TRESOR Runs Encryption Securely Outside RAM

Tilo Müller Felix C. Freiling
Department of Computer Science
University of Erlangen

Andreas Dewald
Laboratory for Dependable Distributed Systems
University of Mannheim

Abstract

Current disk encryption techniques store necessary keys in RAM and are therefore susceptible to attacks that target volatile memory, such as Firewire and cold boot attacks. We present *TRESOR*, a Linux kernel patch that implements the AES encryption algorithm and its key management solely on the microprocessor. Instead of using RAM, TRESOR ensures that all encryption states as well as the secret key and any part of it are only stored in processor registers throughout the operational time of the system, thereby substantially increasing its security. Our solution takes advantage of Intel's new *AES-NI* instruction set and exploits the x86 debug registers in a non-standard way, namely as cryptographic key storage. TRESOR is compatible with all modern Linux distributions, and its performance is on a par with that of standard AES implementations.

1 Introduction

Disk encryption is an increasingly used method to protect confidential information in computer systems. It is particularly effective for mobile systems, such as laptops, since these are frequently lost or stolen [29]. With the growing availability of disk encryption systems, criminals and law enforcement alike have started to explore ways to circumvent this protection. Since current disk encryption techniques store keys in main memory, one approach to access the encrypted data is to acquire the key physically.

When physical access to the machine is given, keys can be extracted from main memory of running and suspended machines without privileged user access. Such attacks can broadly be classified into *DMA attacks* and *cold boot attacks*. DMA attacks use direct memory access through ports like Firewire [4, 3, 5], PCI [6, 28, 13], or PC Card [8, 17] to access RAM, while cold boot attacks [14] exploit the fact that memory contents fade away gradually over time. This allows to restore RAM contents after a short power-down by rebooting the machine with a boot device that directly reads out memory. Widespread disk encryption systems like *BitLocker* [1] (Windows), *FileVault* (MacOS), *dm-crypt* [30] (Linux), and TrueCrypt [36] (multi-platform) do not protect against such attacks. The current technological response, namely to keep the key in RAM but obfuscate its presence by using dispersal techniques, only partly counters the threat of memory attacks.

1.1 TRESOR

In this paper, we present the design and implementation of *TRESOR* (pronounced [trε:zoa]), a Linux kernel patch for the x86 architecture that implements AES in a way that is resistant to the attacks mentioned above and hence, allows for disk drive encryption with improved security.

TRESOR runs encryption securely outside RAM. Its underlying idea is to avoid RAM usage completely by both storing the secret key in CPU registers and running the AES algorithm *entirely on the microprocessor*. Towards this goal, TRESOR (mis)uses the debug registers as secure cryptographic key storage. While the principle of TRESOR is basically applicable to most x86 compatible CPUs, we focus on an implementation exploiting Intel's new *AES-NI* [31] extensions. The new AES instructions, currently available on all Core i7 processors and most Core i5, allow for accelerated AES using short and efficient code which implements most of the cryptographic primitive in hardware.

On systems running TRESOR, setting hardware breakpoints is no longer possible because the breakpoint registers are occupied with key data. However, as there are only four breakpoint registers, debuggers like GDB must deal with the possibility that all of them are busy anyway, for example, when more than four are set in parallel.

1.2 Related Work

TRESOR is the successor of *AESSE* [24], which was our prototype implementation but not well applicable in practice because it incurred two major problems. First, the Streaming SIMD Extension (SSE) [34] were used as key storage, breaking binary compatibility with many multimedia, math, and 3d applications. Second, AESSE was a pure software implementation and, due to the shortage of space inside CPU registers, the algorithm performed about six times more slowly than comparable standard implementations of AES.

During the work on TRESOR, Simmons independently developed a system called *Loop-Amnesia* [27] which pursues the same idea of holding the cryptographic key solely in CPU registers. In difference to TRESOR, Loop-Amnesia stores the key inside machine specific registers (MSRs) rather than in debug registers. Currently, it does not support the AES-NI instruction set and only a 128-bit version of AES. However, it allows to store multiple disk encryption keys securely inside RAM by scrambling them with a master key.

With *BitArmor* [23] there exists a commercial solution that claims to be resistant against cold boot attacks in particular. But as BitArmor does not generally avoid storing the secret key in RAM, it cannot protect from other attacks against main memory. Consequently, its cold boot resistance is not perfect, too (though quite good to resist the most common attacks of this kind).

Additionally, Pabel proposed a solution called *Frozen Cache* [21, 22] that exploits CPU caches rather than registers as secure key storage outside RAM. To our knowledge, this project is currently work in progress at an early development stage. Although it is a nice idea, a secure and efficient implementation is very difficult, if possible at all, because x86 caches can hardly be controlled by the system programmer.

Last, hardware solutions like *Full Disk Encryption Hard Drives (HDD-FDE)* [16, 35] use specialized crypto chips for encryption instead of the system CPU and RAM. Indeed, this is an effective method to defeat memory attacks, but it does not compete with TRESOR as a software solution. In our opinion, software solutions are not obsolete as they have several advantages: they are cheaper, highly configurable, vendor independent, and, last but not least, quickly employable on many existing machines.

1.3 Contributions

The central innovations of TRESOR are storing the secret key in CPU registers and utilizing AES-NI for encryption. AES-NI offers encryption (and decryption) primitives directly on the processor. This, however, does not mean that AES-NI based implementations of AES withstand memory attacks out-of-the-box. A typical AES-NI based implementation uses RAM to store the secret key and, for reasons of performance, the key schedule. The AES key schedule is required by the individual AES rounds and is generally computed only once and then stored inside RAM to speed up the encryption process. In contrast, TRESOR implements AES using AES-NI without leaking any key-related data to RAM.

The contributions of this paper are:

- We implement AES without storing any sensitive information in RAM.
- To this end, we present a kernel patch (TRESOR) that is binary compatible with all Linux distributions.
- We show that by using Intel's new AES-NI instructions, the performance of TRESOR is as fast (even slightly faster) than standard AES implementations that use RAM.
- By running TRESOR in a virtual machine and constantly monitoring its main memory, we demonstrate that TRESOR can withstand considerable efforts to compromise the encryption key. The only method to access the key with reasonable effort is compromising the system space, using a loadable kernel module, for example. Many other attacks, such as hardware attacks targeting processor registers, are defeated by TRESOR.

Overall, TRESOR is a disk encryption system that is both secure against main memory attacks and well applicable in practice.

1.4 Outline

The rest of this paper is structured as follows: In Section 2 we explain our design choices and give implementation details. We have evaluated TRESOR regarding three aspects: compatibility (Section 3), performance (Section 4) and, most importantly, its security (Section 5). We conclude in Section 6.

2 Design and Implementation

We now give an overview over design choices regarding the interface and the implementation of TRESOR.

2.1 Security Policy

The goal of TRESOR is to run AES entirely on the microprocessor without using main memory. This implies that neither the secret key, nor the key schedule, nor any

intermediate state should ever get into RAM. With this restrictive policy, any attacks against main memory become useless. But such an implementation cannot be achieved simply in user space for two reasons:

- First of all, user space is affected by scheduling, meaning that CPU registers are frequently swapped out to RAM due to context switching. That is the key and/or intermediate states of AES would regularly enter RAM – even though AES was implemented to run solely on the microprocessor.

- Second, the key storage registers should not be accessible from unprivileged tasks. Otherwise a local attacker could easily read out and overwrite the key.

Both problems can only be solved by implementing TRESOR in kernel space. To suppress context switching, we run AES atomically. The atomic section is entered just before an input block is encrypted and left again right afterwards. Therefore, we can use arbitrary CPU registers to encrypt a block; we just have to reset them before leaving the atomic section. This guarantees that no sensitive data leaks into RAM by context switches. Between the encryption of two blocks, scheduling and context switches can take place as usual, so that the interactivity of multitasking environments is not affected.

To restrain userland from reading out the secret key, it is stored inside a CPU register set accessible only with ring 0 privileges. Any attempt to read or write the debug registers from other privilege levels generates a general-protection exception [18]. This defeats attackers who gained local user privileges and try to read out the key on software layer.

Due to the necessity to implement TRESOR in system space, we choose Linux for our solution because of its open source kernel. But in general our approach is portable to any x86 operating system.

2.2 Key management

AES uses a symmetric secret key for encryption and decryption. We now show how this key is managed by TRESOR.

Key storage

The first question regarding key management is: In which registers is the key stored within the processor? We now discuss several requirements these registers should meet.

Since the key registers are exclusively reserved over the entire uptime of the system, they will not be available for their designated use. Hence, to preserve binary compatibility with as many existing applications as possible, only seldom used registers are qualified to act as cryptographic key storage. Frequently used registers, like the general purpose registers (GPRs), are not an option since all computer programs need to read from and write to those registers. The loss of registers occupied by TRESOR should not break binary compatibility.

Another requirement is that the key registers should not be readable from user space as this would allow any unprivileged process to read or write the secret key. A key stored in GPRs, for example, could not be hidden from userland as the GPRs are an unprivileged resource, available to all processes.

Last but not least, the register set must be large enough to hold AES keys, i.e., 128 bits for AES-128, 192 bits for AES-192, and 256 bits for AES-256, respectively. A single register is too small to hold AES keys – on both 32- and 64-bit systems and thus, we have to use a set of registers.

Summarizing, a register set must satisfy four requirements to act as cryptographic key storage. The key registers must be:

1. seldom used by everyday applications,
2. well compensable in software,
3. a privileged resource, and
4. large enough to store at least 128, better 256 bits.

After considering all x86 registers we chose the debug registers, because they meet these requirements as we explain now.

The debug register set comprises four breakpoint registers `dr0` to `dr3`, one status register `dr6` and one control register `dr7`. Depending on the operating mode, `dr4` and `dr5` are reserved or just synonyms for `dr6` and `dr7`. Thus, the only registers which can be freely set to any value, are the four breakpoint registers `dr0` to `dr3`. On 32-bit systems these have $4 \times 32 = 128$ bits in total, just enough to store the secret key of AES-128. But on 64-bit systems these have $4 \times 64 = 256$ bits in total, enough to store any of the defined AES key lengths 128, 192, and 256 bits. [1]

The actual intention of breakpoint registers is to hold hardware breakpoints and watchpoints – features which are only used for debugging. And even for debugging their functionality can be compensated quite well in software, because software breakpoints can be used instead of hardware breakpoints. TRESOR reserves all four x86 breakpoint registers exclusively as key storage, i.e., TRESOR reduces the number of available breakpoint registers for other applications from at most 4 to always

[1] Although the principle of TRESOR is applicable to 32-bit systems, we recommend the usage of 64-bit CPUs to support full AES-256. Intel's Core-i processors are such 64-bit CPUs; these processors are also recommended because of their AES-NI support.

0. Since the breakpoint registers may be in use anyhow, by debuggers for example, unavailability of them can happen regularly as well, and thus, applications should be able to tolerate lack of them. This ensures binary compatibility with almost all user space programs.

Debug registers are a privileged resource of ring 0, meaning that none of the user space applications running in ring 3 can access debug registers directly. Any such access is done via system calls, namely via `ptrace`. As we show later, we patched the `ptrace` system call to return `-EBUSY` whenever a breakpoint register is requested, to let the user space know all of them are busy.

Key derivation

The key we store in debug registers is derived from a user password by computing a SHA-256 based message digest. To resist brute force attacks, we strengthen the key by applying 2000 iterations of the SHA-256 algorithm. The password consists of 8 to 53 printable characters[2] and is read from the user early during boot by an ASCII prompt, directly in kernel space. Only in kernel space we have full control over side effects like scheduling and context switching.

But how do we actually compute the key and get it into debug registers without using RAM? The answer is, that we *do* use RAM for this transaction – but only for a very short time frame during system startup. Although there is a predefined implementation of SHA-256 in the kernel, we implemented our own variant to ensure that all memory lines holding sensitive information, like parts of the key or password, are erased after usage. That is, during boot, password and key do enter RAM very briefly. But immediately afterwards, the key is copied into debug registers and all memory traces of it are overwritten. All this happens before any userland process comes to life.

Once the key has been entered and the machine is up and running, it cannot be changed from user space during runtime as it would be impossible to do so without polluting RAM. The password must only be re-read upon ACPI wakeup, because during suspend mode, the CPU is switched off and its context is copied into main memory. Naturally, we bar the debug registers from being copied into RAM, and hence, the key is lost during suspension and the password must be re-entered. Again, this happens early in the wakeup process, directly in kernel space before any user mode process is unfrozen.

On 64-bit systems, like Intel's Core-i series, we copy always 256 key bits into the debug registers and each of the AES variants (AES-128, AES-192, and AES-256) takes as many bits from the key storage as it needs. On multi-core processors, we copy the key bits into the debug registers of *all* CPUs. Otherwise we constantly had to ensure that encryption runs on the single CPU which holds the key. In terms of performance such migration steps are very costly and it is more efficient to duplicate the AES key onto all CPUs once. Furthermore, this allows us to run several TRESOR tasks in parallel.

[2] More characters do not add to security as it becomes easier to attack the key itself rather than the password because $95^{53} \gg 2^{256}$.

2.3 AES implementation

The challenge we faced was implementing the AES algorithm without using main memory. This implies we were not allowed to store runtime variables on the stack, heap or anywhere else in the data segment. Naturally, our implementation was written in assembly language, because neither the usage of debug registers as key storage nor the avoidance of the data segment is supported by any high-level language compiler.

Encryption algorithm

Storing only the secret key in CPU registers would already defeat common attacks on main memory, but following our security policy mentioned above, absolutely no intermediate state of AES and its key schedule should get into RAM. This aims to thwart future attacks and cryptanalysis. In other words, after a plaintext block is read from RAM, we write nothing but the scrambled output block back. No valuable information about the AES key or state is visible in RAM at any time.

From earlier experiments with AESSE [24], we were concerned about the performance penalty of encryption methods implemented without RAM. We therefore investigated the utilization of the AES-NI instruction set of new Intel processors [31]. AES-NI allows for hardware accelerated implementations of AES by providing the instructions `aesenc`, `aesenclast`, `aesdec`, and `aesdeclast`. Each of them performs an entire AES round with a single instruction, exclusively on the processor without involving RAM. Hence, they are compatible with our design.

Overall, utilizing AES-NI has several advantages:

- The code is clear and short.
- It runs without RAM usage.
- It is highly efficient.

The four AES instructions mentioned above work on two operands, the AES state and an AES round key; the round key is used to scramble the state. Instead of using memory locations for these operands, the AES instructions work on SSE registers. On 64-bit systems there are sixteen 128-bit SSE registers `xmm0` to `xmm15`. AES states and AES round keys exactly fit into one SSE register as they encompass 128 bits, too.

```
pxor        %xmm0, %xmm15
aesenc      %xmm1, %xmm15
aesenc      %xmm2, %xmm15
aesenc      %xmm3, %xmm15
aesenc      %xmm4, %xmm15
aesenc      %xmm5, %xmm15
aesenc      %xmm6, %xmm15
aesenc      %xmm7, %xmm15
aesenc      %xmm8, %xmm15
aesenc      %xmm9, %xmm15
aesenclast  %xmm10,%xmm15
```

Figure 1: AES-128 encryption using AES-NI

Figure 1 shows assembly code of the AES-128 encryption algorithm. Each line performs one of the ten encryption rounds. The second parameter (`xmm15`) represents the AES state and the first ten (`xmm1` to `xmm10`) the AES round keys. Writing a plaintext block into `xmm15`, the secret key into `xmm0`, and the round keys into their respective registers `xmm1` to `xmm10`, these ten lines of assembly code suffice to generate an AES encrypted output block in `xmm15`.

The decryption algorithm of AES-128 basically looks the same, just utilizing `aesdec` instead of `aesenc` and applying the round keys in reverse order. The implementations of AES-192 and AES-256 basically look the same, too, just performing twelve or 14 rounds instead of ten.

Round key generation

The difficulty to implement AES completely within the microprocessor stems from the structure of the AES algorithm. As shown in Figure 1, encryption works with round keys which we assumed to be stored in `xmm1` to `xmm10`. In conventional AES implementations, these round keys are calculated once and then stored inside RAM over the entire lifetime of the system. Only when needed, they are copied from RAM into SSE registers to be used in combination with AES-NI.

In TRESOR we cannot calculate the AES key schedule beforehand and store it inside RAM as this would obviously violate our security policy. On the other hand, we cannot store the entire key schedule in CPU registers either, because debug registers are too small to hold it and we do not want to occupy further registers for TRESOR. Consequently, we have to use an *on-the-fly* key schedule that recalculates the round keys each time when entering the atomic section. This means the round keys must be recalculated for each input block. Inside the atomic section we can safely store the round keys inside SSE registers as they are known not to be swapped out during this period.

Fortunately, the AES-NI extensions comprise an instruction for hardware accelerated round key generation, namely `aeskeygenassist`. Apparently, recomputing the entire key schedule again and again is a significant performance drawback compared to standard implementations of AES. By using this specialized instruction, key generation is relatively efficient, as we show later in this paper.

```
.macro  key_schedule last next rcon
  pxor              %xmm14,%xmm14
  movdqu            \last,\next
  shufps            $0x1f,\next,%xmm14
  pxor              %xmm14,\next
  shufps            $0x8c,\next,%xmm14
  pxor              %xmm14,\next
  aeskeygenassist   $\rcon,\last,%xmm14
  shufps            $0xff,%xmm14,%xmm14
  pxor              %xmm14,\next
.endm
```

Figure 2: AES-128 round key generation

Figure 2 lists assembly code to generate the next round key of AES-128. As each round key computation is based on slightly different parameters, we define a macro called `key_schedule` awaiting these parameters: `last` is an SSE register containing the previous round key, `next` is one that is free to store the next round key and `rcon` is an immediate byte, the round constant. Inside this macro `xmm14` is utilized as temporary helping register. To generate the ten round keys of AES-128, `key_schedule` has to be called ten times: `key_schedule %xmm0 %xmm1 0x1`, `key_schedule %xmm1 %xmm2 0x2`, and so on. Initially the secret key has to be copied from debug registers into `xmm0`.[3]

Using AES-NI it is more complex to generate round keys than to actually scramble or unscramble blocks, because with `aeskeygenassist` Intel provides an instruction to assist the programmer in key generation, but none to perform it autonomously. We conjecture that this is because key generation of the three AES variants AES-128, AES-192, and AES-256 differs slightly (for details see the original standard on AES [12]).

2.4 Kernel patch

Many operating system issues have to be solved when implementing encryption solely on processor registers.

[3] A full source code listing including all steps can be found in Appendix A.1.

As mentioned above, we have to patch the OS kernel for two reasons: First, we have to run parts of AES atomically in order to ensure that no intermediate state leaks into memory during context switches. Second, only in kernel space we can protect the debug registers from being overwritten or read out by unprivileged user space threads. We chose the most recent Linux kernel at that time (version 2.6.36) to implement these changes.

Key protection

For the security of TRESOR it is essential to protect the key storage against malicious user access. Even if no local attacker would read the debug registers on purpose, the risk remains that a debugger is started accidentally and pollutes the key storage. With a disk encryption system being active in parallel, such a situation would immediately lead to data corruption. Hence, the kernel must be patched in a way that it denies any attempt to access debug registers from user space.

```
int ptrace_set_debugreg
              (tsk_struct *t,int n,long v)
{
    thread_struct *thread = &(t->thread);
    int rc = 0;
    if (n == 4 || n == 5)
        return -EIO;
+   #ifdef CONFIG_CRYPTO_TRESOR
+   else if (n == 6 || n == 7)
+       return -EPERM;
+   else
+       return -EBUSY;
+   #endif
    if (n == 6) {
        thread->debugreg6 = v;
        goto ret_path;
    }
    if (n < HBP_NUM) {
        rc=ptrace_set_breakpoint_addr(t,n,v);
        if (rc) return rc;
    }
    if (n == 7) {
         rc=ptrace_write_dr7(t, v);
         if (!rc) thread->ptrace_dr7 = v;
    }
    ret_path: return rc;
}
```

Figure 3: Patched setter for debug registers

The debug registers can only be accessed from privilege level 0, i.e., from kernel space but not from user space. Only the `ptrace` system call allows user space applications like GDB to read from and write to them in order to debug a traced child. This makes it effectively possible to control access to debug registers centrally, i.e., on system call level. Running an unpatched Linux kernel, user space threads can access debug registers via `ptrace`; running a TRESOR patched kernel, we filter this access.

Figures 3 and 4 list patches we applied to functions of the `ptrace` implementation in `/arch/x86/kernel/ptrace.c`: `ptrace_set_debugreg` and `ptrace_get_debugreg`. The first patch returns `-EBUSY` whenever the user space attempts to write into breakpoint registers and `-EPERM` whenever it tries to write into debug control registers. The second patch returns just `0` for any read access to debug registers.

```
long ptrace_get_debugreg(tsk_struct *t, int n)
{
    thread_struct *thread = &(t->thread);
    unsigned long val = 0;
+   #ifndef CONFIG_CRYPTO_TRESOR
    if (n < HBP_NUM) {
        struct perf_event *bp;
        bp = thread->ptrace_bps[n];
        if (!bp) return 0;
        val = bp->hw.info.address;
    }
    else if (n == 6)
        val = thread->debugreg6;
    else if (n == 7)
       val = thread->ptrace_dr7;
+   #endif
    return val;
}
```

Figure 4: Patched getter for debug registers

Additionally, we patched elementary functions in `/arch/x86/include/asm/processor.h` to prevent kernel internals other than ours from accessing the debug registers: `native_set_debugreg` and `native_get_debugreg`. While the `ptrace` patches prevent user space threads from accessing debug registers, these patches prevent the kernel itself from accessing them, e.g., during context switching and ACPI suspend.

Atomicity

The operating system regularly performs context switches where processor contents are written out to main memory. When TRESOR is active, the CPU context encompasses sensitive information because our implementation uses SSE and general purpose registers to store round keys and intermediate states. These registers are not holding sensitive data persistently, like the debug registers do, but they hold them temporarily for the period of encrypting one block. Thus, although our AES implementation runs solely on registers and although we have patched the kernel to protect debug registers, sensitive data may still be written to RAM whenever the

scheduler decides to preempt AES in the middle of an encryption phase.

We solved this challenge by making the encryption of individual blocks atomic. Resetting the contents of SSE and general purpose registers before leaving the atomic section is an effective method to keep their contents away from context switching. Our atomicity does not only concern scheduling, but interrupt handling, too, because interrupt handlers, spontaneously called by the hardware, can write the CPU context into RAM as well.

Hence, to set up an atomic section we have to disable interrupts. On multi-core systems it is sufficient to disable interrupts locally, i.e., on the CPU the encryption task actually takes place on. Other CPUs can proceed with their tasks as a context switch on one CPU does not affect registers of another.

```
preempt_disable();
local_irq_save(*irq_flags);
// ... (encrypt block)
local_irq_restore(*irq_flags);
preempt_enable();
```

Figure 5: AES block encryption runs atomically

Figure 5 illustrates how to set up an atomic section in the Linux kernel that meets our needs. First `preempt_disable` is called to pause kernel preemption, meaning that running kernel code cannot be interrupted by scheduling anymore. Second, `local_irq_save` is called to save the local IRQ state and to disable interrupts locally. Next we are safe to encrypt an AES block as we are inside the atomic section. SSE and general purpose registers are only allowed to contain sensitive data within this section and must be reset before it is left. Once they are reset, local interrupts can be re-enabled (by `local_irq_restore`) and kernel preemption can be continued (by `preempt_enable`).

Crypto API

We integrated TRESOR into the Linux kernel Crypto-API, an interface for cryptographic ciphers, hash functions and compression algorithms. Besides a coherent design for cryptographic primitives, the Crypto-API provides us with several advantages:

- It allows ciphers to be dynamically (un)loaded as kernel modules. We left support for the standard AES module untouched and inserted TRESOR as a completely new cipher module. This enables end users to choose between TRESOR and standard AES, to run them in parallel, to compare their performance, etc.
- We do not have to implement cipher modes of operation, like ECB and CBC, ourselves since the Crypto-API handles them automatically. We have to provide the code to encrypt a single input block only and encrypting larger messages is done by the API.
- Existing software, most notably the disk encryption solution dm-crypt, is based on the Crypto-API and open to new cipher modules. That is, we do not have to patch dm-crypt to support TRESOR, but it is supported out-of-the-box. (Only third party encryption systems which do not rely on the Crypto-API, like TrueCrypt, cannot benefit from TRESOR without further ado.)

All in all, integrating TRESOR into the Crypto-API simplifies design. However, there is also a little drawback of the Crypto-API: It comes with its own key management which is too insecure for our security policy because it stores keys and key schedules inside RAM. To overcome this difficulty without changing the Crypto-API, we pass on a *dummy key* and look after the real key ourselves. Setting up an encryption system, the end user can pass on an arbitrary bit sequence as dummy key, but for apparent reasons it should *not* be equal to the real key.

3 Compatibility

We evaluated TRESOR regarding its compatibility with existing software (Section 3.1) and hardware (Section 3.2).

3.1 Software compatibility

Running on a 64-bit CPU, TRESOR is compatible with all three variants of AES, i.e., with AES-128, AES-192, and AES-256. To verify that no mistake slipped into the implementation we show its compatibility to standard AES: First of all we used official test vectors as defined in FIPS-197 [12]. TRESOR is integrated into the Crypto-API in such a way that a test manager proves its correctness based on these vectors each time the TRESOR module is loaded. Second, we scrambled a partition with TRESOR, unscrambled it with standard AES and vice versa. Along with structured data like text files and the filesystem itself, we created large random files and compared both plaintext versions, i.e., before scrambling with AES and after unscrambling with TRESOR. We compared these files and found them to be equal. This indicated the correctness of our implementation – not only in terms of single, predefined blocks (as test vectors do) but also regarding a great amount of random data.

Thanks to the Crypto-API, TRESOR is compatible with all kernel and user space applications relying on

```
> cryptsetup create tr /dev/sdb1 -c tresor
  Enter passphrase: ******
> mkfs.ext2 /dev/mapper/tr
> mount /dev/mapper/tr /media/tresor/
```

Figure 6: Create TRESOR partition using cryptsetup

this API. Among others these are the kernel-based disk encryption solution dm-crypt and all its user space frontends, e.g., `cryptsetup` and `cryptmount`. Figure 6 lists shell instructions to set up a TRESOR encrypted partition on the device `/dev/sdb1`. The password can be any arbitrary string as it is only used to create the dummy key; it has no effect on the actual encryption process. Consequently, a partition can be encrypted with the password "foobar" and decrypted with the password "magic" (as long as the TRESOR key stays the same).

TRESOR is expected to be compatible with all Linux distributions, meaning that all prepackaged user mode binaries are expected to run on top of the TRESOR kernel. For "normal" user mode applications like a shell, the desktop environment, your web browser, etc., this is pretty much self-evident – for a debugger it is not. But even for debuggers like GDB, binary compatibility is not broken because access to debug registers is handled via `ptrace` and we intercept this system call to inform the user space that all breakpoint registers are busy – a situation which could occur without TRESOR as well. To be more precise, we have to distinguish *breakpoints* and *watchpoints*:

1. Breakpoints: Calling `break`, GDB does not use hardware breakpoints by default. Instead it uses software breakpoints because their performance penalty is negligible and they can be defined in any quantity. Hardware breakpoints must explicitly be invoked by calling `hbreak` which fails on TRESOR with *"Couldn't write debug register: Device or resource busy."*

2. Watchpoints: Unlike breakpoints, watchpoints cannot be implemented well in software and run about a hundred times slower than normal execution [33]. Thus, GDB sets hardware watchpoints by default. Calling `watch` fails on TRESOR with *"Couldn't write debug register: Device or resource busy."* as well. To use software watchpoints instead, `set can-use-hw-watchpoints 0` must be run before.

Admittedly, not being able to use hardware breakpoints may be a reasonable drawback for malware analysts and software reverse engineers. Here we must limit the target audience to end-users and "normal" developers.

3.2 Hardware compatibility

TRESOR is only compatible with real hardware. Running TRESOR as guest inside a virtual machine is generally insecure as the guest's registers are stored in the host's main memory.

On hardware level, TRESOR's compatibility is further restricted to the x86 architecture. It is possible to run AES entirely on the microprocessor, even without an AES-NI instruction set (given that your CPU supports at least SSE2, which is the case for Pentium 4 and later CPUs). But in order to run full AES efficiently, processor compatibility is restricted to Intel's Core-i series at present. More clearly, we recommend the usage of 64-bit CPUs supporting the AES-NI instruction set. More and more processors will fall into this category in the future. Intel supports AES-NI since its mircoarchitecture code-named *Westmere*. AMD announced to support AES-NI starting with its *Bulldozer* core; processors based on this core are going to be released in 2011 [20]. All in all, many, if not most, upcoming x86 CPUs will support AES-NI.

4 Performance

We present performance measurements running a 64-bit Linux on an Intel Core i7-620M. The two performance aspects we evaluated are encryption speed and system reactivity. The latter may be affected because we halt the scheduler and run AES atomically.

4.1 Encryption benchmarks

We expected a performance penalty of TRESOR because of its recomputation of the key schedule for each input block – a substantial computing overhead compared to standard implementations which calculate the round keys only once. As shown in Section 2.3, round key generation is a heavy operation compared to the rest of the AES algorithm; and we are running through the entire key schedule for each 128-bit chunk, even when encrypting megabytes of data.

To measure the throughput of TRESOR in practice, we performed several disk encryption benchmarks. For disk benchmarking we mounted four partitions, one encrypted with TRESOR, one encrypted with generic AES, one encrypted with common AES-NI, and a plain one that was not encrypted at all. We mounted all of them with the `sync` option, meaning that I/O to the filesystem is done synchronously. We did this to avoid unrealistically high speed measurements that arise from disk caching. Caching would falsify our results because encryption does not take place before the data is actually going to disk.

Key	Generic AES	AES-NI	TRESOR	Plain
128	14.67	15.63	17.04	
192	14.89	15.40	16.47	47.32
256	15.04	15.92	15.77	

Table 1: `dd` throughput (in MB/s)

Table 1 lists average values over 24 `dd` runs, each writing a 400M file.[4] TRESOR-128 is faster than TRESOR-192 which again is faster than TRESOR-256, because with an increasing key size, more rounds are performed (10, 12 and 14, respectively) and thus, more round keys must be calculated on-the-fly.

The table also shows that TRESOR performs well in comparison to conventional AES variants: TRESOR is faster than the generic implementation of AES and even slightly faster than common AES-NI implementations. We were surprised by this ourselves and double-checked the results – once with the AES-NI module shipped with Linux 2.6.36 and additionally with a self-written variant. Currently, we have no good explanation for this effect. One possibility is that TRESOR gains advantage over other threads due to its atomic sections. Another possibility is that linear key generation on registers performs generally better than fetching round keys one after another from RAM.

	Generic AES	AES-NI	TRESOR	Plain
read	2.10	2.54	2.80	7.95
write	6.92	8.39	9.26	26.23

Table 2: Postmark benchmarks for AES-256 (in MB/s)

Besides measuring the throughput of `dd`, we utilized the disk drive benchmarking utility *Postmark* [19]. Postmark creates, reads, changes, and deletes many small files rather than just writing a single large file. As shown in Table 2, TRESOR has an advantage over generic AES and common AES-NI here as well.

Additionally, to measure the exact time needed to encrypt a single block, we wrote a kernel module named `tresor-test`. Inserting this module, diverse performance tests can be run for AES-128, AES-192, and AES-256. Findings from this module confirm our assumption that TRESOR runs faster than standard AES. For example, with TRESOR an AES-128 block is encrypted in about 440 nanoseconds (ns), while standard AES needs about 538 ns (but these values fluctuate heavily in practice, by more than 100%, and thus, we consider disk benchmarks as more reliable).

Overall, TRESOR involves no performance penalty and the impact of an on-the-fly round key generation is negligible.

4.2 Interactivity benchmarks

Performing heavy TRESOR operations in background, the OS reactivity to interactive events may be affected because TRESOR disables interrupts in order to run encryption atomically. In a desktop environment, for instance, mouse and keyboard events are raising interrupts which are now delayed until the end of a TRESOR operation. Furthermore, automatic scheduling is disabled for this period.

Hence, to preserve the reactivity of the system, we set the scope of atomicity to the smallest reasonable unit, namely to the encryption of a single 128-bit input block. Between processing two 128-bit blocks, interrupt processing and scheduling can take place as usual. Thereby interactivity is hardly affected because – as mentioned in the last section – it takes only 500 ns on average to encrypt a single block. Assuming it never takes more than 1000 ns, interrupt handling and scheduling can take place each microsecond if needed. But only delays greater than 150 milliseconds are perceptible by humans [10] and Linux scheduling slices are commonly between 50 and 200 milliseconds as well.

Crypto	Add. Load	AVG	MAX	STD
Generic AES	None	1.40	34.2	4.96
	X	6.73	43.0	11.30
	Compile	26.80	64.7	30.30
TRESOR	None	0.93	37.3	3.99
	X	6.65	44.3	11.30
	Compile	26.40	79.2	30.30
Plain	None	0.14	26.2	1.51
	X	0.40	23.8	2.34
	Compile	0.74	32.4	3.47

Table 3: Interbench (latencies in ms)

To prove that interactivity is indeed not affected in practice, we draw upon measurements from the benchmarking utility *Interbench* [9]. We used Interbench to simulate a video player trying to get the CPU 60 times per second, i.e., simulating 60 fps. Table 3 lists a selection of interactivity benchmarks running on an Intel Core i7-620M under different loads.[5] We disabled all but the first CPU core to get a more convincing test set-up. As shown by the table, latencies introduced by TRESOR do not differ much from those introduced by generic AES: Average latencies are slightly better for TRESOR,

[4] The underlying series of tests can be found in Appendix A.2.

[5] Full benchmarks are listed in Appendix A.3.

maximum latencies are slightly better for generic AES, and standard deviations are almost the same.

Overall, the atomic sections introduced with TRESOR are too short to have any measurable effect to the reactivity of the Linux kernel.

5 Security

Although it performs quite well, the ultimately decisive factor to employ TRESOR should not be its performance but its security qualities. Therefore we prove TRESOR's resistance against attacks on debug registers and, above all, attacks on main memory.

5.1 Memory attacks

We have implemented AES in a way that nothing but the scrambled output block is actively written into main memory. However, this alone does not guarantee the security of TRESOR, because sensitive data may be copied passively into RAM by side effects of the OS or hardware, such as interrupt handling, scheduling, swapping, ACPI suspend modes, etc. For example, we cannot directly exclude the possibility that there is a piece of kernel code reading from debug registers in assembly rather than calling our patched `native_get_debugreg` in C. To minimize this risk, we performed extensive tests observing the main memory of a TRESOR system at runtime.

The problem we faced was how to observe main memory reliably and efficiently. Just reading from `/proc/kcore` or `/dev/mem` in a running Linux system was not an option as the reading process itself invokes kernel code which may falsify the result. On the other hand, performing real attacks on main memory, like cold boot attacks, to read out what is physically left in RAM is very time consuming. Thus, we decided to run TRESOR as guest inside a virtual machine and to examine its "physical" memory from the host.

As VM we chose Qemu/KVM [11, 2] because it is lightweight, has a debug console and – last but not least – is compatible with TRESOR (many other VMs are not; VirtualBox [25], for instance, does not support AES-NI). The debug console of Qemu allows to read CPU registers and to take physical memory dumps in a comfortable way.

We started to browse the VM memory of an active disk encryption system with key recovery tools like *AESKeyFind* [15] and *Interrogate* [7]. As was expected, these tools successfully reconstructed the key of standard AES but not that of TRESOR. However, this alone is not a meaningful result because AES key recovery is commonly based on the AES key schedule (since the secret key itself has no structure; it is just a random bit sequence). As shown in Section 2.3, key schedules are not persistently stored under TRESOR and thus, key recovery must fail – it would even fail if the key actually leaks to RAM.

Unlike real attackers, we are aware of the secret key. We took advantage of this knowledge and searched for the key bit pattern. Overall, we could find a bit sequence matching the key of generic AES but none matching that of TRESOR. However, even these findings do not necessarily imply that the key is not present in RAM, because it could be stored discontinuously. This is not even unlikely, because inside the CPU it is stored discontinuously as well (in four breakpoint registers, 64-bit each). Context switching may store each register separately, for example.

Consequently, we had to perform more meaningful tests taking fractions of the key into account. And thus, we sought after the longest match of the key pattern and its reverse and any parts of those, in little and in big endian. We did not observe any case under TRESOR where the longest match exceeded three bytes. And matches of no more than three bytes can be explained purely by probabilities (as also attested by searching for random bit sequences instead of real key fractions).

This further raised our confidence that neither the secret key nor any part of it was in RAM – at the time we took the memory dump. This leads us to the immediately next problem: How can we ensure that main memory does not hold any part of the key at other times? In principle, this question is impossible to answer fully because of the intricacies of information leakage. In practice, it is hardly feasible to put the Linux kernel into all its possible states and to take a memory dump at the precise moment.

We tried to analyze at least the in our view most relevant states concerning swapping and suspend. Both are of special interest for TRESOR as they swap CPU registers into RAM or even further onto disk. We induced swapping by creating large data structures in RAM. Once Linux began to swap data onto disk we took a memory dump and a disk dump and analyzed both with the methods mentioned before. Parts of the secret key could neither be traced on disk nor in RAM. (Also for generic AES, we never found sensitive information on disk because kernel space memory is not swappable under Linux.)

To examine TRESOR's behavior for ACPI S3 (suspend to RAM) we performed tests in Qemu and additionally on real hardware because Qemu fails to wake up after S3. ACPI S4 (suspend to disk) on the other hand works just fine under Qemu. Our findings indicate that knowledge about the secret key is lost during both suspend modes, because again, neither in RAM nor on disk we could trace the key. As the CPU is switched off dur-

Active AES	Generic	TRESOR			None
Kernel state	normal	normal	swapping	suspend	normal
Key recovery (AESKeyFind)	yes	no	no	no	no
Dummy key matches	–	yes	yes*	yes	–
Real key matches	yes	no	no	no	no
Longest match of real key (bytes)	32	3	3	3	3
	*) found in RAM, not on disk				

Table 4: AES-256 key tracing, an overview

ing suspension, the key is irretrievably lost. We also verified this by looking into the CPU registers before, during, and after suspension. After suspension, the CPU context is restored completely except for the debug registers. (Therefore, TRESOR prompts the user to re-enter the password upon wakeup).

Table 4 summarizes our findings of tracing an AES-256 key in RAM and disk storage. Only using Linux' generic implementation of AES, the secret key can be recovered by AESKeyFind and Interrogate. Using TRESOR, or no disk encryption system at all, no key can be recovered. Indeed, we can trace the dummy key of TRESOR as it is stored in RAM by the Crypto-API, but the dummy key is of absolutely no importance. AESKeyFind cannot even recover the dummy key because no key schedule of it is ever computed or stored in RAM. The full 256-bit pattern of the real key can only be traced running generic AES. Under TRESOR, the longest sequence matching the real key has no more than three bytes in all kernel states we tested.

Concluding, we searched for the secret AES key with different methods in different situations and neither the entire key, nor any parts of it, could ever be traced in RAM. While this proves that we are successfully keeping the key away from RAM in general, we have no persuasive argument that the key *never* enters RAM. Admittedly, it is unlikely that a piece of code other than context switching swaps debug registers into RAM, but it cannot be ruled out. Anyhow, even for the hypothetical case where such a piece of code exists, we are quite confident that we can patch it. Hence, the feasibility of a system like TRESOR and its fundamental idea to store secret keys in debug registers is not at risk.

5.2 Processor attacks

Now that the key is stored inside the CPU and never enters RAM, attackers may target processor registers rather than RAM. Basically there are two ways to attack processor registers: on software and on hardware layer.

On software layer we distinguish attackers who could gain root access and attackers with an unprivileged access. Naturally, for attackers with standard user privileges there should be no way to read out the key. As debug registers are only accessible from kernel space and as the only way for standard users to execute kernel code are system calls, unprivileged attackers are successfully defeated by the `ptrace` patch. We verified this with a user space utility making `ptrace` calls to read out debug registers; as was expected, only `0` is returned to user space. Overwriting the key via `ptrace` is not possible either; here `-EBUSY` (dr0 to dr3) and `-EPERM` (dr6 and dr7) are returned.

For root the situation is different, because for root there are more ways to execute kernel code: via modules (LKMs) and via `/dev/kmem`. If Linux is compiled with LKM or KMEM support, root can insert arbitrary code into a running kernel and execute it with ring 0 privileges. To demonstrate this for LKMs, we have created a small malicious module reading out the debug registers and writing them into the kernel log file. A similar attack is possible by writing to `/dev/kmem`. Thus, if compiled with LKM or KMEM support, root can gain full access to the TRESOR key. On the other hand, if compiled without LKM and KMEM support, even root has no ability to access the secret key – an advantage over conventional disk encryption systems where root can always read and write the secret key from RAM. Running TRESOR without LKM and KMEM support, the key can be set once upon boot but never be retrieved or manipulated while the system is running.

Besides the software layer, the hardware layer is critical. With physical access to the machine, new possibilities open up for the attacker. First of all, for advanced electrical engineers it may be possible to read out registers of a running CPU with an oscilloscope, by measuring the electromagnetic field around the CPU or whatever else. But we are not aware of any successful attacks of this type.

Instead, we focus on a simpler scenario: it may be possible to reboot the machine with a malicious boot device reading out what is left in CPU registers (similar to cold boot attacks [14]). Performing such an attack, the interesting question is whether CPU registers are reset to zero upon reboot or keep their contents until they are used otherwise. Besides the BIOS version and CPU

reinitialization code, the answer may depend on whether the machine was rebooted by a software interrupt (e.g., by pressing *CTRL-ALT-DEL*) or by pressing a hardware reset button. While the former method keeps the CPU on power, the latter switches it off briefly.

To investigate the practical impact of such an attack, we developed a malicious boot device called *Cobra* (Cold Boot Register Attack). First tested on virtual machines, Cobra revealed that debug registers are reset on hardware reboots but not on software reboots. On software reboots, Cobra was able to restore debug registers; all tested virtual machines (Qemu, Bochs, VMware and VirtualBox) showed this behavior. If real hardware showed this behavior as well, the consequences would be fatal. It would be an ease to read out the secret key and hence, TRESOR would be practically useless. Fortunately, it turned out that all VMs have a little implementation flaw regarding this attack. On real hardware, debug registers are always reset to zero – also upon software reboots. We verified this by testing different machines with different processors and BIOS versions. Table 5 gives an overview of our findings.

	BIOS	Soft Reboot	Hard Reset
Athlon 64	AMI	-	-
Pentium 4	Phoenix	-	-
Pentium M	First	-	-
Celeron M	Phoenix	-	-
Core2 Duo	First	-	-
Core i5	Phoenix	-	-
Core i7	Lenovo	-	-
Qemu	Bochs	x	-
Bochs	Bochs	x	-
VMware	Phoenix	x	-
VirtualBox	N/A	x	-
	[x] = vulnerable	[-] = not vulnerable	

Table 5: Cobra (Cold Boot Register Attack)

Overall, we argue that TRESOR is secure against local, unprivileged attacks in any case. Beyond that, TRESOR is even secure against attackers who could gain root access, if the kernel is compiled without LKM and KMEM support. On hardware level, TRESOR withstands cold boot attacks against both main memory and CPU registers. This does only hold for real hardware, but running TRESOR inside a virtual machine is insecure anyhow as register contents of the guest are simulated in the host's main memory.

5.3 Side channel attacks

Last, we want to mention briefly that TRESOR is resistant to timing attacks [26]. This is not the achievement of ourselves but that of Intel, or, to be more precise, that of AES-NI. Intel states: *"Beyond improving performance, the AES instructions provide important security benefits. By running in data-independent time and not using tables, they help in eliminating the major timing and cache-based attacks that threaten table-based software implementations of AES."* [31] Based on this statement and the fact that there are no input dependent branches in the control flow of our code, we argue that TRESOR is resistant to side channel attacks, too.

6 Conclusions and Future Work

In the face of known attacks against main memory (above all, DMA and cold boot attacks) we consider RAM as too insecure to guarantee the confidentiality of secret disk encryption keys today. Thus we presented TRESOR, an approach to prevent main memory attacks against AES by implementing the encryption algorithm and its key management entirely on the microprocessor, solely using processor registers. We first explained important design choices of TRESOR and the key aspects of its implementation. We then discussed how we integrated it into the Linux kernel. Eventually we showed that it performs well in comparison to the generic version of AES and, most importantly, that it satisfies our security policy.

6.1 Conclusions

Our primary security goal was to prevent tracing of the secret key in volatile memory, effectively making attacks on main memory pointless. Despite considerable effort, we were not able to retrieve the key in RAM. Therefore, we are confident that TRESOR is a substantial improvement compared to conventional disk encryption systems. As we took perfectly intact memory images of a running TRESOR VM and knew the key beforehand, we had an advantage over real attackers trying to retrieve an unknown key. This strengthens our test results, because if we cannot retrieve the known key in an unscathed image, it is even more unlikely that an attacker can retrieve an unknown key in a partially damaged image.

Another security goal was, of course, not to introduce flaws which are not present in ordinary encryption systems. Therefore, we showed that TRESOR is safe against local attacks on the software layer as well as on the hardware layer. Interestingly, if the kernel is compiled without LKM and KMEM support, there is no (known) way to retrieve the secret key even though privileged root access is given – again, a substantial improvement compared to conventional disk encryption systems.

Besides evaluating security aspects, we collected performance benchmarks, revealing that TRESOR is

slightly faster than common versions of AES. Furthermore, we showed that the reactivity of Linux is not affected by the atomicity of encryption and decryption.

Summarizing, TRESOR runs encryption securely outside RAM and thereby it achieves a higher security than any disk encryption system we know – without losing performance or compatibility with existing applications. To conclude, it is possible to treat RAM as untrusted and to store secret keys in a safe place of today's x86 standard architecture.

6.2 Future Work

Currently, TRESOR allows only to store a single, static key, because the debug registers cannot hold a second one. Future versions of TRESOR may keep multiple disk encryption keys securely inside RAM by srambling them with a master key, like in Loop-Amnesia [27].

This idea may be extended to an even broader use case in the future: Further AES keys to be used in conjunction with IPSec or SSL, i.e., to be used in conjunction with the userland, could be encrypted with the TRESOR master key and stored securely inside RAM. Session keys could be set and removed dynamically in any quantity. Using such a session key to encrypt an input block, the user space application would have to make a special system call that: 1) invokes TRESOR to read and decrypt the desired key and 2) lets TRESOR use the recently decrypted key to encrypt the input block. Between these steps, the session key may not leave the processor, meaning both steps need to happen inside the same atomic section. As a downside, such a system would require user space support and would induce a performance penalty.

Another future task is to move the secret key into registers which are even less frequently used than the debug registers, e.g., into machine specific registers (MSRs). As a benefit, by using MSRs as cryptographic key storage, debuggers would be able to use hardware breakpoints and watchpoints again. However, the best way to get round this problem would be the introduction of a special key register into future versions of AES-NI by Intel or AMD.

Last, we want to investigate the possibility of implementing a TRESOR like system as third party application for Windows.

Acknowledgments

We would like to thank Hans-Georg Esser, Thorsten Holz, Ralf Hund, Stefan Vömel, and Carsten Willems for reading a prior version of this paper and giving us valuable suggestions for improving it.

Availability

TRESOR is free software published under the GNU GPL v2 [32]. Its source is available at `www1.informatik.uni-erlangen.de/tresor`.

References

[1] Windows BitLocker Drive Encryption Frequently Asked Questions. http://technet.microsoft.com/en-us/library/cc766200(WS.10).aspx, July 2009.

[2] KVM: Kernel Based Virtual Machine, 2010. http://www.linux-kvm.org/.

[3] Becher, M., Dornseif, M., and Klein, C. N. FireWire - All Your Memory Are Belong To Us. In *Proceedings of the Annual CanSecWest Applied Security Conference* (Vancouver, British Columbia, Canada, 2005), Laboratory for Dependable Distributed Systems, RWTH Aachen University.

[4] Böck, B. *Firewire-based Physical Security Attacks on Windows 7, EFS and BitLocker*. Secure Business Austria Research Lab, Aug. 2009.

[5] Boileau, A. Hit by a Bus: Physical Access Attacks with Firewire. In *Proceedings of Ruxcon '06* (Sydney, Australia, Sept. 2006). Tool (2008): http://storm.net.nz/static/files/winlockpwn.

[6] Carrier, B. D., and Grand, J. A Hardware-Based Memory Acquisition Procedure for Digital Investigations. *Digital Investigation 1*, 1 (Feb. 2004), 50–60.

[7] Carsten Maartmann-Moe. Interrogate. http://interrogate.sourceforge.net/, Aug. 2009.

[8] Christophe Devine and Guillaume Vissian. Compromission physique par le bus PCI. In *Proceedings of SSTIC '09* (June 2009), Thales Security Systems.

[9] Con Kolivas. Interbench: The Linux Interactivity Benchmark, 2006. http://users.on.net/~ckolivas/interbench/.

[10] Dabrowski, R., J., Munson, and V., E. Is 100 Milliseconds Too Fast? In *Proceedings of the CHI Conference on Human Factors in Computing Systems* (2001), vol. 2 of *Short talks: interaction techniques*, ACM, pp. 317–318.

[11] Fabrice Bellard. Qemu: Open Source Processor Emulator, 2010. http://qemu.org.

[12] FIPS. Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, National Institute for Standards and Technology, Nov. 2001.

[13] Guillaume Delugre. Reverse Engineering the Broadcom NetExtreme's firmware. In *Proceedings of HACK.LU '10* (Luxembourg, Nov. 2010), Sogeti ESEC Lab.

[14] Halderman, J. A., Schoen, S. D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J. A., Feldman, A. J., Appelbaum, J., and Felten, E. W. Lest We Remember: Cold Boot Attacks on Encryptions Keys. In *Proceedings of the 17th USENIX Security Symposium* (San Jose, CA, Aug. 2008), Princeton University, USENIX Association, pp. 45–60.

[15] Heninger, N., and Feldman, A. AESKeyFind. http://citp.princeton.edu/memory-content/src/, July 2008.

[16] Hitachi Global Storage Technologies. *Safeguarding Your Data with Hitachi Bulk Data Encryption*, July 2008. http://www.hitachigst.com/tech/techlib.nsf/techdocs/74D8260832F2F75E862572D7004AE077/$file/bulk_encryption_white_paper.pdf.

[17] HULTON, D. Cardbus Bus-Mastering: 0wning the Laptop. In *Proceedings of ShmooCon '06* (Washington DC, USA, Jan. 2006).

[18] INTEL CORPORATION. *Intel 64 and IA-32 Architectures Software Developer's Manual*, Volume 3A and 3B ed., Jan. 2011. System Programming Guide.

[19] JEFFREY KATCHER. PostMark: A New File System Benchmark. http://communities-staging.netapp.com/servlet/JiveServlet/download/2609-1551/Katcher97-postmark-netapp-tr3022.pd, 1997. Network Appliance, Inc.

[20] JOHN FRUEHE, DIRECTOR OF PRODUCT MARKETING. Following Instructions. http://blogs.amd.com/work/2010/11/22/following-instructions/, Nov. 2010. AMD.

[21] JÜRGEN PABEL. Frozen Cache. Blog: http://frozenchache.blogspot.com/, Jan. 2009.

[22] JÜRGEN PABEL. FrozenCache Mitigating cold-boot attacks for Full-Disk-Encryption software. In *27th Chaos Communication Congress* (Berlin, Germany, Dec. 2010), CCC. Video: http://blog.akkaya.de/jpabel/2010/12/31/After-the-FrozenCache-presentation.

[23] MCGREGOR, P., HOLLEBEEK, T., VOLYNKIN, A., AND WHITE, M. Braving the Cold: New Methods for Preventing Cold Boot Attacks on Encryption Keys. In *Black Hat Security Conference* (Las Vegas, USA, Aug. 2008), BitArmor Systems, Inc.

[24] MÜLLER, T., DEWALD, A., AND FREILING, F. AESSE: A Cold-Boot Resistant Implementation of AES. In *Proceedings of the Third European Workshop on System Security (EUROSEC)* (Paris, France, Apr. 2010), RWTH Aachen / Mannheim University, ACM, pp. 42–47.

[25] ORACLE CORPORATION. VirtualBox: x86 and AMD64 virtualization, 2011. http://www.virtualbox.org.

[26] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache Attacks and Countermeasures: the Case of AES. In *Topics in Cryptology - The Cryptographers' Track at the RSA Conference 2006* (San Jose, CA, USA, Nov. 2005), Weizmann Institute of Science, Springer, pp. 1–20.

[27] PATRICK SIMMONS. Security Through Amnesia: A Software-Based Solution to the Cold Boot Attack on Disk Encryption, Apr. 2011. University of Illinois at Urbana-Champaign.

[28] PETRONI, N. L., FRASER, T., MOLINA, J., AND ARBAUGH, W. A. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 13th USENIX Security Symposium* (Aug. 2004), University of Maryland, USENIX Association, pp. 179–194.

[29] PONEMON, L. *Airport Insecurity: The Case of Missing & Lost Laptops*. Ponemon Institute, June 2008. http://www.dell.com/downloads/global/services/dell_lost_laptop_study.pdf.

[30] SAOUT, C. dm-crypt: a device-mapper crypto target, 2006. http://www.saout.de/misc/dm-crypt/.

[31] SHAY GUERON. *Intel Advanced Encryption Standard (AES) Instruction Set White Paper*, Rev. 3.0 ed. Intel Corporation, Jan. 2010. Intel Mobility Group, Israel Development Center.

[32] STALLMAN, R., AND COHEN, J. GNU General Public License Version 2. http://www.gnu.org/licenses/gpl-2.0.html, June 1991. Free Software Foundation.

[33] STALLMAN, R., AND PESCH, R. H. Debugging with GDB: The GNU Source-Level Debugger. Tech. rep., The Free Software Foundation, 2010. Ninth Edition.

[34] THAKKAR, S., AND HUFF, T. The Internet Streaming SIMD Extensions. *IEEE Computer 32*, 12 (Apr. 1999), 26–34. Intel Corporation.

[35] TOSHIBA CORPORATION. *Toshiba Self-Encrypting Drive Technology at RSA Conference 2009*, Apr. 2009. http://sdd.toshiba.com/techdocs/WaveRSADemoFinalPressRelease4152009.pdf.

[36] TRUECRYPT FOUNDATION. TrueCrypt: Free Open-Source Disk Encryption Software for Windows, Mac OS and Linux. http://www.truecrypt.org/, 2010.

A Appendix

A.1 AES-128 Source Code

```
.set rstate, %xmm0  // AES state
.set rhelp,  %xmm1  // helping reg
.set rk0,    %xmm2  // round key 0
.set rk1,    %xmm3  // round key 1
.set rk2,    %xmm4  // round key 2
.set rk3,    %xmm5  // round key 3
.set rk4,    %xmm6  // round key 4
.set rk5,    %xmm7  // round key 5
.set rk6,    %xmm8  // round key 6
.set rk7,    %xmm9  // round key 7
.set rk8,    %xmm10 // round key 8
.set rk9,    %xmm11 // round key 9
.set rk10,   %xmm12 // round key 10

.macro  key_schedule r0 r1 rcon
  pxor            rhelp,rhelp
  movdqu          \r0,\r1
  shufps          $0x1f,\r1,rhelp
  pxor            rhelp,\r1
  shufps          $0x8c,\r1,rhelp
  pxor            rhelp,\r1
  aeskeygenassist $\rcon,\r0,rhelp
  shufps          $0xff,rhelp,rhelp
  pxor            rhelp,\r1
.endm

movq        %db0,%rax
movq        %rax,\r0
movq        %db1,%rax
movq        %rax,rhelp
shufps      $0x44,rhelp,\r0
pxor        rk0,rstate

key_schedule    rk0  rk1  0x1
key_schedule    rk1  rk2  0x2
key_schedule    rk2  rk3  0x4
key_schedule    rk3  rk4  0x8
key_schedule    rk4  rk5  0x10
key_schedule    rk5  rk6  0x20
key_schedule    rk6  rk7  0x40
key_schedule    rk7  rk8  0x80
key_schedule    rk8  rk9  0x1b
key_schedule    rk9  rk10 0x36
```

```
aesenc       rk1,rstate
aesenc       rk2,rstate
aesenc       rk3,rstate
aesenc       rk4,rstate
aesenc       rk5,rstate
aesenc       rk6,rstate
aesenc       rk7,rstate
aesenc       rk8,rstate
aesenc       rk9,rstate
aesenclast   rk10,rstate
```

A.2 dd Benchmarks for AES-192

```
--- Plain
410 MB copied, 6.30053 s, 65.0 MB/s
410 MB copied, 6.93762 s, 59.0 MB/s
410 MB copied, 10.0737 s, 40.7 MB/s
410 MB copied, 9.66396 s, 42.4 MB/s
410 MB copied, 8.20149 s, 49.9 MB/s
410 MB copied, 7.42723 s, 55.1 MB/s
410 MB copied, 7.16408 s, 57.2 MB/s
410 MB copied, 8.54818 s, 47.9 MB/s
410 MB copied, 9.91214 s, 41.3 MB/s
410 MB copied, 6.91875 s, 59.2 MB/s
410 MB copied, 10.3003 s, 39.8 MB/s
410 MB copied, 8.63959 s, 47.4 MB/s
410 MB copied, 10.3342 s, 39.6 MB/s
410 MB copied, 8.75659 s, 46.8 MB/s
410 MB copied, 8.12789 s, 50.4 MB/s
410 MB copied, 8.96658 s, 45.7 MB/s
410 MB copied, 7.90555 s, 51.8 MB/s
410 MB copied, 11.7209 s, 34.9 MB/s
410 MB copied, 8.31128 s, 49.3 MB/s
410 MB copied, 11.8716 s, 34.5 MB/s
410 MB copied, 9.90721 s, 41.3 MB/s
410 MB copied, 8.57025 s, 47.8 MB/s
410 MB copied, 9.34468 s, 43.8 MB/s
410 MB copied, 9.14162 s, 44.8 MB/s

--- TRESOR
410 MB copied, 23.9045 s, 17.1 MB/s
410 MB copied, 24.1203 s, 17.0 MB/s
410 MB copied, 26.3410 s, 15.5 MB/s
410 MB copied, 22.1279 s, 18.5 MB/s
410 MB copied, 24.9356 s, 16.4 MB/s
410 MB copied, 25.0071 s, 16.4 MB/s
410 MB copied, 23.5777 s, 17.4 MB/s
410 MB copied, 27.8006 s, 14.7 MB/s
410 MB copied, 24.8987 s, 16.5 MB/s
410 MB copied, 25.8959 s, 15.8 MB/s
410 MB copied, 25.7694 s, 15.9 MB/s
410 MB copied, 26.5178 s, 15.4 MB/s
410 MB copied, 25.3663 s, 16.1 MB/s
410 MB copied, 25.0566 s, 16.3 MB/s
410 MB copied, 25.4963 s, 16.1 MB/s
410 MB copied, 24.3083 s, 16.9 MB/s
410 MB copied, 23.9965 s, 17.1 MB/s
410 MB copied, 25.2287 s, 16.2 MB/s
410 MB copied, 24.5554 s, 16.7 MB/s
410 MB copied, 23.5884 s, 17.4 MB/s
410 MB copied, 24.2647 s, 16.9 MB/s
410 MB copied, 25.1395 s, 16.3 MB/s
410 MB copied, 25.0933 s, 16.3 MB/s
410 MB copied, 24.8469 s, 16.5 MB/s

--- Common AES-NI
410 MB copied, 26.2926 s, 15.6 MB/s
410 MB copied, 30.8604 s, 13.3 MB/s
410 MB copied, 26.1996 s, 15.6 MB/s
410 MB copied, 28.0075 s, 14.6 MB/s
410 MB copied, 23.5519 s, 17.4 MB/s
410 MB copied, 27.0643 s, 15.1 MB/s
410 MB copied, 30.2133 s, 13.6 MB/s
410 MB copied, 25.8206 s, 15.9 MB/s
410 MB copied, 22.3430 s, 18.3 MB/s
410 MB copied, 24.9686 s, 16.4 MB/s
410 MB copied, 25.1107 s, 16.3 MB/s
410 MB copied, 28.1641 s, 14.5 MB/s
410 MB copied, 23.6934 s, 17.3 MB/s
410 MB copied, 24.9228 s, 16.4 MB/s
410 MB copied, 25.4900 s, 16.1 MB/s
410 MB copied, 28.8577 s, 14.2 MB/s
410 MB copied, 31.2964 s, 13.1 MB/s
410 MB copied, 26.1635 s, 15.7 MB/s
410 MB copied, 29.8904 s, 13.7 MB/s
410 MB copied, 26.8250 s, 15.3 MB/s
410 MB copied, 26.8389 s, 15.3 MB/s
410 MB copied, 29.5131 s, 13.9 MB/s
410 MB copied, 24.2083 s, 16.9 MB/s
410 MB copied, 26.9091 s, 15.2 MB/s

--- Generic AES
410 MB copied, 28.6924 s, 14.3 MB/s
410 MB copied, 31.0992 s, 13.2 MB/s
410 MB copied, 31.5867 s, 13.0 MB/s
410 MB copied, 29.4021 s, 13.9 MB/s
410 MB copied, 28.9778 s, 14.1 MB/s
410 MB copied, 25.9368 s, 15.8 MB/s
410 MB copied, 27.3885 s, 15.0 MB/s
410 MB copied, 26.5581 s, 15.4 MB/s
410 MB copied, 29.6886 s, 13.8 MB/s
410 MB copied, 29.4497 s, 13.9 MB/s
410 MB copied, 27.6539 s, 14.8 MB/s
410 MB copied, 24.9992 s, 16.4 MB/s
410 MB copied, 26.6642 s, 15.4 MB/s
410 MB copied, 24.2744 s, 16.9 MB/s
410 MB copied, 26.0460 s, 15.7 MB/s
410 MB copied, 31.6460 s, 12.9 MB/s
410 MB copied, 26.6546 s, 15.4 MB/s
410 MB copied, 24.6940 s, 16.6 MB/s
410 MB copied, 27.0945 s, 15.1 MB/s
410 MB copied, 26.8366 s, 15.3 MB/s
410 MB copied, 29.6911 s, 13.8 MB/s
410 MB copied, 29.4740 s, 13.9 MB/s
410 MB copied, 25.5362 s, 16.0 MB/s
410 MB copied, 24.3959 s, 16.8 MB/s
```

A.3 Interbench for AES-256

```
--- Plain
Load    Latency +/- SD (ms)  Max Latency   % Desired CPU  % Deadlines Met
None      0.143 +/- 1.51        26.2             100           99.3
X         0.399 +/- 2.34        23.8             100           98.6
Burn       0.23 +/- 1.93          39            99.9           99.2
Write      0.16 +/- 0.852       20.7             100           99.9
Read      0.118 +/- 0.772       20.2             100           99.9
Compile   0.738 +/- 3.47        32.4             100             97
Memload   0.009 +/- 0.027      0.498             100            100

--- TRESOR
Load    Latency +/- SD (ms)  Max Latency   % Desired CPU  % Deadlines Met
None      0.926 +/- 3.99        37.3            99.9           94.9
X          6.65 +/- 11.3        44.3            99.6           64.8
Burn         28 +/- 31          66.6            48.6           2.45
Write       1.9 +/- 5.87        33.7            99.8           89.9
Read       2.41 +/- 6.32        27.8             100           86.2
Compile    26.4 +/- 30.3        79.2            50.1           4.73
Memload    9.24 +/- 13.2        48.7            98.3           48.7

--- Generic AES
Load    Latency +/- SD (ms)  Max Latency   % Desired CPU  % Deadlines Met
None        1.4 +/- 4.96        34.2            99.9           92.3
X          6.73 +/- 11.3          43            99.2           63.5
Burn       29.2 +/- 32.4        66.5            45.8           2.22
Write     0.657 +/- 3.39        36.5            99.8           96.9
Read       3.05 +/- 7.18        36.9            99.9           82.5
Compile    26.8 +/- 30.3        64.7            48.9           4.09
Memload    9.34 +/- 13.4        45.3            97.7           48.5
```