Memory Forensic Challenges under Misused Architectural Features

Ning Zhang*, Ruide Zhang*, Kun Sun†, Wenjing Lou*, Y. Thomas Hou*, Sushil Jajodia†
*Virginia Polytechnic Institute and State University
†George Mason University

***Abstract*—With increasingly complex cyber attacks occurring every day, memory-based forensic techniques are becoming instrumental in digital investigations. Forensic examiners can unravel what happened on a system by acquiring and inspecting in-memory data. However, the foundation of this analysis can be invalidated if the memory acquisition has been altered. In this paper, we study the feasibility of malicious software misusing architectural features to sabotage memory forensics. The misuse of two architectural features, namely physical address layout and secure containers, is presented.**

The first architectural feature explored in this paper is the physical address layout. It is used by the northbridge to route memory access to either physical memory or I/O devices on x86 platforms. Observing this design choice, we propose Hidden in I/O Space (HIveS), which manipulates CPU registers to alter the physical address layout to conceal memory. The system uses a novel *I/O Shadowing* technique to lock a memory region named *HIveS memory* into I/O address space to prevent access. Two novel techniques, *Blackbox Write* and *TLB Camouflage*, are developed to further protect the unlocked HIveS memory against memory forensics while allowing access for attackers.

The second architectural feature explored in this paper is hardware-aided secure execution technology. More specifically, hardware-enforced memory encryption in Intel secure guard extension (SGX) is used in *Malicious Enclave Software (Malclaveware)* to prevent introspection and memory forensics.

A prototype of HIveS is built and tested against a set of memory acquisition tools for both Windows and Linux running on the x86 platform. Malclaveware is also prototyped in Windows to demonstrate the risk. More importantly, we proposed countermeasures and mitigations for the newly discovered attacks. Through these discussions, we aim to raise the awareness of the potential risks of misusing hardware architectural features.

I. 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 [1], [2].

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 forensics, adversaries are starting to hide the presence of malicious code and data only in the memory [3], [4], [5]. To tackle this problem, forensic examiners are increasingly relying on live memory forensics to uncover the malicious content in the memory [1].

Memory forensics often involve memory acquisition and memory analysis. In memory acquisition, the system memory is collected as an image. This image is then examined by the forensic examiner in memory analysis. The accuracy of memory analysis builds on the sound acquisition of system memory. 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 [6], [7]. Hardware-based solutions often utilize dedicated I/O devices, such as a network interface card, to capture the physical memory image via direct memory access (DMA) [8], [9], [10] with the processor totally bypassed. Some hardware-based approaches use the remanence of physical memory to extract sensitive data from the memory modules in systems that are powered off for a short time [11], [12].

To counter live memory forensics, attackers have developed various anti-forensic techniques to sabotage the memory acquisition process [13]. 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 [14], [15], [5], hooking operating system APIs [16], or installing a thin hypervisor on the fly [17]. Based on this observation, it is suggested that the memory acquisition process can be trusted if the acquisition software along with all its dependencies has not been tampered with [6]. However, in this paper, we show that architectural features in modern systems can be used to sabotage the memory acquisition step in memory forensics; and more importantly, what forensic examiners can do to mitigate these threats. The misuse of two architectural features is presented. Physical address remapping is misused in *Hidden in I/O Space (HIveS)* as an operating system (OS) agnostic anti-forensic mechanism. Hardware-assisted secure containers are misused in *Malicious Enclave Software (Malclaveware)* to provide memory encryption and introspection shielding for malware.

Misusing physical address layout-HIveS. The first architectural feature we explore is the physical address layout. 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 the location 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 the 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's physical address layout while the system is in an operational state by modifying registers in the processor. With this manipulated address layout, HIveS can conceal a memory region called *HIveS memory* from being observed and acquired by memory forensics tools.

The basic idea of HIveS is to map (or lock) 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 content 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 several existing anti-forensic techniques, such as *RAM-less encryption* and *Cache-based I/O storage*, to defeat hardware-based memory acquisition approaches.

Misusing secure computing-Malclaveware. The second architectural feature we explore is the recently widely used secure execution technology [18], [19]. Strong hardware-assisted execution environments are capable of protecting applications even from the traditional highest-privilege software, the operating system. While this property is very attractive for security-critical tasks, such protection can also be misused by attackers to deny forensic memory acquisition. To demonstrate the potential threat, we adopt the secure enclave technology of Intel SGX [18] in malware, and call the new family *Malclaveware*. Transparent, hardware-based CPU-bound memory encryption of SGX is used in Malclaveware to deny forensic examiner access to plaintext memory. Using the remote attestation, the malware will execute only in the designated environment, effectively evading most of the current sandbox-based detections [20], [21]. Lastly, by protecting the malware execution in the secure enclave, it would be impossible for the host-based protection to introspect the malware internals. We apply the design to ransomware [22], [23], and build a highly targeted ransomware prototype that exploits the protection of SGX.

Instead of presenting only the attacks, we reflex on the root causes of insecurity and provide discussion on the countermeasures. We propose several countermeasures for detecting and mitigating HIveS and Malclaveware. One approach to detect HIveS is to directly inspect the CPU registers. However, it remains a challenge to distinguish proper configurations from malicious usages. We find application whitelisting to be an important step towards preventing secure container misuses.

To summarize, we make the following contributions:

- We identify a general class of anti-memory-forensic technique that exploits hardware architectural features and present two attacks within the class, HIveS and Malclaveware.
- We present *HIveS*, an OS agnostic system that exploits hardware features on x86 platform to conceal memory in I/O space, effectively subverting the foundation of memory acquisition. We develop two novel techniques, *Blockbox Write* and *TLB Camouflage* to enable covert operations on the unlocked HIveS memory against memory forensics. A prototype of HIveS is built on the x86 platform to demonstrate its capability for concealing the HIveS memory against several of the most updated memory forensic tools on both Windows and Linux.
- We present *Malclaveware*, which takes advantage of hardware-assisted secure execution technology to prevent memory forensics and system introspection. We apply the concept of Malclaveware to ransomware to create a new breed of ransomware that is highly targeted and able to hide the file encryption key-even in the presence of a higher-privilege forensics subsystem. Experiments on our prototype show little performance impact.
- We provide discussion on the countermeasures and limitations of the newly presented attacks to fuel development of future system defense.

II. Background

A. Physical Address Layout

The entire range of memory addresses accessible by x86 processors is often referred to as *physical address space*. Contrary to popular beliefs, 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).

B. Allocations in Physical Address Space

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 [24]. The layout is configured via several configuration registers in the northbridge (NB) and the processor. The DRAM Base/Limit register pair is among the earliest ones configured by the BIOS. They define the ranges of physical address space mapped to DRAM in the northbridge. Any access to these areas will be forwarded to the DRAM Controller (DCT). These registers are configured by the BIOS

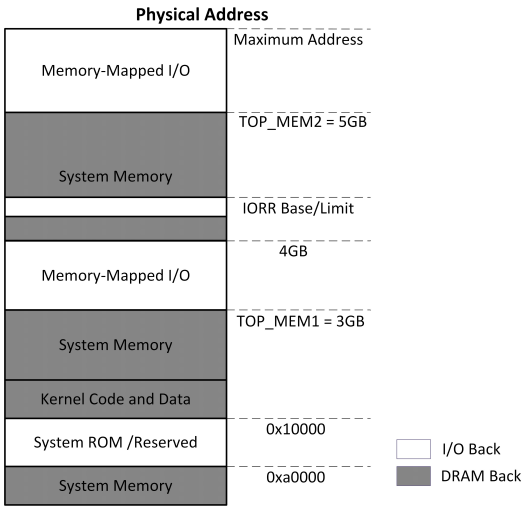


Fig. 1. Physical Address Layout on AMD Architecture

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)-the 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 [25]. 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 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 shape the layout is also MSR in the processor. They are the Input Output Remap Registers (IORR). This 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. The registers are designed to enable system software to shadow the ROM device in memory to improve system performance.

III. Misusing Physical Address Layout - HIveS

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 that 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 *HIveS memory*.

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 northbridge. The MUX inside the northbridge is responsible for forwarding the memory request to either the DRAM controller or the southbridge, 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 southbridge. 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 the 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 modify kernel data structures or operating system APIs, HIveS cannot be detected by checking the integrity of the OS.

A. 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 as an anti-forensics 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 content in the DRAM memory are shadowed by the memory-mapped I/O (MMIO). Among the various controls that shape 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 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

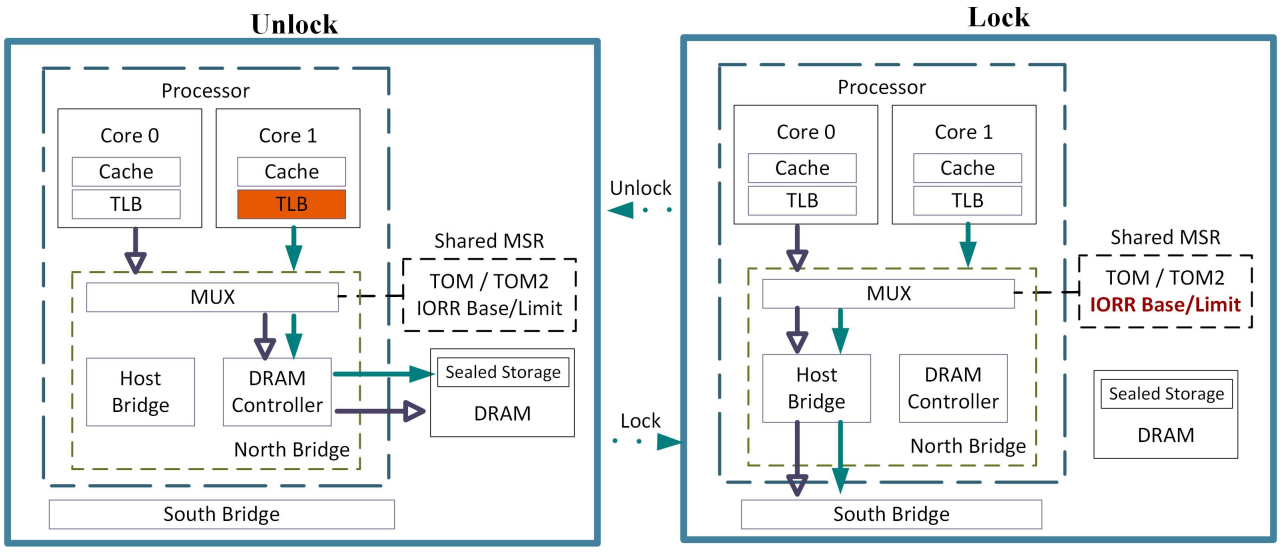


Fig. 2. Architecture of HIveS - Lock and Unlock state of the HIveS storage by manipulating physical address space using IORR registers

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 [25], 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. The IORR base register contains the starting address of the IORR region, as well as two important flag bits, *WrMem* and *RdMem* [25]. When these two bits are set to 1, the northbridge 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 the 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 IORRs has little impact on the OS. Furthermore, IORR registers offer great adaptability in both the location and size of the HIveS memory.

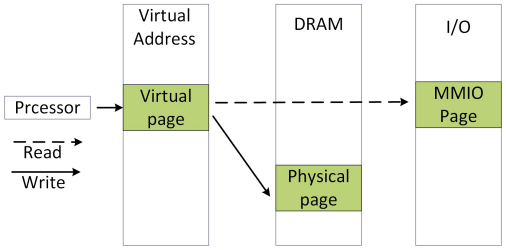


Fig. 3. Blackbox Write - asymmetric read/write destination for memory access

B. Exclusive Access in the Unlocked State

The HIveS memory in the unlocked state is designed to allow 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 attacker 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 content in the HIveS memory, and then lock it by enabling the I/O shadowing. However, it becomes a challenge to ensure 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.

1) Blackbox Write

The key idea behind *Blackbox Write* technique is the asymmetric memory access to the DRAM between the attacker and forensic examiner. Attackers that utilize the HIveS memory to store the stolen information often require only write access to the memory region, and occasionally read it during exfiltration. On the other hand, forensic examiners are interested only on reading the memory content. To preserve the integrity of the evidence, memory forensic tools always read the memory content and never write to the memory.

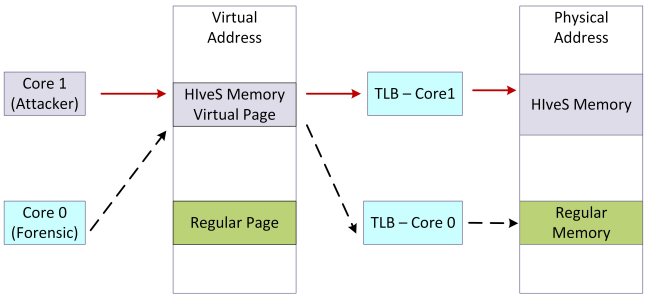


Fig. 4. TLB Camouflage - core specific memory address mapping

Based on the above asymmetric operations,*Blackbox Write* redirects all memory read requests to the I/O space by setting $RdMem = 0$ in IORR and redirects all the memory write requests to the HIveS memory by setting $WrMem = 1$. Under this setting, attackers can write new content into the HIveS memory while preventing forensic examiners from analyzing it. Furthermore, since 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 [26]) is returned instead. Since the IORR register allows the creation of such redirection on any physical address, the attacker can always use the physical address range that does not correspond to a valid I/O device for HIveS memory. Note that HIveS memory is also writable by the forensic examiner. However actively modifying memory is 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 fixed amount of user key strokes have been stealthily recorded, the keylogger needs to send the data to a remote server. One approach is to temporarily modify the IORR to allow access then disable the access after data exfiltration. On the other hand, instead of unlocking processor's read access to the HIveS memory, the attacker can also exfiltrate the data using DMAs. For example, the attacker who controls the kernel can program the network interface adapter that is commonly included in most of the modern computing systems to exfiltrate the stolen data hidden in HIveS memory.

2) TLB Camouflage

While 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, it does not meet the requirement for malware that requires continue read and write access to 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 content to be accessed only by the processor core that is running the malicious software.

Modern operating systems enable paging mechanism to translate virtual memory addresses into physical memory addresses before passing the memory access requests to the DRAM Controller (DCT) [24]. The Translation-Lookaside Buffer (TLB), also known as page-translation caches, is designed to reduce the performance penalty during the time-consuming address translation process [25]. 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. Additionally, 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 [25], [27]. 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. The idea behind TLB Camouflage 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 make sure that there is no pre-existing translation stored already for the newly allocated page. 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. Furthermore, 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 a false mapping. 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 improves the usability of HIveS memory, which can be used not only as temporary storage with few interactions, but also as 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 [6]. For example, a DMA-based memory acquisition device [28] acquires memory bypassing the processor. 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.

C. 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 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, she can use the Blackbox Write technique. Moreover, if the attacker also

needs to frequently read the memory content, she can use the TLB Camouflage technique. Table I shows the different access privileges to the HIveS memory for both attackers and forensic examiners when applying different anti-forensic techniques.

TABLE I
COMPARISON OF ACCESS TO HIVES MEMORY

	Attacker Rd	Attacker Wr	User Rd	User Wr
I/O Shadowing	no	no	no	no
Blackbox Write	no	yes	no	yes
TLB Camouflage	yes	yes	no	no

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.

D. 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 path. 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 requests 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 images via direct memory access, which totally bypasses the processor hardware configurations made by HIveS. Moreover, the Cold Boot technique [12] exploits the physical remanence property of memory chips to directly extract sensitive data from the chips. The 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 several existing techniques in HIveS, including IOMMU, RAM-less encryption, and Cache-based I/O storage.

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 addresses to physical memory addresses performed by the MMU, IOMMU is a hardware device that translates device DMA addresses into proper physical memory addresses [27]. 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 northbridge. The northbridge 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 northbridge 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 [29].

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*. The basic idea of RAM-less encryption is to encrypt all the memory content in the HIveS memory with a secret key stored in CPU registers [30], [31]. 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 content of HIveS are still being protected, because the encryption key in the CPU registers is lost forever due to the system reset. The basic idea of cache-based I/O storage is to save HIveS memory only in the CPU cache [32], [33], [34] 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 content 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. However, it remains a challenge to maintain the content in the cache considering the limited cache control provided by the x86 architecture [25], [27].

IV. HIVES IMPLEMENTATION AND EVALUATION

We build a prototype of HIveS on an x86 desktop with an 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 several 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. The source code and dataset can be found in [35].

TABLE II
VERIFICATION AGAINST MEMORY FORENSIC TOOLS

Tool	Version	OS	Detect HIveS
UnitTest	1.0	Linux	No
LiME	1.1	Linux	No
MemDump	1.01	Linux	No
DD	8.13	Linux	No
WinPmem	2.3.1	Windows	No
Mem Marshall	1.0	Windows	No
Memoryze	3.0	Windows	No
Dumpit	1.3.2	Windows	No

A. 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 bits of the physical address, and inserts it into bits 12 to 47 in the I/O Range Base register, *MSRC001_0016*, since the physical addressing in AMD x64 is 47 bits. 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 [25], 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 the repeating pattern of *0x12345678*. The cache is also flushed to make sure the 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 content 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 II summarizes the tools that we used in our experiments. For all the tools we tested, 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.

B. 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 the 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. Further, 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 content 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.

C. HIveS Extensions

For RAM-less encryption, we use a secret key to XOR the plain text instead of using the AES function, since the feasibility of RAM-less encryption has already been verified [30], [31] 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.

For 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*, the modifications to the memory we wrote were truly stored in the cache. However, when the processor is busy, such content stored in the cache are flushed out to the physical DRAM in a very short time.

V. Misusing *Secure* Computing - Malclaveware

Based on the same intuition of HIveS, we study another widely-used architectural feature, hardware-assisted secure execution. More specifically, we present a case study of adopting the Intel SGX secure execution in malicious software, which we call *Malclaveware*.

A. Secure Computing - SGX

As modern software becomes increasingly complex, producing bug-free program binaries remains an open research challenge. To provide trusted execution environments for security-sensitive tasks, system designers often rely on hardware-assisted secure containers [18], [36], [37]. The Intel SGX technology is a newly developed instruction set extension to provide secure execution on commodity processors [18]. SGX allows an application to place sensitive portions of its program inside a secure container, called *enclave*. The confidentiality and integrity of the application code and data inside the enclave is protected by the hardware, even from a compromised privilege operating system and certain hardware attacks such as Cold Boot attack [12].

B. Ransomware and Cryptography

Malicious software continues to be one of the biggest security threats today. *Ransomware* [38] is a type of malware that denies users the availability of computing resources, often by encrypting user files or locking the workstation. Though the concept of ransomware is not new [38], it has gained momentum lately due to the growing importance of data in computing systems. Coupled with recent advancements in crypto-currency [39], ransomware offers cyber criminals an easy way to generate significant amount of revenue safely. IBM found that spam emails containing ransomware increased 6,000 percent in 2016 compared to 2015 [40]. The FBI expected the costs of financial loss due to ransomware to reach as much as one billion US dollars for year 2016 [40].

The primary method used by ransomware to deny users access to their data is by encrypting the content with a secret key generated by the attacker. Victims can obtain the secret decryption key from the attacker only when the requested ransom is paid in full, commonly via cryptocurrency, such as Bitcoin [39]. Recent ransomware variants often use a combination of symmetric key and asymmetric key cryptography. Symmetric cryptography, such as AES, is generally orders of magnitude faster than the asymmetric counterpart. However, when the secret key is recovered via memory forensics, it can be used to decrypt the victim files. On the other hand, asymmetric cryptography allows the attacker to encrypt files using the public key, and the content can only be decrypted with the private key, which cannot be recovered in the victim system via memory forensics. To reap the benefit of both worlds, advanced ransomware generates a random key to encrypt victims' files using symmetric cryptography, then encrypts the random encryption key using the public key of the attacker. Such an approach provides both efficiency and secrecy, and is becoming more widespread in newer versions of ransomware such as CryptoLocker [22]. Furthermore, the cryptographic implementation of these malware has also evolved from naive

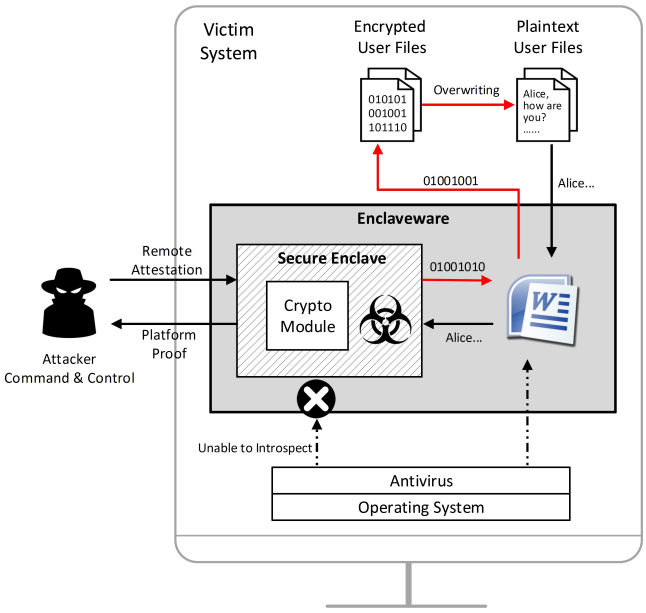


Fig. 5. Architecture of Malclaveware- hiding behind *secure* computing

use of Windows CryptoAPI, which can be hooked easily for introspection, to statically-linked OpenSSL code that is more difficult to analyze [41].

However, regardless of the implementation, the use of different types of cryptography in ransomware forces the attacker to make the tradeoff between secrecy and efficiency. He can either use symmetric cryptography and risk the keys being extracted by memory forensics, or use asymmetric cryptography and risk being discovered before finishing the encryption of victims' files.

C. Malclaveware

Malware can be significantly harder to counter when it is constructed with the SGX secure execution technology. We call this new breed of malware Malclaveware. An overview of the design is shown in Fig. 5. Malware is separated into two parts, the benign portion of the code runs in the application space, while the malicious logic is shifted into the secure enclave that is shielded from the operating system.

1) Enclave Attestation for Anti-Analysis

SGX supports remote software attestation with the CPU as the trusted computing base (TCB) [18]. It allows a remote party to verify cryptographically the software that is loaded inside the enclave and its execution environment. Shared secrets generated during the attestation process can be used to bootstrap secure communication between the code inside the enclave and the remote party.

One of the most powerful methods for malware detection today is through behavior-based dynamic analysis. The key component of these malware scanning engines is to observe the behavior of unknown software in a controlled environment [20], [21]. To resist such dynamic analysis, malware authors started to embed environment detection capabilities into the binary [21]. The execution timing, vendor-specific CPUIDs, or the existence of paravirtualization driver are among the most commonly used features for detection. How-

ever, it remains an open challenge to allow users to reliably verify the execution environment remotely.

The remote attestation enabled by the Intel SGX fundamentally changes the problem. The SGX platform offers remote users the ability to verify the execution environment, including the platform specification. The signature generated through remote attestation is cryptographically verifiable. Upon successful infiltration to the victim systems, ransomware typically connects back to the attacker server immediately for key generation. However, with the support of SGX, the attacker can verify the execution environment before proceeding. If a virtual environment is detected, the malware can terminate to avoid detection. To make program analysis more difficult for the defender, the attacker can even encrypt the malicious payload and only decrypt it when the environment is attested. The ability to launch such highly targeted attacks significantly diminishes the effectiveness of behavior-based malware detection systems.

2) Enclave Protection for Malware Protection

The enclave is entered by invoking the EENTER instruction. Execution in enclave can either run to completion and exit using EEXIT, or until it encounters an exception or interrupt. Enclave memory protection is accomplished by encrypting all data outside the processor boundary, and decrypting them only within the processor for computation. Upon context switch, the application context is saved and sanitized by SGX before handing the control to the OS. The current generation of ransomware encrypts files using statically linked symmetric key cryptographic functions. During the file encryption phase, the symmetric key could be captured by the operating system by inspecting the process memory. With the OS transparent memory encryption capability provided by Intel SGX, it is possible to conceal not only the malicious code but also the data, such as secret keys from memory forensics.

VI. Malclaveware Implementation and Evaluation

We built a prototype of the Malclaveware using the Intel SGX SDK platform. The experimental platform runs Windows 10 on the Intel NUC, powered by Intel i7-6770HQ with 8 GB memory. According to the vendor specification, the read and write speed of our SSD is 560 MB/s and 400 MB/s, respectively. Our prototype invokes the AES-NI instruction inside the enclave to encrypt all file buffers. The source code and dataset can be found in [35]. SGX enclave applications need to be signed using the developer key. However, due to the lack of an Intel approved developer key, we evaluate the prototype of Malclaveware in the debug mode. Despite running in debug mode, enclave protection discussed previously remains active, and the OS can only access data in the enclave through special instructions, such as EDBGRD [42]. Therefore, there are two strategies of deployment for malware authors. To make an attempt to evade the unsuspected victims, he can choose to use debug mode to avoid identification of the developer key. On the other hand, he can also deploy Malclaveware in release mode. This way, the malicious content is guaranteed to be protected by the hardware, which is likely the preferred method for high-value targets.

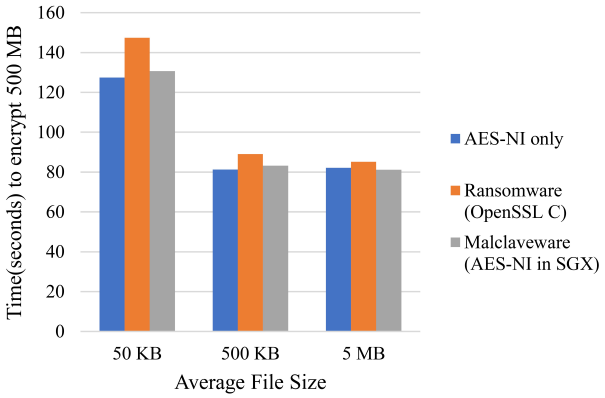


Fig. 6. Malclaveware File Encryption Speed Evaluation

A. Memory Protection Validation

We want to verify that the sensitive cryptographic key is indeed protected under the SGX protection even under debug mode, since that could be one of the deployment models. We dump the register content and system memory during the file encryption of Malclaveware using Dumpit v1.3.2 [43]. To validate the system memory image, we search the raw image and found the known key values outside the enclave. To verify that secrets inside the enclave are not in the image, we generated a new secret key inside the enclave by calculating the XOR of the key outside the enclave and a random string. This newly generated secret key is not found in the memory image.

B. Malclaveware Performance Evaluation

We also evaluated the performance impact on encryption in Malclaveware for using the SGX environment. Fig. 6 shows the comparison of three implementations, which are using AES-NI instructions to encrypt files inside the SGX enclave, using AES-NI to encrypt files without using SGX and using C implementation of OpenSSL to encrypt files without using SGX. We use the OpenSSL AES C implementation as the baseline for current ransomware performance. AES-NI without SGX represents the performance while using hardware cryptography acceleration. Lastly, AES-NI inside the SGX enclave is our proposed system. To evaluate the tradeoff in different user environments, we also repeat the experiment using file repositories of different average file sizes to study the effect of file size on the performance. The average size of a word document is about 300KB, according to a study done by Microsoft [44]. Therefore, we created three repositories of text files with randomly generated content. The average file sizes are 50 KB, 500 KB, and 5 MB respectively. We can see that by adopting AES-NI encryption, it is possible to encrypt files faster than the current implementation of ransomware [22]. Furthermore, the performance impact of the enclave is very small across repositories of files with different average sizes.

VII. Countermeasures and Discussion

The goal of studying anti-forensics techniques is to move another step forward in digital forensics. While architectural

features can be misused by attackers to incapacitate digital evidence collection, they can also be clues for forensic examiners to uncover truth masked by the malware.

A. Countermeasures to HIveS

HIveS represents a general approach to hide malicious memory by subverting the organization of the 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 accuracy of the layout. For the rest of the discussion, we focus on countermeasures towards our implementation. In such, there are two ways to mitigate the attack.

First, the manipulation of IORR is essential in HIveS in achieving asymmetric memory read/write destination. The utilization of the register can be a good hint. It can be checked by inspecting the valid bit in the IORR mask register. The direction of memory operation to I/O bus can also be detected by the access time, since memory bus is considerably faster. On the other hand, legitimate I/O devices may also use the IORR to map physical memory address to the I/O space. For instance, a AGP video driver in the Linux kernel uses the IORR register in some systems. Furthermore, AMD provides two pairs of IORR registers; so the examiner also needs to examine the other pair, even if one might be used for legitimate purposes. If the creation of the physical address remapping is malicious, then the examiner can disable the IORR register to examine true memory content. On the other hand, if the use is benign, or if the use is system originated, directly manipulating such mapping can lead to system instability, causing loss of digital evidence. Given such risk, the examiner can read the suspicious address directly. Even though this might cause system instability, the probability of a system catastrophic failure caused by memory read is often smaller. If all the bytes read back are identical values 0xFF or 0x00, then most likely, there is no real I/O device behind these I/O addresses.

Second, HIveS can be defeated by physical forensic methods such as a Cold Boot [12], when HIveS extensions such as RAM-less encryption are not in use. To defeat HIveS with an extension, forensic examiners can first dump the registers, including all the MSRs and debugging registers from all processor cores, and flush all cache. This way, the examiner will have most of content outside of physical memory. The system can then be reset to extract memory content exploiting the memory remanence characteristics. We verified that it is possible to extract the memory of the malware after resetting the system to a clean state. This, however, changes many system configurations in the system and potentially violates the forensic principle of not altering the crime scene.

B. HIveS Limitation

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

First, similar to other anti-forensics rootkits, HIveS requires kernel privilege and is not available to user space malware. Furthermore, the code of the rootkit must be exposed to the operating system for execution. This is often referred to as the *rootkit paradox* [45]. While such common limitation among all rootkits is solved in HIveS, it pushes the boundary of the field by offering the ability to conceal the actual data stolen. Understanding data breath damage is often a much bigger forensic challenge for many cyber crime investigations, according to our conversations with the practitioners in the field. Second, since the basic idea behind HIveS is manipulation of the physical address layout, 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 [25], therefore porting of the malware to other platforms requires careful design changes. Lastly, HIveS focuses on defeating the software-based memory acquisition approaches, so it must be augmented with other anti-forensic mechanisms to defeat the hardware-based memory acquisition approaches. Those mechanisms increase the complexity of HIveS and sometimes make the targeted system unstable.

C. Discussion on HIveS

Extending HIveS to Intel Platforms: Our implementation of HIveS on the AMD platform misused the AMD specific IORR registers alter the physical address layout. To the best of our knowledge, there is no equivalent MSR in the Intel platform [27]. 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 [46]. 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). Our objective in this work is to raise awareness of the architectural features which a malware can use to manipulate physical address layout. From the perspective of forensic examiner, the key is to validate the physical address layout by examining registers such as IORR in AMD and REMAPBASE in Intel. We plan to investigate manipulation techniques on Intel platform as future work.

HIveS for Defense: Techniques in computer security are like weapons, they can be used either to defend righteousness or cause damage to society. For instance, the virtual machine-based rootkit (VMBR) introduced by Rutkowska et al. [17] has been used to capture host images in forensic memory analysis [47], [48]. 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 the HIveS memory without having to worry about malware reading the passwords from the physical memory.

D. Limitation and Countermeasure of Malclaveware

Malclaveware represents a general approach in using hardware-based secure computation mechanisms to conceal malicious applications. Secure computation, such as Intel TXT [27], ARM TrustZone [19], and SGX [18], provides an isolated execution environment for sensitive applications [18], [27], [19]. Using a small TCB, the environment is protected

from compromised components within the system. By design, content inside such environments often cannot be accessed by even the operating system [18], [19]. Therefore, once the malware is executed inside the protected environment, it would be very difficult to identify and remove them. The best approach to counter these malware is to deny their access to secure containers. Applications can be whitelisted for access to secure execution. Another common approach is to disable the unused advanced security features in the system.

Even though our implementation of Malclaveware with SGX as the secure computation environment poses a significant threat, there are several ways to mitigate the threat. First, the OS can verify if the enclave application is on the whitelist before loading the binary. Second, even though once the malware is inside the SGX enclave, it would be impossible for the OS to introspect on the program internals. Malclaveware remains a user space program that relies on the OS through system calls to perform meaningful actions. It is possible for the OS to analyze the behavior of an application by analyzing the system calls from the enclave-protected program. For example, Malclaveware invokes file system utilities in the OS to read and write files. Frequent file operations and asymmetric file read/write entropy can be used to detect ransomware [20]. Third, only enclave applications signed by Intel-issued developer keys can be deployed in the release mode. Malicious attackers might not want to use the enclave in release mode if they want to conceal their identities. Therefore, we verified the possibility for the forensic examiners to use debug instructions to introspect the malware running in debug mode for forensic analysis.

Malclaveware also has its own limitations. First, each malware needs to be designed specifically for a type of secure computation. Second, the use of hardware features limits the applicable target platforms of the technique to only those that support the secure computation hardware feature.

E. Discussion on Malclaveware

The concept of Malclaveware can be applied widely to different types of secure computation platforms. However, depending on the secure computation technology, the security properties could be different. For example, if malicious code can be loaded into the secure world of ARM TrustZone [19] as a trusted security module, it will be able to evade introspection from the normal world; however, it will remain exposed to introspection within the secure world. The anticipated Secure Encrypted Virtualization (SEV) technology from AMD [49] can also be used to create a container for malware that prevents introspection from the hypervisor.

VIII. Related Works

There is an ongoing arms race between the attackers and the forensic examiners in computer forensics [15], [14], [50], [51]. 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], [7], [10], [9], [6], [52], [53], [54], [55]; and hardware-based methods that rely on dedicated I/O devices to access physical memory image via Direct Memory Access [8], [6], [28], [56].

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 [3], [14], [15], [5]; using rootkits to hook operating system APIs [16]; or installing a thin hypervisor on the fly [17]. To defeat those anti-forensic techniques, Stüttgen et al. [6] propose an anti-forensic resilient method to acquire physical memory by eliminating its dependence on the operating system routines and data structures. Schatze [53] 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 [9], [10]. 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 [47], [48]. However, our preliminary work [57] showed that it is possible to conceal memory used by an attacker, even when the acquisition software is trusted due to malicious manipulation of the physical address layout.

Besides software-based memory acquisition, several hardware-based memory acquisition methods have been developed recently [28], [58], [59], [8] to use 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 [29] shows that it is possible to present a different view of the physical memory to the peripherals by reprogramming the northbridge. Therefore, in-memory data acquired by DMAs could be compromised, as well [9], [6].

A special type of memory acquisition technique relies on the unique remanence property of physical DRAM [12], [11]. Despite the popular belief that volatile content in DRAM are gone once the computer resets or powers off, Halderman et al. [12] demonstrate a Cold Boot attack that can reliably recover the content 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, it is also an effective method for memory forensics.

TABLE III
Attacks by Misusing Architectural Features

System	Feature misused	Cold Boot
SMM Rootkit [60]	System management mode	✓
Bluepill Attack [61]	HW virtualization	✓
Shadow Walker [15]	TLB Incoherence	✓
TPM Cloaking [62]	TPM - Intel TXT	✓
HIveS [57]	Physical address layout	✓
Cloaker [63]	IV reloation & TLB locking	✓
CacheKit [64]	Cache, physical address layout	x
Malclaveware	Intel SGX	x

Even though we are the first to study the impact of architectural feature misuse in memory forensics, there has been a line of research that examines how hardware resources can be misused for malware [64], [60], [63], [61] to impede system introspection. Table III shows the architectural features that each of the previous works utilized and whether hardware-based memory acquisition, such as Cold Boot, can be used to detect the malware. As shown in the table, the most effective method to detect these attacks that misused architectural features [60], [63], [61], [15], [62] is memory forensics [62] using Cold Boot, like memory acquisition procedures [12], [11]. Building on our preliminary work [57] that focused on the potential threat of physical address layout manipulation on memory forensics, we presented Malclaveware, which takes a different approach against memory forensics. Instead of manipulating the system to conceal the presence of memory, we apply hardware encryption to encrypt the memory to sabotage the memory acquisition process. Using the processor-bounded memory encryption in SGX, Malclaveware effectively denies memory forensics access to plaintext data.

IX. Conclusions

In this paper, we present a new class of anti-memory-forensic techniques that misuse features in modern computer architecture to prevent digital forensics. The prototypes of two attacks are built to demonstrate the feasibility. The first prototype, HIveS, manipulates the physical address space to conceal data. In-memory data is shadowed behind the I/O address. Besides the I/O Shadowing technique that prevents forensic memory acquisition via processor, we also develop two new techniques: blackbox write and TLB Camouflage. Blackbox write enables the attacker exclusive write access to HIveS memory, while TLB Camouflage grants a single malicious core exclusive read and write access. The second prototype, Malclaveware, exploits secure execution technology to prevent the operating system from uncovering malicious activity. The cryptographic component in Ransomware is inserted in the SGX enclave in Malclaveware. Due to the protection of SGX technology, the operating system is unable to access the encryption key stored in the malware enclave. To mitigate the newly discovered threats, we provide discussion on the possible countermeasures in an effort to fuel development of future secure systems.

Acknowledgment

This work was supported in part by US National Science Foundation under grants CNS-1446478. Dr. Kun Suns work is supported by U.S. Office of Naval Research under grants N00014-16-1-3214 and N00014-16-1-3216.

References

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

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

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

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

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

[6] J. Stüttgen and M. Cohen, "Anti-forensic resilient memory acquisition," *Digital Investigation*, vol. 10, pp. S105–S115, 2013.

[7] E. Libster and J. D. Kornblum, "A proposal for an integrated memory acquisition mechanism," *SIGOPS Oper. Syst. Rev.*, vol. 42, pp. 14–20, Apr. 2008.

[8] B. D. Carrier and J. Grand, "A hardware-based memory acquisition procedure for digital investigations," *Digital Investigation*, vol. 1, no. 1, pp. 50 – 60, 2004.

[9] 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, (New York, NY, USA), pp. 79–88, ACM, 2012.

[10] 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*, pp. 1–5, IEEE, 2011.

[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*, pp. 307–316, ACM, 2010.

[12] 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*, vol. 52, no. 5, pp. 91–98, 2009.

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

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

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

[16] 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.

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

[18] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution.," in *HASP@ ISCA*, p. 10, 2013.

[19] "ARM Security Technology, Building a Secure System using TrustZone Technology," apr 2009.

[20] A. Kharraz, S. Arshad, C. Mulliner, W. Robertson, and E. Kirda, "Unveil: A large-scale, automated approach to detecting ransomware," in *USENIX Security 16*, pp. 757–772, 2016.

[21] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in *Proceedings of the 15th ACM conference on Computer and communications security*, pp. 51–62, ACM, 2008.

[22] T. Fischer, "Private and public key cryptography and ransomware," 2014.

[23] E. Kirda, "Cutting the gordian knot: A look under the hood of ransomware attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings*, vol. 9148, p. 3, Springer, 2015.

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

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

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

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

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

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

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

[31] 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*, pp. 73–82, ACM, 2011.

[32] 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.

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

[34] 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.

[35] "Memory Forensic Challenges under Misused Architectural Features." http://memoryforensic.weebly.com/.

[36] N. Zhang, K. Sun, W. Lou, and Y. T. Hou, "Case: Cache-assisted secure execution on arm processors," in *Security and Privacy (SP), 2016 IEEE Symposium on*, pp. 72–90, IEEE, 2016.

[37] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," OSDI'14, 2014.

[38] A. Young and M. Yung, "Cryptovirology: Extortion-based security threats and countermeasures," in *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pp. 129–140, IEEE, 1996.

[39] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[40] H. Taylor, "Ransomware spiked 6,000% in 2016 and most victims paid the hackers, IBM finds." https://goo.gl/8afou8.

[41] V. Kotov and M. Rajpal, "Understanding crypto-ransomware," *Bromium whitepaper*, 2014.

[42] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual, System Programming Guide*, Sep 2016.

[43] "Dumpit." https://zeltser.com/memory-acquisition-with-dumpit-for-dfir-2/.

[44] D. G. Lunde, "What is the average size of an office document?." https://blogs.technet.microsoft.com/dangl/2012/10/18/what-is-the-average-size-of-an-office-document/.

[45] J. D. Kornblum and C. ManTech, "Exploiting the rootkit paradox with windows memory analysis," *International Journal of Digital Evidence*, vol. 5, no. 1, pp. 1–5, 2006.

[46] "Intel Chipset 4 GB System Memory Support ," Feb 2005.

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

[48] 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*, p. 13, ACM, 2011.

[49] D. Kaplan, J. Powell, and T. Woller, *AMD MEMORY ENCRYPTION*, Apr 2016.

[50] 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.

[51] S. Vömel and F. C. Freiling, "A survey of main memory acquisition and analysis techniques for the windows operating system," *Digital Investigation*, vol. 8, no. 1, pp. 3–22, 2011.

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

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

[54] J. Sylve, "Lime-linux memory extractor," *ShmooCon12*, 2012.

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

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

[57] N. Zhang, K. Sun, W. Lou, T. Hou, and J. Sushil, "Now you see me: Hide and seek in physical address space," in *ASIACCS*, ACM, 2015.

[58] J. Wang, A. Stavrou, and A. K. Ghosh, "Hypercheck: A hardware-assisted integrity monitor," in *RAID*, pp. 158–177, 2010.

[59] BBN(Raytheon), "Fred: Forensic ram extraction device." http://www.digitalintelligence.com/products/fred/.

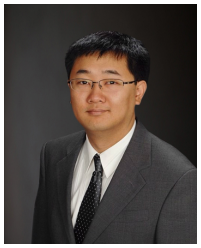
[60] S. Embleton, S. Sparks, and C. C. Zou, "SMM rootkit: a new breed of OS independent malware," *Security and Communication Networks*, vol. 6, no. 12, pp. 1590–1605, 2013.

[61] J. Rutkowska, "Subverting VistaTM kernel for fun and profit," *Black Hat Briefings*, 2006.

[62] A. M. Dunn, O. S. Hofmann, B. Waters, and E. Witchel, "Cloaking malware with the trusted platform module.," in *USENIX Security Symposium*, 2011.

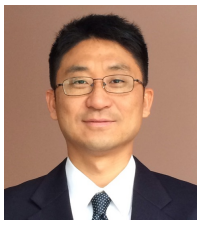
[63] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell, "Cloaker: Hardware supported rootkit concealment," in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pp. 296–310, IEEE, 2008.

[64] N. Zhang, H. Sun, K. Sun, W. Lou, and Y. T. Hou, "Cachekit: Evading memory introspection using cache incoherence," in *EuroS&P*, IEEE, 2016.



Ning Zhang received his Ph.D. degree in Department of Computer Science from Virginia Polytechnic Institute and State University, in 2016. He is a cyber engineer at Raytheon company from 2007 till now. He is also an Adjunct Assistant Professor with Virginia Tech since 2016. His research focuses on systems and network security, including trusted computing, binary analysis and cyber-physical systems.

Ruide Zhang received his B.S. degree from the Department of Electrical Engineering, Beijing University of Posts and Telecommunications (BUPT), Beijing, China, in 2014. Since the fall of 2014, he has been pursuing the Ph.D. degree in the Computer Science Department at Virginia Tech, under the supervision of Prof. Wenjing Lou. His research interests lie in system security and wireless security.



Kun Sun received his Ph.D. from Department of Computer Science at North Carolina State University. Dr. Kun Sun is an Associate Professor in the Department of Information Sciences and Technology at George Mason University. He is also the director of Sun Security Laboratory. He has more than 10 years working experience in both industry and academia. His research focuses on systems and network security. The main thrusts of his research include moving target defense, trustworthy computing, password management, and mobile security. He published over 50 technical papers on security conferences and journals including IEEE S&P, ACM CCS, NDSS, IEEE DSN, ESORICS, ACSAC, IEEE TDSC, and IEEE TIFS.

Wenjing Lou received the Ph.D. degree in electrical and computer engineering from the University of Florida, in 2003. From 2003 to 2011, she was a Faculty Member with the Worcester Polytechnic Institute. She has been a Professor with Virginia Tech since 2011. Since 2014, she has been serving as a Program Director at the U.S. National Science Foundation (NSF), where she is involved in the Networking Technology and Systems program and the Secure and Trustworthy Cyberspace program. Her current research interests focus on privacy protection techniques in networked information systems and cross-layer security enhancement in wireless networks, by exploiting intrinsic wireless networking and communication properties.

Y. Thomas Hou received the Ph.D. degree from the New York University Tandon School of Engineering. He is currently the Bradley Distinguished Professor of Electrical and Computer Engineering with Virginia Tech, Blacksburg, VA. His current research focuses on developing innovative solutions to complex cross-layer problems in wireless and mobile networks. He has authored two graduate textbooks, Applied Optimization Methods for Wireless Networks (Cambridge University Press, 2014) and Cognitive Radio Communications and Networks: Principles and Practices (Academic Press/Elsevier, 2009). He is a member of the IEEE Communications Society Board of Governors and the Steering Committee Chair of the IEEE INFOCOM Conference.

Sushil Jajodia is University Professor, BDM International Professor, and the director of Center for Secure Information Systems at George Mason University. His research interests include information security and privacy. He has authored or coauthored six books, 41 edited books, and more than 425 papers. He is an IEEE fellow and has an h-index of 96.