

AESSE: A Cold-boot Resistant Implementation of AES

Tilo Müller
Computer Science Department
RWTH Aachen University

Andreas Dewald Felix C. Freiling
Laboratory for Dependable Distributed Systems
University of Mannheim, Germany

ABSTRACT

Cold boot attacks exploit the fact that memory contents fade with time and that most of them can be retrieved after a short power-down (reboot). These attacks aim at retrieving encryption keys from memory to thwart disk drive encryption. We present a method to implement disk drive encryption that is resistant to cold boot attacks. More specifically, we implemented AES and integrated it into the Linux kernel in such a way that neither the secret key nor any parts of it leave the processor. To achieve this, we used the SSE (streaming SIMD extensions) available in modern Intel processors in a non-standard way. We show that the performance penalty is acceptable and present a brief security analysis of the system.

Categories and Subject Descriptors

D.4.6 [Software]: Operating Systems—*Security and Protection*; E.3 [Data]: Data Encryption

General Terms

Security

Keywords

Cold Boot Attacks, Advanced Encryption Standard, Streaming SIMD Extensions, Linux Kernel

1. INTRODUCTION

According to a recent survey [12], business travelers lose about 12,000 laptops per week in US airports. Only a minor fraction is stolen and only about one third are actually reclaimed. The rest are sold off, with the result of “potentially millions of files containing sensitive or confidential data” leaking to non-authorized parties. Hard disk encryption is usually advocated as the mechanism to prevent such losses, and fortunately, modern operating systems are increasingly supporting this feature. Examples are Linux’ dm-crypt [13] and Microsoft’s BitLocker [1].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EUROSEC '10 Paris, France

Copyright 2010 ACM 978-1-4503-0059-9/10/0004 ...\$10.00.

With the growing availability of disk encryption, criminals and law enforcement alike have started to explore ways to circumvent the protection. Since the decryption key is mostly stored in volatile memory, one approach is to acquire this key physically and to use it to access the encrypted data. Keys can be extracted from main memory of a running or suspended system, even though the attacker has no user access to the machine – physical access suffices [4].

1.1 Cold Boot Attacks

Memory contents were usually regarded as lost once the computer had been switched off. But in fact, memory contents can still be recovered for a few seconds after the system has been switched off or rebooted since they fade away gradually. Cooling the memory chips slows down the fading process even more, up to minutes. In an approach known as *cold boot* [4], the machine is reset and booted with a special device that directly stores a memory image. Cryptographic keys can be extracted from this image with special key finding algorithms [9]. This attack has been successful against many disk encryption techniques including dm-crypt.

Without special cryptographic support by the processor, only physical protection of the key can really prevent cold boot attacks. This can be performed by tamper proof hardware security modules [2, 11, 5]. This is however rather costly. Another approach is to use microprocessors that directly support cryptographic instructions. For example, modern (i.e., post-2010) Intel processors contain a special AES instruction set [6].

But since the vast majority of users does not have such new processors yet, a solution that relies on these features would not be usable for them. In contrast, our approach works for all x86 standard processors of the last ten years.

1.2 Contributions

This paper presents an approach to prevent cold boot attacks by implementing AES *entirely on the microprocessor*, i.e., using only processor registers. This means that neither the master key, nor any round keys, nor any intermediate states leave the processor at any time. In effect, this prevents any traces of the key to be found in volatile memory, making cold boot attacks useless. The idea is rather simple but until recently was considered infeasible to implement for two reasons:

1. Processor registers are a very scarce resource and the AES key alone can be up to 256 bits long (not speaking of intermediate results such as round keys).
2. The operating system regularly performs context switches

where processor contents are written to main memory and it was unclear how these effects can be contained.

We present the design and implementation of AESSE, an implementation of AES for Linux that is resistant to cold boot. We tackled the first challenge by making use of the Streaming SIMD Extensions (SSE) available in modern processors. SSE is mostly used for vector based parallel calculations typical for multimedia applications. SSE introduced 8 new 128 bit registers that we use to implement AES.

We integrated our approach into Linux 2.6.30 running on the Intel x86 platform and present first performance measurements. Roughly speaking, we solved the second challenge (context switch problem) by (1) making individual encryption operations atomic and (2) by persuading the kernel not to “notice” SSE, even though it was present. More extensive measurements and a rigorous security evaluation will follow in an extended version of this paper.

1.3 Paper Outline

We first give some background on AES and SSE in Section 2. We then give some details of the implementation in Section 3, followed by a brief evaluation in Section 4. We conclude in Section 5.

2. BACKGROUND

2.1 AES

AES, the Advanced Encryption Standard [10], is a symmetric block cipher that handles key lengths of 128, 192 and 256 bit. Depending on the key size 10, 12 or 14 cipher rounds are performed on each input block. In every round a different round key is used that is derived from the AES key. Round keys always have the same size (128 bit). The block length of AES is 128 bit as well.

Following the reference algorithm published by FIPS [10], input blocks are transformed into 4×4 matrices of one-byte elements. All AES operations modify this matrix that is also called the *state*. The first state is the input block and the last state is the output block. In each round the following operations are used sequentially to modify the state:

- **SubBytes:** Each byte of the state is substituted independently using a predefined substitution box [10], see Figure 1.
- **ShiftRows:** Rows are rotated by 0, 1, 2 and 3 bytes, respectively, to the left, see Figure 2.
- **MixColumns:** A complex permutation of columns (for details, see FIPS [10]). This operation is not performed in the final round.
- **AddRoundKey:** The round key is added to the state by a bitwise XOR operation.

Since most operations work on the state directly, they do not need much memory to be implemented. What does need a lot of memory are the round keys of AES. In common implementations the round keys are usually calculated only once when a new key is loaded, and kept in main memory. Round keys are derived from the secret AES key in an incremental manner (sketched in Figure 3 assuming 128 bit keylength). Roughly speaking, the first round key is the AES key; the next round key is based on the previous round

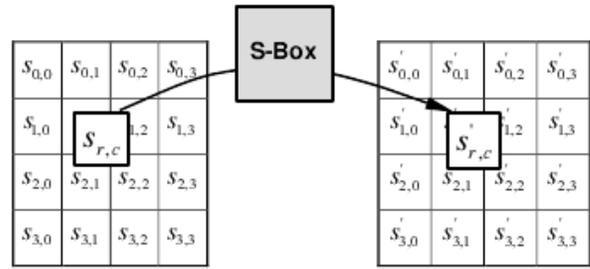


Figure 1: AES operation SubBytes [10].

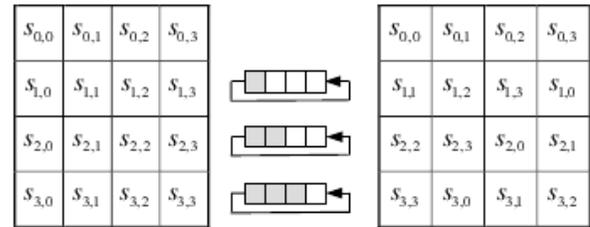


Figure 2: AES operation ShiftRows [10].

key using a bitwise XOR addition of selected words from the previous key schedule. In effect, the new round key can always be calculated from the old round key. To save memory, it is therefore only necessary to store one round key in the implementation of AES-128. (For AES-192 and AES-256 it is effectively necessary to store two round keys.)

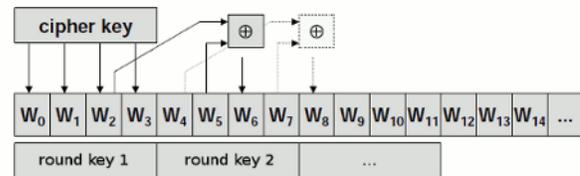


Figure 3: AES key expansion (simplified).

2.2 SSE

The Streaming SIMD Extensions (short SSE) are a SIMD (Single Instruction Multiple Data) instruction set that was introduced by Intel in 1999. SSE allows efficient vector-based calculations, e.g., instead of adding two vectors (x_1, y_1) and (x_2, y_2) sequentially by first calculating $x_1 + x_2$ and then $y_1 + y_2$ SSE can directly add the two vectors and effectively perform the two additions in parallel.

Today, SSE is heavily used in 3D applications but only very seldom in other desktop applications and even less in terminal applications. For example OpenGL 2.0 and later versions require SSE, but GCC normally does not introduce SSE itself when compiling, unless you pass on some of the high optimization flags like `-ffast-math`.

Technically, SSE introduces eight new registers, 128 bit each, called `xmm0` to `xmm7`. In total, this adds 1024 bits of register space to the CPU. These registers can be used, for example, as a vector of four 32 bit floating point numbers as shown in Figure 4. But they can also be used to store 128 bits of contiguous data, as shown below.

Sometimes, SSE is mistaken for MMX, the Multi Media Extensions published by Intel a few years earlier. MMX have a similar goal as SSE, however, MMX did not introduce new registers. Instead, MMX uses virtual names for registers that are in fact equal to the floating point unit (FPU) registers. Misusing MMX registers (to store a key for example) would have a severe effect on the performance of the system since the FPU is used by too many programs.

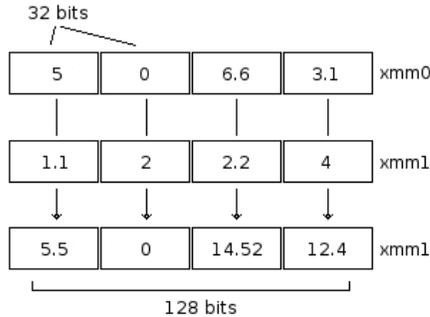


Figure 4: Multiplication of two SSE registers.

One particularly useful feature of SSE is that it can be completely disabled for programs running in user mode. This means that information stored in SSE registers is inaccessible even for programs running with root privileges in Linux for example. Enabling and disabling SSE is done by writing a flag of the control register CR4, which is only allowed for ring 0 (i.e., the kernel). Any attempt to execute an SSE instruction from userland will lead to a SIGILL (illegal instruction), leading to a crash of the process.

There are several versions of SSE. SSE4.2 is the latest. All these versions extend the instruction set [7, 8] but not the registers. For our implementation at least SSE2 is needed. SSE2 entered the market with the Pentium4 in the year 2000. All later Intel processors support SSE2 as well. AMD supports it since Athlon64 (Opteron) around 2003.

3. AESSE IMPLEMENTATION

We now give an overview of our implementation of AES using SSE.

3.1 Implementing AES on Registers

The challenge was to implement AES without writing to main memory. We therefore implemented it entirely on registers (AES-128, -192, and -256). We took care that not only the key is kept back from RAM, but also all intermediate states. The only data that is written back to RAM after the input block has been read is the output block. This restrictive policy ensures that no valuable information about the key is directly visible in RAM.

The implementation was performed directly in assembly language since an implication of our policy was that a runtime stack could not be used for passing parameters. Subroutines had to be programmed so that parameters were passed using registers. Moreover, we could not use the push instruction to make registers free for subroutine calls. In effect, all registers were global variables (as long as the caller needed a register, it could not be overwritten within a subroutine). This problem amplified the shortage of register space within the implementation, so we needed to revert

to some technical tricks. For example, we used individual registers to store several variables. In a 32 bit register, for example, you can store one 16 bit and two 8 bit variables. Furthermore we did ignore the usual interpretation of registers. For example, ESI and EDI are usually used as string indexes. Since we had no strings in our program we could use these registers for other purposes.

Figure 5 gives an overview how we allocated the additional space from the SSE registers. The SSE registers xmm0 to xmm6 had a rather static meaning, whereas xmm7 was used for multiple purposes. The general purpose registers were also used in multiple ways, except for ecx that stored the round counter and other round dependent variables. Note that in AES encryption and decryption are done with the same key. However, in our implementation we store encryption and decryption key separately. The reason for this is that the decryption key is composed of the last two round keys. Without storing the last two round keys the decryption would take considerably more time than the encryption because for decryption the key schedule is used in reverse order. That means we would have to go through the entire key schedule first.

```
xmm0 // encryption key: low (0-127)
xmm1 // encryption key: high (128-255)
xmm2 // decryption key: low (0-127)
xmm3 // decryption key: high (128-255)
xmm4 // current round key
xmm5 // last round key
xmm6 // current state
xmm7 // variable shortterm usage
```

Figure 5: Usage of the SSE registers.

0	0	0	1	0	2	0	3
1	0	1	1	1	2	1	3
2	0	2	1	2	2	2	3
3	0	3	1	3	2	3	3

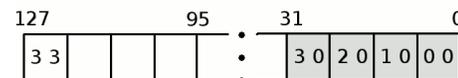


Figure 6: Mapping AES matrices onto flat SSE registers.

Probably the most important register beside the key registers is the state register xmm6. First the input block is stored here, then this register is modified over the rounds and finally it contains the output block. For the AES state it is also important to define the mapping of rows and columns. In general there are several possibilities how to map a matrix to a flat register. Figure 6 shows how we handled this: the matrix is stored column by column and — due to little endian — in reversed order.

For brevity, we just show some simple examples how the AES operations were implemented using SSE. For example, AddRoundKey can be implemented with just one assembly instruction, namely pxor xmm6, xmm4. The operation SubBytes is only slightly more complex. We just have to extract

each byte of the state, substitute it and write it back into the register. For the first element $a_{0,0}$ this can be done as shown in Figure 7. The operations `psrldq` and `pslldq` denote right shift and left shift, respectively. Note that there are no special SSE instructions to access individual bytes of an SSE register directly.

```

// load
movd    eax, xmm6
and     eax, 0x000000ff
// substitute
movb    al, [sbox+eax]
// store
psrldq  xmm6, 1
pslldq  xmm6, 1
movd    xmm7, eax
pxor    xmm6, xmm7

```

Figure 7: Substitution of the first byte.

The instruction set on SSE registers is very specialized and tailored to arithmetic operations. For example there is neither a logical `not` nor a `rotate` operation for SSE registers. Fortunately, operations like `and`, `xor` and `shift` are available, making it possible to emulate the missing ones. However, especially the fact that `rotate` is missing leads to cumbersome code for the `ShiftRows` operation in AES. Figure 8 shows how it is implemented.

```

movd    eax, xmm6
psrldq  xmm6, 4
movd    xmm7, eax
pslldq  xmm7, 12
pxor    xmm6, xmm7

```

Figure 8: Simulate right rotate on SSE registers.

3.2 Need for Operating System Support

Even if AES is implemented only on registers, it is still possible that critical data is written to RAM unless there is special support of the operating system. This is because the scheduler may decide to run another task just in the middle of an AESSE encryption and critical registers are written to RAM as part of the context switch. Consequently, we must execute AESSE atomically which is only possible in kernel mode. In Linux (and most other multitasking OSs) there is no way for user mode processes to be executed atomically (otherwise user mode processes could freeze the system).

Even if AESSE is executed atomically, we additionally have to ensure that the operating system does not write out the SSE registers to RAM during any context switch because some of these registers are persistently filled with critical data. Non-critical data is stored in general purpose registers that are reset after an AESSE operation and consequently the GPRs are safe to be handled within a context switch.

As mentioned above, SSE can be disabled in kernel mode (ring 0), effectively hiding the contents of the registers from processes running in user mode. Processes that rely on SSE will crash if they try to access disabled registers. So in effect, disabling SSE also prevents the key to be read or overwritten. The trick is to disable SSE in general and only to

re-enable it in protected AES code sections that are executed atomically in kernelspace.

On many modern architectures there is even a fourth reason to protect the code in kernel mode: For symmetric multiprocessing (SMP) machines we have to ensure that AESSE is always running on the same CPU. (Just imagine we store the key in CPU0 and decryption runs on CPU1 — obviously the decryption will fail.)

To summarize, Figure 9 illustrates the *AESSE prologue*, which contains all operations that must be done before calling AESSE to encrypt a block. Its counterpart is the *AESSE epilogue*, that contains everything that must be done after AESSE was called.

```

// run on CPU0
if (cpus > 1)
    cpumask_set_cpu(0);
// enter atomicity
local_irq_disable();
preempt_disable();
// enable SSE
movl    eax, cr4
orl     eax, 0x600
movl    cr4, eax

```

Figure 9: AESSE prologue.

3.3 Integration into the Linux Kernel

As seen above, AESSE requires some changes of the operating system code to be resistant against cold-boot attacks. This restricts us to open source operating systems, from which we have chosen Linux. However, in general our approach should work for any other operating system as well.

Usually Linux checks the CPUID early in the boot process and enables SSE if it is supported. We need to modify this process: We forge the result of CPUID and let the kernel believe there is no SSE present. Unfortunately Intel does not provide instructions to forge the CPUID on hardware level, so we have to do so in software. Just after the CPUID has been read into memory we reset the SSE flags. Figure 10 shows a patch for `arch/x86/include/asm/processor.h`.

```

static inline void native_cpuid(...)
{
+   unsigned int level = *eax;
   asm("cpuid"
       : "=a" (*eax),
         "=b" (*ebx),
         "=c" (*ecx),
         "=d" (*edx)
       : "0" (*eax), "2" (*ecx));
+   /* forge cpuid: no SSE */
+   if (level == 1) {
+       *ecx &= 0xffe7fdfe;
+       *edx &= 0xf9ffffff;
+   }
}

```

Figure 10: Forging of CPUID.

The advantage of this approach is that we have a single,

very central function that allows us to disable SSE in the entire kernel. We do not need a lot of other kernel hacking here. In fact even Linux' context switching does not swap out SSE registers anymore when we forge the CPUID like this.

We integrated the AES implementation into the kernel using the kernel *Crypto-API*. This API is an interface for different ciphers. Consequently it was relatively easy to integrate our code into this API. We left the normal AES support of the kernel untouched and inserted with AESSE a completely new crypto module into the Crypto-API. Besides simplifying the design, there were more advantages of this approach:

1. The Crypto-API handles dynamic loading of ciphers as kernel modules.
2. It also handles the modes of operation for us: We do not have to implement ECB and CBC ourselves. The Crypto-API just takes our block encryption and provides an ECB and CBC mode automatically.
3. A lot of existing software, especially disk encryption software such as dm-crypt, is already using the kernel Crypto-API and open to all new ciphers. So we do not have to patch these programs to be compatible with our implementation.

However, there is also a drawback: The Crypto-API comes with its own key management. This key management was too insecure for us, since it stores the keys in RAM. To overcome this problem without hacking the Crypto-API too much, we passed a dummy key and took care of the real key ourselves, as we explain in the next section.

Overall, there are some minor changes to a wide range of files in different source directories of the kernel, e.g., to `/arch/x86/boot`, `/crypto`, `/init` and `/kernel`. But more than ninety percent of our patch went to `/arch/x86/crypto/`. This is the main directory where we implemented AESSE. The choice of this directory is clear: AESSE is x86 architecture dependent, because SSE is so and most of our code is written in x86 assembly.

3.4 Key Management

The final question refers to bootstrapping: How do we actually get the key into the SSE registers without using the RAM in the first place? The answer is, that we *do* use the RAM for the transaction of the key but only very shortly. The idea is to prompt the user for a key early in the boot process, far before any user mode process is started. So the key prompt is displayed from within kernel mode. To generate the AES key the user is asked to enter a password between eight and 53 characters from which the key is derived as SHA256 hash. There is already a SHA256 algorithm in the kernel that we could use, but we implemented our own SHA256 variant to have greatest possible control about where and when parts of the key are in memory. Our SHA256 variant takes care that all critical memory locations that are not used anymore are set back to zero. So after the key has been written into SSE registers, we just delete all occurrences of the key and the password.

In a similar way we have to read in the key after suspend. In suspend mode all CPU registers are swapped out to RAM and the CPU is powered off. Our patched kernel naturally does not swap out the SSE registers. Therefore the SSE

text case	AES	AESSE	slowdown
key 128, input 128 bit	1,1 μ s	2,5 μ s	2,27
key 128, input 8 kB	135 μ s	858 μ s	6,35
key 192, input 128 bit	1,2 μ s	3,0 μ s	2,5
key 192, input 8 kB	163 μ s	1130 μ s	6,93
key 256, input 128 bit	1,1 μ s	3,0 μ s	2,72
key 256, input 8 kB	189 μ s	1081 μ s	5,72

Table 1: Encryption performance of AES vs. AESSE.

registers cannot be restored on wakeup and the user is asked to enter the password manually again. This might be a drawback for some people, but it is a natural implication of our requirements and can only be alleviated using tamper-proof hardware.

4. EVALUATION

4.1 Performance Analysis

We measured the performance of our implementation of AESSE in comparison to the standard AES implementation on an Intel Pentium M / 1.5 GHz platform running Linux 2.6.30.

Our first implementation was about 100 times slower than standard AES. The reason was that we programmed AES very close to the official FIPS documentation [10], whereas the kernel is using a highly optimized variant of AES designed by Gladman [3]. Gladmans algorithm achieves most of its performance gain by the use of lookup tables. Almost all AES operations, such as `ShiftRows` and `MixColumns`, can be replaced by quick table lookups. These tables are relatively big and must reside in RAM. But this does not violate our strong security requirements, because the lookup tables are key-independent and publicly known. Still no critical data is going to RAM. So we developed a new version of AESSE from scratch that is also based on the Gladman algorithm.

The performance results are shown in Table 1. Overall, there is still a performance penalty for using AESSE instead of AES. Our current AESSE implementation is about six times slower than the kernel's standard AES. But a performance drop of factor six is relatively good compared to what we started with. This performance drawback is, if at all, only little noticeable on standard desktop systems working with office applications or web browsers. Though, copying large files such as videos, the performance drawback becomes perceptible.

We conjecture that the main reason for the performance penalty is the shortage of memory which forces us to recalculate the round keys on demand. For each 128 bit input block 10 roundkeys have to be calculated. When encrypting megabytes of data this is a substantial overhead compared to standard AES that calculates the roundkeys only once.

Another reason for the decrease of performance is the limited SSE instruction set. We sometimes need four or more instructions where standard AES can use a single instruction. Just remember the effort we spent to write back a single byte to an SSE register in Figure 7. To write back a single byte to main memory we would need only one instruction.

Finally, there is also some overhead due to the AESSE

prologue and epilogue. We have to enter atomic mode and to enable SSE on each 128 bit input block, even when encrypting megabytes of data. Note that it is no alternative to run the AESSE prologue once per kilobyte or once per megabyte. Firstly, the overhead here is negligible compared to the overhead of the roundkeys. Secondly, this would slow down the entire system because the encryption of kilobytes or megabytes of data would occur atomically. With 128 bit portions other applications can work as usual while heavy encryption tasks are processed in parallel.

4.2 (Brief) Security Analysis

From a security point of view we still need to do some evaluations that prove that our implementation lives up to its promise. Indeed it is clear that the AESSE assembly does not use RAM, since this is just the way we programmed it. But there may be some side channel and operating system effects that have not been taken into account yet.

Examples for operating system effects that we have already dealt with are context switches and the ACPI suspend mode. In both cases SSE registers are swapped out to RAM normally. But so far we have no good justification that we are not missing something here. So one of the important points we have to do next is to search the entire kernel for instructions that imply SSE registers and prove each of them secure.

When we find further locations in the kernel that do swap out SSE registers, we are most likely able to patch these kernel parts as well. The critical problem is another one: The hardware layer. There might be ways to read or overwrite SSE registers even if SSE is disabled. For example there is the assembly instruction `fxsave` that saves *all* registers to RAM and can be executed in user mode. Would this instruction store SSE registers when SSE is disabled, all user mode processes could read out the key. So exactly what is meant with *all* registers? Intel itself defines the semantics of this operation as follows:

If the OSFXSR bit in control register CR4 is not set (= SSE off), the FXSAVE instruction may not save these (= the SSE) registers. This behavior is implementation dependent. [7]

Fortunately, all processors we have tested do not save the SSE registers. But there is no guarantee that this will remain the case with future processors.

5. CONCLUSIONS

In this paper, we presented AESSE, an implementation of AES that is resistant to cold boot attacks on memory. We have discussed how we integrated it into the Linux kernel and shown the performance in comparison to the standard implementation of AES.

Our implementation ideas work for other ciphers, too. For example, we have also implemented DES in the manner described in this paper, resulting in “DESSE”. Both DESSE and AESSE can be used as ciphers to perform disk encryption in Linux since dm-crypt uses the kernel Crypto-API.

Maybe the most severe drawback of our approach is that the patched system loses binary compatibility with applications that use SSE. If the source code of such applications is available, in most cases it is quite easy to patch them in such a way that SSE is not used. Basically, there are two cases that we have to distinguish: If the programmer of the

application explicitly used SSE assembly instructions, the program needs to be patched manually. Fortunately, in most cases every application comes with an implementation that does not need SSE for compatibility with older hardware. Thus, patching them to a SSE-free version is as simple as branching into this non-SSE compatible code. If, however, the programmer did not explicitly want to use SSE but the compiler decided to use it for optimization reasons, using a patched GCC did the trick for us. This way we already managed to get nearly all applications that are common in desktop environments running with AESSE.

Acknowledgements

We would like to thank Dirk Stegemann and Stefan Voemel for reading through this paper and giving us valuable suggestions for improvement.

6. REFERENCES

- [1] Bitlocker drive encryption. Internet: <http://technet.microsoft.com/en-us/windows/aa905065.aspx>, November 2009.
- [2] Utimaco safeguard cryptoserver. Internet: <http://hsm.utimaco.com/home/>, February 2010.
- [3] Brian Gladman. BRG Main Site. Internet: <http://www.gladman.me.uk>, February 2010.
- [4] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *Proceedings of the 17th USENIX Security Symposium*, pages 45–60, 2008.
- [5] Infineon Technologies. Embedded Security. Internet: <http://www.infineon.com/tpm>, February 2010.
- [6] Intel. Advanced Encryption Standard (AES) Instruction Set. Internet: <http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set/>, February 2010.
- [7] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 2A: Instruction Set Reference A-M*, December 2009.
- [8] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 2B: Instruction Set Reference N-Z*, December 2009.
- [9] Carsten Maartmann-Moe, Steffen E. Thorkildsen, and Andre Arnes. The persistence of memory: Forensic identification and extraction of cryptographic keys. *Digital Investigation*, 6(1):S132–S140, 2009. Finding cryptographic keys in memory.
- [10] NIST. FIPS 197: Advanced Encryption Standard (AES). Technical report, NIST, 2001.
- [11] Martin Oczko. KryptoNAS: Open Source Based NAS Encryption. In Norbert Pohlmann, Helmut Reimer, and Wolfgang Schneider, editors, *ISSE 2009 Securing Electronic Business Processes*. Vieweg+Teubner, 2009.
- [12] Larry Ponemon. *Airport Insecurity: The Case of Missing & Lost Laptops*. Ponemon Institute, June 2008. http://www.dell.com/downloads/global/services/dell_lost_laptop_study.pdf.
- [13] Christophe Saout. dm-crypt: a device-mapper crypto target, 2006.