TRESOR-HUNT: Attacking CPU-Bound Encryption

Erik-Oliver Blass
Northeastern University
blass@ccs.neu.edu

William Robertson
Northeastern University
wkr@ccs.neu.edu

ABSTRACT

Hard disk encryption is known to be vulnerable to a number of attacks that aim to directly extract cryptographic key material from system memory. Several approaches to preventing this class of attacks have been proposed, including Tresor [18] and LoopAmnesia [25]. The common goal of these systems is to confine the encryption key and encryption process itself to the CPU, such that sensitive key material is never released into system memory where it could be accessed by a DMA attack.

In this work, we demonstrate that these systems are nevertheless vulnerable to such DMA attacks. Our attack, which we call Tresor-Hunt, relies on the insight that DMA-capable adversaries are not restricted to simply reading physical memory, but can write arbitrary values to memory as well. Tresor-Hunt leverages this insight to inject a ring 0 attack payload that extracts disk encryption keys from the CPU into the target system's memory, from which it can be retrieved using a normal DMA transfer.

Our implementation of this attack demonstrates that it can be constructed in a reliable and *OS-independent* manner that is applicable to any CPU-bound encryption technique, IA32-based system, and DMA-capable peripheral bus. Furthermore, it does not crash the target system or otherwise significantly compromise its integrity. Our evaluation supports the OS-independent nature of the attack, as well as its feasibility in real-world scenarios. Finally, we discuss several countermeasures that might be adopted to mitigate this attack and render CPU-bound encryption systems viable.

1. INTRODUCTION

Hard disk encryption is an increasingly popular set of techniques for preserving the confidentiality of persistent data. In such approaches, each block is encrypted before writing it to disk, and blocks are decrypted after reading them from disk. Disk encryption is completely transparent from the user's perspective, and virtually all major operating systems support this security mechanism—e.g., BitLocker [16] for Microsoft Windows, FileVault [1] for Mac OS X, and dm-crypt [4] for Linux. A similar functionality is provided by prominent third-party applications such as TrueCrypt [28] and PGP [26].

However, it has been shown previously that an adversary with physical access to a machine can circumvent disk encryption and access sensitive data. For instance, by attaching a malicious device to a running target machine, the adversary can perform a so-called DMA attack [5, 2, 13, 3, 20, 12].[1] Certain peripheral hardware busses—such as FireWire, Thunderbolt, or ExpressCard—give direct, unfettered access to a system's main memory. As such, the malicious device simply reads out the encryption key used to encrypt the hard disk using a DMA transfer. Knowing the secret key, the adversary can decrypt the hard disk and access data. Such DMA attacks are not only academic, but have already been seen in the real world [22]. As of today, there are password-recovery toolkits available that render DMA attacks accessible to everyone [21]. In conclusion, DMA attacks pose a major threat to any unattended machine.

To mitigate the problem of DMA attacks, recent work [18, 25] has suggested moving the encryption key from RAM to the CPU, which is inaccessible via DMA. Additionally, encryption is solely performed using CPU registers, thwarting any attempts to reveal sensitive key material using DMA transfers. We refer to cryptographic systems with this property as *CPU-bound*.

In this paper, we show that CPU-bound hard disk encryption is insecure as presented in prior work. We present a *novel*, *realistic*, and *concrete* attack, where an adversary with access to a DMA-capable hardware bus can access encryption keys of a CPU-bound encryption system. The critical observation underlying our work is that attackers are not only able to read from a system's memory, but are also able to write arbitrary code and data into memory. Using this capability, we demonstrate that an attacker can expose a CPU-bound encryption key by injecting a small piece of code into the operating system kernel. This code transfers the encryption key from the CPU into RAM, from which it can be accessed using a standard DMA transfer.

To summarize, our contributions are the following:

- We demonstrate that by leveraging the write capability

[1]DMA, or Direct Memory Access, refers to the capability of peripheral system hardware to transfer data to or from main memory without the involvement of the CPU. This feature is intended to improve system performance, but comes at the expense of centralized memory access enforcement.

of DMA transfers, an attacker can bypass the protections afforded by CPU-bound disk encryption systems such as TRESOR [18] and LOOPAMNESIA [25].

- We experimentally validate the feasibility of the attack by implementing it against TRESOR in an *OS-independent* way that only depends upon details of the IA32(e) architecture. The resulting attack is capable of circumventing TRESOR in a matter of seconds without crashing or otherwise significantly compromising the integrity of the target system.

  We note that while we concretely focus on TRESOR and FireWire-based DMA, our attack is directly applicable to all CPU-bound disk encryption systems, all IA32(e)-based systems, and all peripheral busses with DMA capabilities such as Thunderbolt or ExpressCard.

- We discuss potential mitigation strategies for our attack that improve the security of CPU-bound disk encryption.

The remainder of this paper is structured as follows. In Section 2, we present relevant background information on CPU-bound disk encryption, and on TRESOR in particular. We present our specific attack on TRESOR in Section 3, and evaluate its efficacy in Section 4. We discuss the feasibility of our attack and of CPU-bound encryption in Section 5. Finally, we present related work and briefly conclude in Sections 6 and 7.

2. BACKGROUND

CPU-bound disk encryption systems are intended to render normal disk encryption systems resilient to evil maid attacks [23], cold boot attacks [7], and other scenarios where attackers might gain physical access to a running target system. In this section, we describe the threat model, assumptions, and implementation of TRESOR [18], a recent, representative example of CPU-bound disk encryption. We stress, however, that while we ground our discussion in the example of TRESOR, the main ideas directly carry over to similar proposals such as LOOPAMNESIA. Where appropriate, we highlight details specific to TRESOR.

2.1 Threat Model

TRESOR adopts a strong adversarial model, in which attackers can execute arbitrary code in ring 3 on IA32 systems. Therefore, attackers can execute code with root privileges on UNIX-like systems, or as ADMINISTRATOR or SYSTEM on Windows-based systems. However, attackers should not be able to execute code in ring 0, and TRESOR takes several steps to prevent this from occurring—even when such execution would normally be allowed—that are described below. Similarly, attackers should not be able to access kernel memory.

TRESOR assumes that it is run directly on the hardware, and not as a virtualized guest that could be introspected by a privileged host operating system.

TRESOR does not guarantee that legitimate ring 0 code will not leak information about the encryption key from the CPU, e.g., by copying values from debug registers or executing program paths that are dependent on debug register values.

As TRESOR is intended to prevent most realistic physical attacks against hard disk encryption, attackers can examine the contents of memory using a number of techniques, including hardware bus inspection, cold boot attacks, and DMA transfers initiated by peripheral devices. It is assumed that directly inspecting CPU state is difficult, for instance by attaching JTAG debuggers or specialized hardware probes.

2.2 Implementation

TRESOR, and other CPU-bound disk encryption systems, maintains the confidentiality of the encryption key by confining it and the encryption process to the CPU. In the following, we discuss the specifics of how TRESOR accomplishes this.

Key Storage.

Because CPU registers are not directly accessible by DMA transfers, TRESOR uses them to store the encryption key. However, even though CPU registers cannot be directly accessed through DMA, they could be indirectly read without taking extra precautions. Software running in user space can be preempted, and user space registers will automatically be persisted to memory as part of a context switch. Using user space registers would therefore be prone against a well-timed DMA attack against the process control blocks of the kernel. This automatically disqualifies pure user space-based approaches.

Therefore, TRESOR uses the debug registers `dr0`, `dr1`, `dr2`, and `dr3`, giving a total of $4 \cdot 32 = 128$ bits of storage for 32-bit machines, or $4 \cdot 64 = 256$ bits of storage for 64-bit machines. This is enough to accommodate AES-128 or AES-256, respectively. In addition, the IA32 hardware specification ensures that the debug registers are inaccessible outside of ring 0 code.

An ASCII passphrase is initially entered at system boot time. The kernel then derives an AES key from the user's input using SHA-256 applied a number of times, and stores the resulting key into the debug registers. The kernel immediately wipes the memory used for the initial passphrase and AES key derivation. On multi-core machines, the same key is written in the debug registers of all cores.

Other CPU-bound encryption schemes use different registers for the same purpose. For instance, LOOPAMNESIA uses Intel Machine Specific Registers (MSRs), and the previous version of TRESOR uses the Intel Streaming SIMD Extension (SSE) registers [17].

AES Implementation.

TRESOR relies upon the Intel AES-NI instruction set to perform AES encryption. Recent Intel CPUs—e.g., Core i5 and Core i7—implement this instruction set to support AES encryption and decryption in hardware. The `aesenc(x,y)` CPU instruction performs one round of AES encryption (`SubBytes`, `ShiftRows`, `MixColumns`, and `AddRoundKey`) in one CPU cycle. Here, `(x,y)` are drawn from the set of sixteen SSE registers `xmm0` to `xmm15`. The first register `x` contains the current round key, and the second register `y` contains the current state of the AES encryption. The output of `aesenc`, the AES state, is written into the destination register `y`. Therewith, TRESOR encrypts one plaintext block completely outside of RAM.

To generate the different round keys completely outside of RAM, TRESOR uses the `aeskeygenassist` instruction. The

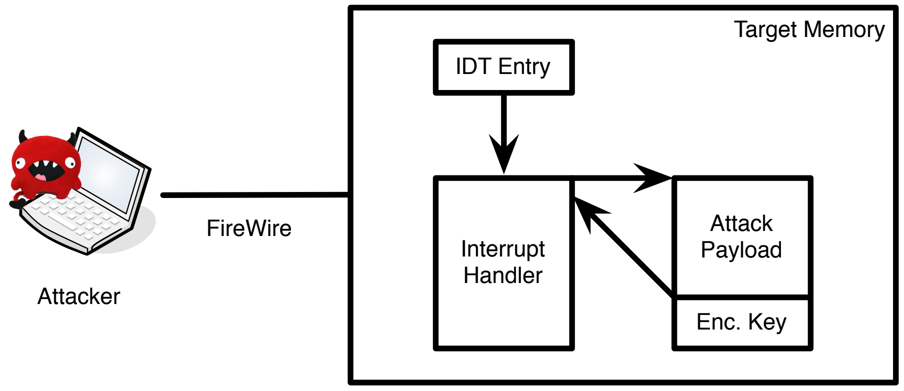

Figure 1: Overview of TRESOR-Hunt. The attacker overwrites physical memory in the target system to hook an interrupt handler. Then, an attack payload is executed that extracts the disk encryption key from the CPU into memory. The attacker can then initiate a DMA transfer to obtain the key.

AES key is copied from the debug registers into the first SSE register `xmm0`. Using `aeskeygenassist`, the 13 subsequent round keys can be derived from `xmm0` and are written into `xmm1` to `xmm13`.

To enable access to SSE registers during encryption, TRESOR defines the complete 14 round AES encryption within an atomic block in the kernel. This block cannot be interrupted, and SSE registers are saved while entering this block and restored before leaving. Being atomic, neither preemption by scheduling nor interrupts can disturb AES encryption and potentially leak the content of the registers to RAM.

Finally, TRESOR exports its AES interface to the Linux Crypto-API. This makes it available to standard Linux disk encryption software such as dm-crypt.

Kernel Integrity.

As dictated by the threat model, TRESOR assumes that attackers cannot run arbitrary code in ring 0. To enforce this, TRESOR takes additional steps to prevent the otherwise-allowed execution of attacker-controlled code in a kernel context.

1) The `ptrace` system call is modified to disallow access to the debug registers. Normally, `ptrace` allows user space programs such as debuggers to read and write debug registers, e.g., to set hardware breakpoints.

2) Kernel memory introspection from user space, enabled by special devices such as `/dev/kmem`, is disallowed. Otherwise, an attacker with sufficient ring 3 privileges could trivially read and write into kernel space, modifying, e.g., the atomic block within the kernel to read out debug registers.

3) Similarly, support for loadable kernel modules (LKMs) is removed. Otherwise, an attacker can insert a kernel module that reads out the debug registers.

These assumptions can be generalized to a *kernel integrity* property. Essentially, any CPU-bound encryption system depends on the integrity of kernel code, which is difficult to guarantee in general. We will now demonstrate that, even if the above assumptions hold and kernel integrity can be guaranteed against any local attacker with full user space privileges, DMA-based attacks are powerful enough to circumvent all of the above protection mechanisms.

3. A CPU-BOUND ENCRYPTION ATTACK

While CPU-bound disk encryption systems like TRESOR increase the difficulty for an adversary to compromise disk encryption keys, they are not impervious to attack under the threat model discussed in Section 2. In this section, we present an end-to-end attack against TRESOR, which we call TRESOR-Hunt. Note that while we focus on TRESOR as a case study, the techniques we employ generalize to any combination of CPU-bound disk encryption scheme, IA32-based OS, and DMA-capable hardware bus. In particular, while incorporating OS-specific knowledge would greatly simplify the attack—and would clearly be desirable should this information be confirmed for a particular target—we show that OS-specific details are not required, and that the attack is equally applicable to Windows, Mac OS X, and Linux without *a priori* knowledge of which OS is deployed on the target.

3.1 Attack Overview

Figure 1 presents a graphical overview of the attack. We outline the individual steps here.

1) First, the attacker gains physical access to the running IA32-based target system and attaches a device to a DMA-capable bus, e.g., FireWire.

2) The device initiates a DMA transfer to recover the contents of physical memory.

3) The device analyzes the physical memory dump, and identifies the kernel paging structures and interrupt descriptor table (IDT).

4) Using information derived from these structures, the device prepares an attack payload.

5) The device performs a DMA transfer to inject the attack payload into the memory of the target system.

6) The payload executes within a kernel context—i.e., ring 0—and copies the Tresor encryption key from the CPU into a predetermined location in physical memory.

7) The device initiates a final DMA transfer to obtain the disk encryption key from memory.

Note that each of the above steps makes no assumptions on the target system aside from the presence of a DMA-capable bus and an IA32-derived architecture. In particular, virtually any modern OS for this architecture will make use of hardware paging for features such as process isolation and virtual memory, as well as interrupt handling for features such as device service requests and scheduling.

In the following, we discuss details of the individual steps of the attack.

3.2 Accessing System Memory

In our instantiation of the attack, we use a FireWire bus to perform DMA transfers to and from system memory. FireWire, or IEEE 1394, is a hardware specification for high-speed peer-to-peer device communication. Most importantly for our purposes, since FireWire is DMA-capable, it allows for unimpeded access to main memory.

Our prototype extends Inception [13], an existing tool for gaining privileged access to machines with an accessible FireWire bus. The tool's main purpose is to attack common (non-CPU-bound) disk encryption systems by using physical memory overwrites to grant root access to an attacker. The particular overwrites to perform are guided by a set of hardcoded signatures for a set of popular Linux-based operating systems. However, we merely build upon Inception's ability to read and write to physical memory. The remainder of our attack is built as a novel extension to the tool, and bears only superficial resemblance to its approach.

A notable limitation of FireWire is that it is limited to accessing the first 4 GB of memory. This limitation, however, did not impede the ability of our attack to successfully compromise Tresor keys. This is due to the fact that the memory of interest resides at a relatively low physical address for the systems we attacked. Additionally, given the nature of the data structures we target, there is little reason to expect that this condition will not hold in other environments.

3.3 Hijacking Kernel Control Flow

Given a physical memory dump of the system obtained by Inception, the objective is then to analyze the dump in order to successfully execute code in ring 0—i.e., within the kernel context. An obvious approach here would be to use standard kernel hooking techniques; that is, to overwrite a known kernel function pointer to redirect control flow to code that we inject into physical memory. This approach, however, requires OS-specific knowledge to identify these function pointers and, therefore, is not suitable when the target OS is not known *a priori*.

Instead, the approach we adopt is to rely only upon data structures present in the target system's memory that are required by the IA32 architecture specification. In particular, our attack uses a combination of the kernel paging structures and interrupt descriptor table (IDT) to identify suitable locations to a) inject an attack payload to execute in ring 0, and b) to redirect control flow to that payload. In the following discussion, we refer to the IA32e—i.e., 64 bit—representation of these structures. Since it is relatively straightforward to heuristically infer the machine word size for a target architecture by examining a physical memory dump, we assert that this is without loss of generality.

IA32e Interrupt Descriptor Table (IDT).

The IDT is an IA32-specific structure that allows software to register handlers for system events such as interrupts and exceptions. The IDT itself is a contiguous array of descriptors that map an interrupt vector to an interrupt service routine (ISR). Each vector serves as an index into the array. Examples of standard interrupt vectors for the IA32 architecture include the breakpoint exception `#BP` (3), the general protection exception `#GP` (13), and page fault exception `#PF` (14). In addition, system-specific handlers can be mapped for vectors 32-255.

The CPU refers to the IDT's location in memory through the `IDTR` register, the value of which is loaded from memory and stored to memory using the `lidt` and `sidt` instructions, respectively. The `IDTR` specifies both the size of the IDT (minus 1), and the base of the table in memory.

The IDT serves as an ideal, OS-agnostic mechanism for locating code on IA32-based systems with the following properties: a) the code executes in ring 0, and b) it is potentially executed very often. While directly overwriting an IDT in memory is inadvisable due to the certain consequence that the machine will reset itself, each IDT entry does point to a function that can be hooked.

In particular, the approach we adopt is to select a system-specific interrupt vector, resolve its handler, and extract the first 16 bytes. We save these for later restoration. Then, we inject a jump to the location of our attack payload. This payload will be responsible for implementing our attack, as well as removing the hook and restoring the original initial ISR instruction sequence.

Identifying the IDT.

While the attack is, in principle, straightforward, there are several difficulties that arise. The first has to do with identifying the location of the IDT in memory. Recall that the standard means for accomplishing this is to execute the `sidt` instruction, which stores the value of the `IDTR` register into a specific location in memory. For instance, the following assembly routine would place the contents of `IDTR` in the memory location specified by the first argument in `rdi`.

```
; extern void __sidt(void *idtr)

bits 64
section .text

global __sidt

__sidt:
    sidt [rdi]
    ret
```

Unfortunately, this is a chicken-and-egg problem: to execute our attack, we need to locate the IDT, but before we gain control of the system, we cannot execute any instructions. Similarly, our DMA-based access to the system does not allow us to directly examine the state of the CPU.

Instead, we utilize a technique for heuristically identifying an IDT by scanning a physical memory dump. Our heuristics rely both on architectural constraints on IDTs as well as the fact that there exists much regularity in the values for each field of each IDT entry. While these constraints are certainly not foolproof, our experiments demonstrate that our heuristics are effective in practice.

In particular, our scan searches for a block of contiguous memory that satisfies the following properties.

1) The memory region is page-aligned to a 4 KB boundary.

2) The high-order bits of each entry's ISR are self-similar.

3) The type of each entry is one of the three permissible values—i.e., 5, 6, or 7.

We found that this approach was sufficient to reliably identify IDT locations in our experiments. For more details, please refer to Section 4.

Identifying IA32e Paging Structures.

Locating the IDT in memory is not the only challenge, unfortunately. A second obstacle arises from the fact that interrupt vector handlers are specified as linear addresses, while the attacker's device is restricted to a physical view of memory. That is, the target kernel is executing in protected mode, with a set of paging structures mapping linear addresses to physical addresses. In contrast, the attacker's device addresses physical memory. Therefore, without the ability to associate linear addresses with physical addresses, the device is unable to perform (at least) three critical tasks: a) locate the physical address of a particular ISR given its virtual addresses in the IDT, b) hook the ISR with the virtual address of the attack payload, and c) construct the attack payload such that it refers to the virtual address of the original ISR in order to remove the hook.

Therefore, in addition to locating the IDT, it is necessary to parse the kernel memory map as specified by the kernel paging structures. However, we face a similar situation to the case of the IDT and `sidt` instruction. The root of the paging structures is usually contained in the `cr3` register, but accessing that value requires execution of multiple `mov` instructions. Of course, the attacker is unable to do so at this point.

Accordingly, we adopt a similar approach of heuristically identifying the kernel's paging structures. Here, we rely on a combination of architectural constraints and OS-independent characteristics of kernel memory maps. In particular, we scan physical memory for a hierarchical paging structure that exhibits the following properties.

1) The paging structure tree is rooted at a valid page map level 4 (PML4) table.

2) PML4 entries, if present, point to valid page directory pointer tables (PDPTs).

3) PDPT entries, if present, point to valid page directories (PDs).

4) PD entries, if present, point to valid page tables (PTs).

5) Each node in the tree is page-aligned to a 4 KB boundary.

6) Reserved bits in each entry at each node of the tree are properly set to zero.

7) The ratio of pages with only ring 0 access to those with ring 3 access is above a fixed threshold.

8) The number of mapped pages is above a fixed threshold.

The first two properties above are architectural constraints; the second two are universal characteristics of kernel memory maps (most pages should only be accessible to the kernel, and a minimum number of pages should be mapped into physical memory). As in case of the IDT identification heuristics, the above was sufficient to uniquely identify the kernel paging structures in our experiments.

3.4 Preparing an Attack Payload

After resolving the location of the IDT and kernel paging structures, the next step is to construct the actual attack payload—i.e., the code that will be injected into the system to execute with ring 0 privileges. Given the address of the ISR to hook and the first 16 bytes of that ISR, this is quite simply accomplished by patching a compiled attack template such as shown in Figure 2. In particular, `INT_ADDR_MARK` is a special byte sequence that marks the location of the attack payload to patch with the address of the original ISR. Similarly, `INST_BUF_MARK` is a special byte sequence that marks the location to save the original initial instruction sequence for the target ISR.

3.5 Executing the Attack Payload

Execution of the attack payload requires two additional steps: a) injecting the payload into a writable and executable page in kernel memory, and b) patching the target ISR to redirect control flow to the location of the injected payload.

Our attack prototype accomplishes the first task by traversing the kernel memory map to discover a suitable physical page. The second task is completed by selecting a target ISR and replacing its initial instructions with a `jmp ATTACK_PAYLOAD_ADDR` instruction. At that point, the next time that the selected interrupt is raised, control of the system will be redirected to the attack payload. It will copy the contents of the debug registers to a predetermined location in RAM, unhook the targeted ISR, and continue execution of that ISR. The attacker's device will then copy out the encryption key, defeating the CPU-bound property of the disk encryption system.

4. EVALUATION

In the following section, we report on an evaluation of TRESOR-Hunt. In particular, we focus on two key aspects of the attack: a) how effective are the heuristics used to identify IDTs and kernel paging structures, and b) how effective is the attack in practice.

4.1 Data Structure Identification

The goal of this experiment is to quantify the accuracy of the heuristics we use to identify IDTs and kernel paging structures. Accordingly, we extracted physical memory dumps for several IA32-based operating systems, including Linux 3.3.7, FreeBSD 9.0, and Mac OS X 10.7.3, and applied our heuristics. In each case, we successfully identified

```
global extract_key

; extract the disk encryption key
extract_key:
    ; copy debug registers
    mov rax, dr0
    mov [dbg_regs.dr0 wrt rip], rax
    mov rax, dr1
    mov [dbg_regs.dr1 wrt rip], rax
    mov rax, dr2
    mov [dbg_regs.dr2 wrt rip], rax
    mov rax, dr3
    mov [dbg_regs.dr3 wrt rip], rax

    ; restore original instructions
    mov rdx, INT_ADDR_MARK
    mov rax, [inst_buf.x0 wrt rip]
    mov [rdx], rax
    mov rax, [inst_buf.x1 wrt rip]
    mov [rdx+0x08], rax

    ; jump to original handler
    jmp [rdx]

; original instruction buffer
inst_buf:
    .x0 dq INST_BUF_MARK
    .x1 dq 0x00

; debug register dump
dbg_regs:
    .dr0 dq 0x00
    .dr1 dq 0x00
    .dr2 dq 0x00
    .dr3 dq 0x00

global attack_len
attack_len dq $-extract_key
```

Figure 2: Example TRESOR-Hunt attack payload template.

OS	Handler MSBs
Linux 3.3.7	`0xffffffff81000000`
FreeBSD 9.0	`0xffffffff80b00000`
Mac OS X 10.7.3	`0xffffff80002d0000`

Table 1: Common bits for IDT entry handler addresses for a selection of IA32-based operating systems.

the IDT for each operating system when compared to the ground truth of executing the `sidt` instruction.

Table 1 displays the results of one aspect of the heuristics, namely the check for common bits of potential ISR handler addresses. For each OS tested, it is clear that many of the most-significant bits are shared, and they clearly correspond to kernel linear addresses, which tend to be located high in virtual memory.

A similar experiment was performed for the heuristics to resolve the location of the kernel paging structures. In this case, kernel drivers were written to directly access `cr3` as appropriate, since (as opposed to the `sidt` instruction) access to `cr3` is architecturally restricted to ring 0 code. In all cases, our heuristics were able to uniquely identify the correct location of the PML4 table.

As a result, while it is certainly possible to falsely identify an IDT or kernel paging structure, we conclude that our heuristics are effective in practice. We speculate that this might be partially attributable to the fact that we perform our scan as a linear sweep from low to high physical addresses. Since the data structures of interest tend to be initialized early in the boot process, they also tend to reside in low physical memory and are therefore discovered before false positives might be encountered.

4.2 Performance

In this experiment, we validate that TRESOR-Hunt is able to efficient and successfully extract disk encryption keys from the CPU. Since TRESOR is only available for Linux, we performed this experiment solely for that OS, although we speculate that the results of this experiment would not differ significantly in other environments.

We performed 10 trials of the attack against a Linux 3.0.31 kernel patched with TRESOR on machine with an Intel Core i7 CPU and 16 GB of memory. In each case, the time to perform the attack was dominated by the time to extract the initial physical memory dump, which was on the order of several minutes. In comparison, the time required to analyze the memory dump, construct the attack payload, inject the payload, and extract the key was negligible.

As a result, we conclude that the attack is highly feasible for the threat model described in Section 2, where an adversary can gain unobserved physical access to a DMA-capable bus on an unattended target machine.

5. DISCUSSION

We have demonstrated that CPU-bound encryption is insufficient to securely encrypt a hard disk in the face of DMA attacks. However, we will now discuss other techniques that allow efficient, realistic protection against such attacks.

Disabling DMA.

The most intuitive way to prevent DMA attacks is to simply disable DMA. Microsoft suggests this, for example to protect BitLocker full disk encryption [15]. Similarly, the “old” Linux FireWire protocol stack `ieee1394` (until 2010 [10]) offered a command line parameter `phys_dma=0` when loading the FireWire kernel module to completely disable DMA [8].

While disabling DMA is an effective protection of DMA attacks, this solution is not acceptable in many scenarios due to the implied performance penalty.

A more sophisticated way of disabling DMA for FireWire is used in Mac OS X Lion with FileVault 2 [1]: FireWire is only disabled is the host machine enters “sleep” mode, e.g., the lid of a laptop is closed. To re-enable DMA, the user password has to be entered on wake-up of the machine. This protection helps against DMA attacks, e.g., if a laptop is left somewhere “unattended”, but does not offer any defense in case of desktop or server machines that are running most of the time unattended.

Device Whitelisting.

The DMA attack we have demonstrated requires attaching a malicious device to a target machine's FireWire port. The malicious device initiates DMA transfers to carry out the attack. To do so, however, the target machine's FireWire controller has to generally enable DMA transfer for this device. By default, today's operating systems enable DMA for any device attached.

Consequently, one way to mitigate DMA attacks would be to *authenticate* any DMA device to the kernel before allowing to perform DMA transfer, i.e., enabling DMA at the FireWire controller. Only valid, benign FireWire devices trusted by the user will get permission to use DMA. However, the FireWire standard does not support device-host authentication. Enabling strong cryptographic authentication would require deep changes in the FireWire firmware on devices and host controllers.

Still, a weaker form of authentication, simple "identification" is possible even with standard FireWire devices and firmwares. Each FireWire device comes with a 64 bit Globally Unique IDentifier (GUID) readable by the kernel. As soon as a new FireWire device is attached to the FireWire bus, kernel and host controller reset and re-initialize the FireWire bus. As part of the initialization, the host controller can enable DMA for all attached devices.

For increased security, instead of simply enabling DMA by default, the kernel can check in advance the GUID of the attached FireWire device. If the GUID is part of a "whitelist" of allowed FireWire devices, the kernel will allow DMA for the host controller, otherwise the kernel will disable DMA. The only requirement for this is a kernel accessible whitelist where the user stores the GUID of his trusted FireWire devices. Using the FireWire stack of current Linux kernels, this whitelisting technique can be implemented in a straightforward manner by modifying the host controller initialization code in `init_ohci1394_dma.c` and `ohci.c`.

The above identification is clearly not authentication: if an attacker can spoof or guess the GUID of a trusted device, an impersonation attack is possible. Still, if the user protects access to his trusted FireWire devices, the GUID can provide sufficient protection against DMA attacks.

Hardware Disk Encryption.

With hardware disk encryption techniques, the actual encryption and decryption of data is performed by the hard disk itself and not by the host machine. Only during an initial phase at, e.g., boot time the host exchanges an encryption key with the hard disk. DMA attacks are impossible against hardware disk encryption. Many hard disk manufacturers offer hard disk with hardware disk encryption [24, 9, 27]. Although using hardware disk encryption successfully protects against DMA attacks, current solutions only fill a niche: they are (hardware) specific, only support the Windows operating system, often require support for Trusted Computing hardware on the host computer, and are expensive compared to software-based solutions.

IOMMU.

Similar to a traditional memory management unit (MMU), an IOMMU is a piece of hardware that controls access to physical memory for peripheral devices. Located between DMA-capable devices and the physical memory, it translates virtual addresses as used by DMA devices into physical memory addresses. Moreover, an IOMMU controls which device can read or write to which physical memory address.

Consequently, by using an IOMMU, the kernel could explicitly protect certain memory regions against reading or writing. The availability of an IOMMU would not only protect against DMA-based attacks, but also render CPU-bound encryption superfluous. As memory regions containing a cryptographic key can be protected using the IOMMU, there would be no need to perform encryption outside RAM anymore.

In practice, only recently Intel and AMD have introduced IOMMUs ("VT-d" and "AMD-Vi") for their latest chipsets. Currently IOMMUs are used in hypervisors to allow safe DMA transfers between attached devices and guest operating systems. As a matter of fact, today none of the popular operating systems supports IOMMU, and enabling support for IOMMUs requires significant changes to the operating system.

6. RELATED WORK

TreVisor [19] is a CPU-bound encryption system that isolates the encryption process in a hypervisor (BitVisor) on top of Linux. In addition, TreVisor uses Intel's VT-d IOMMU technology to restrict memory regions accessible by DMA to protect the integrity of the hypervisor. While effective, this approach has several drawbacks that render it impractical in the real world. First, the use of a hypervisor automatically disables virtualization software such VirtualBox and VMware, as well as rendering debug registers inaccessible. Second, although the authors use Intel's AES-NI [11] technology to compute individual AES rounds in hardware, performance decreases by up to 50% in this setup. Finally, Intel's recent VT-d IOMMU is not used by any major operating system today, and we do not conjecture its availability in the near future since using an IOMMU requires major changes to an operating system and its kernel.

Mac OS X's FileVault 2 [1] has been reported to disable FireWire and Thunderbolt DMA whenever the computer goes into "sleep mode" (standby), for example as soon as the user closes the lid of a laptop [14, 6]. If the user wakes up the computer from sleep, an "unlock password" has to be entered to resume normal operation and re-enable DMA. While in theory this is an effective countermeasure against any DMA attack, it is not applicable to a running system. Along the same lines, to protect BitLocker disk encryption again DMA attacks, Microsoft suggests disabling DMA transfer completely [15].

A related hardware attack against disk encryption systems is the cold boot attack [7]. A cold boot attack exploits the fact that contents of DRAM memory usually survives for some amount of time without power. An attacker can physically remove memory modules from a target machine, analyze them, and recover sensitive data. Contrary to DMA attacks, cold boot attacks can be mitigated with CPU-bound encryption.

7. CONCLUSIONS

CPU-bound encryption systems, such as LoopAmnesia and Tresor, attempt to prevent the disclosure of disk encryption keys from powerful adversaries that have full ring 3 privileges and physical access to the machine. In this paper, we present Tresor-Hunt, a *novel*, *realistic*, and *concrete* attack that bypasses the protection afforded by one such system.

Our attack relies on the insight that DMA-capable adversaries are not restricted to simply reading physical memory, but can write arbitrary values to memory as well. TRESOR-Hunt leverages this insight to inject a ring 0 attack payload that extracts disk encryption keys from the CPU into the target system's memory, from which it can be retrieved using a normal DMA transfer.

Our implementation of this attack demonstrates that it can be constructed in a reliable and *OS-independent* manner that is applicable to any CPU-bound encryption technique, IA32-based system, and DMA-capable peripheral bus. Furthermore, it does not crash the target system or otherwise significantly compromise its integrity. Our evaluation supports the OS-independent nature of the attack, as well as its feasibility in real-world scenarios. Finally, we discuss several countermeasures that might be adopted to mitigate this attack and render CPU-bound encryption systems viable.

8. REFERENCES

[1] Apple. FileVault 2. `http://support.apple.com/kb/HT4790`, 2012.

[2] B. Böck. Firewire-based Physical Security Attacks on Windows 7, EFS and BitLocker. `http://www.securityresearch.at/publications/windows7_firewire_physical_attacks.pdf`, 2009.

[3] A. Boileau. Hit by a Bus: Physical Access Attacks with Firewire. Ruxcon, `http://www.security-assessment.com/files/presentations/ab_firewire_rux2k6-final.pdf`, 2006.

[4] dm-crypt developers. dm-crypt: a device-mapper crypto target. `http://www.saout.de/misc/dm-crypt/`, 2012.

[5] M. Dornseif. Owned by an iPod: Firewire/1394 Issues. PacSec, `http://md.hudora.de/presentations/firewire/PacSec2004.pdf`, 2004.

[6] T. Garrison. Firewire Attacks Against Mac OS Lion FileVault 2 Encryption. `http://www.frameloss.org/`, 2011.

[7] J. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J.Appelbaum, and E. Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *Proceedings of USENIX Security Symposium*, pages 45–60, San Jose, CA USA, 2008. USENIX Association.

[8] U. Hermann. Physical memory attacks via Firewire/DMA – Part 1: Overview and Mitigation (Update) . `http://www.hermann-uwe.de/blog/physical-memory-attacks-via-firewire-dma-part-1-overview-and-mitigation`, 2008.

[9] Hitachi. Safeguarding Your Data with Hitachi Bulk Data Encryption. `http://www.hgst.com/tech/techlib.nsf/techdocs/74D8260832F2F75E862572D7004AE077`, 2008.

[10] IEEE 1394 FireWire WiKi. JuJu Migration. `https://ieee1394.wiki.kernel.org/index.php/Juju_Migration`, 2012.

[11] Intel. Intel Advanced Encryption Standard Instructions (AES-NI). `http://www.intel.com/`, 2010.

[12] N. P. Jr., T. Fraser, J. Molina, and W. Arbaugh. Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of USENIX Security Symposium*, pages 179–194, San Diego, USA, 2004.

[13] C. Maartmann-Moe. Inception. `http://www.breaknenter.org/projects/inception/`, 2011.

[14] C. Maartmann-Moe. Adventures with Daisy in Thunderbolt-DMA-land: Hacking Macs through the Thunderbolt interface. `http://www.breaknenter.org/`, 2012.

[15] Microsoft. Blocking the SBP-2 driver to reduce 1394 DMA threats to BitLocker. `http://support.microsoft.com/kb/2516445`, 2011.

[16] Microsoft. BitLocker Drive Encryption Overview. `http://technet.microsoft.com/en-us/library/cc732774.aspx`, 2012.

[17] T. Müller, A. Dewald, and F. Freiling. A Cold-Boot Resistant Implementation of AES. In *Proceedings of European Workshop on System Security*, Paris, FR, 2010.

[18] T. Müller, F. Freiling, and A. Dewald. TRESOR runs encryption securely outside RAM. In *Proceedings of USENIX Security Symposium*, San Francisco, CA USA, 2011. USENIX Association.

[19] T. Müller, B. Taubmann, and F. Freiling. TreVisor – OS-Independent Software-Based Full Disk Encryption Secure Against Main Memory Attacks. In *Proceedings of the Conference on Applied Cryptography and Network Security*, Singapore, 2012. To appear.

[20] P. Panholzer. Physical Security Attacks on Windows Vista. `https://www.sec-consult.com/files/Vista_Physical_Attacks.pdf`, 2008.

[21] Passware. Passware Kit Forensic 11.7. `http://www.lostpassword.com/kit-forensic.htm`, 2012.

[22] K. Poulsen. *Kingpin: How One Hacker Took Over the Billion-Dollar Cybercrime Underground.* Crown, 2011.

[23] B. Schneier. Evil Maid Attacks on Encrypted Hard Drives. `http://www.schneier.com/blog/archives/2009/10/evil_maid_attac.html`, October 2009.

[24] Seagate. Momentus 5400 FDE.2. `http://www.seagate.com/docs/pdf/marketing/po_momentus_5400_fde.pdf`, 2008.

[25] P. Simmons. Security Through Amnesia: A Software-Based Solution to the Cold Boot Attack on Disk Encryption. In *Proceedings of Annual Computer Security Applications Conference*, pages 73–82, Orlando, FL USA, 2011.

[26] Symantec. PGP Whole Disk Encryption. `http://www.symantec.com/whole-disk-encryption`, 2012.

[27] Toshiba. Toshiba Debuts Self-Encrypting Drive Technology. `http://www.prnewswire.com/news-releases/toshiba-debuts-self-encrypting-drive-technology-at-rsa-conference-2009-61822062.html`, 2009.

[28] TrueCrypt. Free Open-Source On-the-Fly Encryption. `http://www.truecrypt.org/`, 2012.