platform [3]. This does not imply HIveS is impossible on Intel. Malware authors will need to find another way to alter the physical address layout to launch the attack. For example, Intel Memory Controller Hub (MCH) chipsets also provide capability to recover addressable memory space lost to MMIO space [1]. One can modify the REMAPBASE and REMAPLIMIT register in the chipset to manipulate the physical address layout (also known as system address space in Intel manuals).

7.3 HIveS for Defense

Techniques in computer security are like weapons, it can be used either to defend the righteousness or cause damage to the society. For instance, virtual machine based rootkit (VMBR) introduced by Rutkowska et al. [29] has been used to capture host image in forensic memory analysis [23, 40]. Similarly, though we present HIveS as a powerful anti-forensic tool, it can certainly be developed and used as a defense tool to protect sensitive data against malicious memory scanning. For example, application passwords can be stored in HIveS memory without having to worry about malware reading the passwords from the physical memory.

8. RELATED WORKS

There is an ongoing arms race between the attackers and the forensic examiners in computer forensics [34, 20, 25, 37]. Memory forensic analysis is becoming an indispensable tool for forensic examiners nowadays, and they have two ways to acquire computer memory: software based methods that use a trusted software module to access memory through the CPU processor [11, 21, 39, 28, 35, 14, 31, 36, 12] and hardware based methods that rely on dedicated I/O devices to access physical memory image via Direct Memory Access [10, 35, 27, 6].

Software based memory acquisition techniques rely on the CPU processor to acquire physical memory through the operating system. Unfortunately, after recognizing this dependency, attackers have developed anti-forensic techniques to compromise the memory acquisition process, such as directly modifying the acquisition module or the OS kernel data structure [8, 20, 34, 15], using rootkits to hook operating system APIs [32], or installing a thin hypervisor on the fly [29].

To defeat those anti-forensic techniques, Stüttgen et al. [35] propose an anti-forensic resilient method to acquire physical memory by eliminating its dependence on the operating system routines and data structures. Schatze [31] proposes to bootstrap a trusted new execution environment from the normal one to make sure that the operating system is free of malware. System management mode (SMM) can also be used to create a trusted isolated execution environment [28, 39]. Some researchers propose to go deeper than the operating system level and use hardware virtualization to avoid the memory acquisition software being subverted by rootkits [23, 40].

Stüttgen et al. suggest that the memory acquisition process can be trusted if the acquisition module has not been tampered with and all the operations are performed without relying on the operating system or any other untrusted software [35]. However, in this paper, we show that this assumption is not true. The main reason is that the physical memory layout seen by the processor can be manipulated through the hardware configurations on the chipset. Attackers can misuse hardware configurations to modify this layout and conceal the presence of malware.

A number of hardware based memory acquisition methods have been developed recently [27, 38, 5, 10], using a trusted peripheral device to capture the physical memory image via DMA. Since it does not rely on the CPU processor to get the physical memory, the hardware based approaches can successfully prevent those anti-forensic techniques that are originally designed to defeat the software based approaches. However, Rutkowska [30] shows that it is possible to present a different view of the physical memory to the peripherals by reprogramming the north bridge. Therefore, in-memory data acquired by DMAs could be compromised as well [28, 35].

A special type of memory acquisition technique relies on the unique remanence property of physical DRAM [18, 11]. Despite the popular belief that volatile contents in DRAM are gone once the computer resets or powers off, Halderman et al. [18] demonstrate a Cold Boot attack that can reliably recover the contents in the memory modules even after the power has been cut off for a short period of time. Though the original Cold Boot is demonstrated as an attack to steal cryptographic keys and other sensitive data from the RAM, it is also an effective method that can be used for reliably acquiring physical memory.

9. CONCLUSIONS

In this paper, we propose a different approach to anti-memory forensic. Instead of looking at ways to conceal presence by operating system object manipulation, we can defeat current memory acquisition methods by manipulating the physical address layout, a design architectural feature on modern x86 platforms.

HIveS is an anti-forensic mechanism to conceal in-memory data shadowed behind the I/O address space. Besides I/O Shadowing technique to prevent forensic memory acquisition tools from reading the HIveS memory contents via processor, we also use Blackbox Write and TLB Camouflage to enable the attacker exclusive write access and provide a single malicious core exclusive read and write access, respectively. Furthermore, we propose several add-ons to the basic framework to further hide from physical memory forensics.

A prototype of HIveS is built on an AMD platform to show that none of the popular memory acquisition tools we tested can capture the memory data protected by HIveS. Several countermeasures are discussed in the end. In the future, we intend to further investigate possible mechanisms to retrieve trustworthy physical address layout.

Acknowledgment

This work was supported in part by US National Science Foundation under grants CNS-1217889 and CNS-1405747. Dr. Kun Sun's work is partially supported by U.S. Army Research Office under Grant W911NF-12-1-0060, W911NF-12-1-0448 and U.S. Office of Naval Research under Grant N00014-11-1-0471

10. REFERENCES

[1] Intel Chipset 4 GB System Memory Support . Feb 2005.

[2] Advanced Micro Devices. Amd64 Architecture Programmer's Manual. Vol. 2, may 2013.

[3] Intel 64 and IA-32 Architectures Software Developer's Manual. sep 2013.

[4] Advanced Micro Devices, Inc. BIOS and Kernel Developer's Guide (BKDG) For AMD Family 15h Processors, Rev 3.23.

[5] R. BBN. Fred: Forensic ram extraction device. `http://www.digitalintelligence.com/products/fred/`.

[6] M. Becher, M. Dornseif, and C. N. Klein. FireWire All Your Memory are Belong to us. *Proceedings of CanSecWest*, 2005.

[7] N. Beebe. Digital forensic research: The good, the bad and the unaddressed. In *Advances in digital forensics V*, pages 17–36. Springer, 2009.

[8] D. Bilby. Low down and dirty: Anti-forensic rootkits. *BlackHat Japan*, 2006.

[9] D. Bovet and M. Cesati. *Understanding the Linux kernel*. O'reilly, 2007.

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

[11] E. Chan, S. Venkataraman, F. David, A. Chaugule, and R. Campbell. Forenscope: A framework for live forensics. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 307–316. ACM, 2010.

[12] M. Cohen, D. Bilby, and G. Caronni. Distributed forensics and incident response in the enterprise. *digital investigation*, 8:S101–S110, 2011.

[13] N. R. Council. Strengthening Forensic Science in the United States: A Path Forward. `https://www.ncjrs.gov/pdffiles1/nij/grants/228091.pdf`, 2009.

[14] D. Farmer and W. Venema. *Forensic discovery*, volume 18. Addison-Wesley Reading, 2005.

[15] E. Florio. When malware meets rootkits. *Virus Bulletin*, 2005.

[16] S. L. Garfinkel. Digital forensics research: The next 10 years. *Digital Investigation*, 7:S64–S73, 2010.

[17] L. Guan, J. L. amd Bo Luo, and J. Jing. Copker: Computing with Private Keys without RAM. In *In Network and Distributed System Security Symposium (NDSS)*, 2014.

[18] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.

[19] R. Harris. Arriving at an anti-forensics consensus: Examining how to define and control the anti-forensics problem. *digital investigation*, 3:44–49, 2006.

[20] T. Haruyama and H. Suzuki. One-byte modifications for breaking memory forensic analysis. *Black Hat Europe*, 2012.

[21] E. Libster and J. D. Kornblum. A proposal for an integrated memory acquisition mechanism. *SIGOPS Oper. Syst. Rev.*, 42(3):14–20, Apr. 2008.

[22] Y. Lu, L. Lo, G. Watson, and R. Minnich. CAR: Using Cache as RAM in LinuxBIOS. `http://rere.qmqm.pl/~mirq/cache_as_ram_lb_09142006.pdf`.

[23] L. Martignoni, A. Fattori, R. Paleari, and L. Cavallaro. Live and trustworthy forensic analysis of commodity production systems. In *Recent Advances in Intrusion Detection*, pages 297–316. Springer, 2010.

[24] T. Müller, F. C. Freiling, and A. Dewald. Tresor runs encryption securely outside ram. In *USENIX Security Symposium*, 2011.

[25] T. Newsham, C. Palmer, A. Stamos, and J. Burns. Breaking forensics software: Weaknesses in critical evidence collection. In *Proceedings of the 2007 Black Hat Conference*, 2007.

[26] J. Pabel. Frozencache: Mitigating cold-boot attacks for full-disk-encryption software. In *27th Chaos Communication Congress*, 2010.

[27] N. L. Petroni Jr, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot-a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, pages 179–194, 2004.

[28] A. Reina, A. Fattori, F. Pagani, L. Cavallaro, and D. Bruschi. When hardware meets software: A bulletproof solution to forensic memory acquisition. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 79–88, New York, NY, USA, 2012. ACM.

[29] J. Rutkowska. Subverting vistatm kernel for fun and profit. *Black Hat Briefings*, 2006.

[30] J. Rutkowska. Beyond the CPU: Defeating hardware based RAM acquisition. *Proceedings of BlackHat DC 2007*, 2007.

[31] B. Schatz. Bodysnatcher: Towards reliable volatile memory acquisition by software. *digital investigation*, 4:126–134, 2007.

[32] D. Sd. Linux on-the-fly kernel patching without lkm. *Volume 0x0b, Issue 0x3a, Phile# 0x07 of 0x0e-Phrack Magazine-http://www. phrack-dont-give-a-shit-about-dmca. org/show. php*, 2001.

[33] P. Simmons. Security through amnesia: a software-based solution to the cold boot attack on disk encryption. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 73–82. ACM, 2011.

[34] S. Sparks and J. Butler. Shadow walker: Raising the bar for rootkit detection. *Black Hat Japan*, pages 504–533, 2005.

[35] J. Stüttgen and M. Cohen. Anti-forensic resilient memory acquisition. *Digital Investigation*, 10:S105–S115, 2013.

[36] J. Sylve. Lime-linux memory extractor. *ShmooConâĂŹ12*, 2012.

[37] S. Vömel and F. C. Freiling. A survey of main memory acquisition and analysis techniques for the windows operating system. *Digital Investigation*, 8(1):3–22, 2011.

[38] J. Wang, A. Stavrou, and A. K. Ghosh. Hypercheck: A hardware-assisted integrity monitor. In *RAID*, pages 158–177, 2010.

[39] J. Wang, F. Zhang, K. Sun, and A. Stavrou. Firmware-assisted memory acquisition and analysis tools for digital forensics. In *Systematic Approaches to Digital Forensic Engineering (SADFE), 2011 IEEE Sixth International Workshop on*, pages 1–5. IEEE, 2011.

[40] M. Yu, Q. Lin, B. Li, Z. Qi, and H. Guan. Vis: virtualization enhanced live acquisition for native system. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 13. ACM, 2011.