Lehrstuhl für Informatik 1
Friedrich-Alexander-Universität
Erlangen-Nürnberg

BACHELOR THESIS

Advanced Vector Extensions to Accelerate Crypto Primitives

Johannes Götzfried

Erlangen, July 31, 2012

Examiner: Prof. Dr. Felix Freiling
Advisor: Tilo Müller

Eidesstattliche Erklärung / Statutory Declaration

Hiermit versichere ich eidesstattlich, dass die vorliegende Arbeit von mir selbsttändig, ohne Hilfe Dritter und ausschließlich unter Verwendung der angegebenen Quellen angefertigt wurde. Alle Stellen, die wörtlich oder sinngemäß aus den Quellen entnommen sind, habe ich als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

I hereby declare formally that I have developed and written the enclosed thesis entirely by myself and have not used sources or means without declaration in the text. Any thoughts or quotations which were inferred from the sources are marked as such. This thesis was not submitted in the same or a substantially similar version to any other authority to achieve an academic grading.

Erlangen, July 31, 2012

Johannes Götzfried

Zusammenfassung

Verschlüsselung spielt eine große Rolle in der heutigen IT-Sicherheit und wird in der Regel täglich von jedem benutzt, der mit elektronischen Geräten arbeitet. Obwohl die Rechenleistung stetig wächst, bleibt die Effizienz von kryptographischen Algorithmen ein wichtiges Thema. In dieser Arbeit werden daher schnelle Implementierungen der fünf symmetrischen Blockchiffren Serpent, Twofish, Blowfish, Cast-128 und Cast-256 vorgestellt. Die Implementierungen benutzen die *Advanced Vector Extensions* (AVX), ein neues SIMD Instruktionsset, und den Nachfolger AVX2, der von zukünftigen Prozessoren unterstützt wird. Des Weiteren wurde die AVX Implementierung von jedem Algorithmus in den Linux Kernel integriert und überbietet die derzeitig schnellste Implementierung in jedem der fünf Fälle.

Abstract

Encryption plays an important role in today's IT-Security and is generally used by everybody who is working with electronic devices on a daily basis. Although computing power increases steadily, the performance of cryptographic algorithms remains an important topic. In this thesis fast implementations of the five symmetric block ciphers Serpent, Twofish, Blowfish, Cast-128 and Cast-256 are presented. The implementations make use of the *Advanced Vector Extensions* (AVX), a newly introduced SIMD instruction set, and its successor AVX2, which will be supported by future CPUs. Furthermore the AVX implementation of each algorithm has been integrated into the Linux kernel and outperforms the currently fastest implementation in each of the five cases.

CONTENTS

1 INTRODUCTION

Disk encryption is now widely spread and there are numerous software solutions available, which protect your private data or even your whole hard disk. Most popular services on the Internet, e.g. SMTP, HTTP and SSH, work at least optionally with encryption as additional layer to provide transport security. Hashing is often used to verify a message's integrity and in combination with asymmetric ciphers signatures can be implemented. Software distributors are able to sign their packages with a master-key to verify that they have not been modified and the services mentioned above are able to use signatures to prevent man-in-the-middle attacks.

Despite all the advantages of these techniques, however, there are drawbacks. The performance of a system, that uses cryptography, cannot be as good as the performance of a system, that does not use it. But there are many cases, where the use of encryption techniques is mandatory. More and more people carry sensitive and confidential data about. Employees, that work outside their office or need to travel a lot, are often forced by company regulations to use disk encryption for their mobile devices. A remote connection for controlling machines needs to be secure against eavesdropping and man-in-the-middle attacks as well as a connection used for online banking. In that cases lower performance has to be accepted, but by speeding up cryptographic algorithms the decrease can be minimized.

Because of these and other use cases, the performance of cryptographic algorithms is a big issue. Low-level implementations of crypto primitives, that exploit platform specific assembler implementations, can achieve a considerable gain in performance as compared to generic C implementations. For this thesis, five common symmetric block ciphers, Serpent, Twofish, Blowfish, Cast-128 and Cast-256, have been implemented, exploiting the new *Advanced Vector Extensions* (AVX) and its successor AVX2. As software-based solutions, the AVX implementations are running on recent CPUs, which support AVX. The AVX2 implementations will run on upcoming Intel CPUs launching market not before 2013. Additionally a patch for the Linux kernel has been developed for each AVX implementation. This makes it possible to use the AVX implementations on any standard Linux system in combination with the device mapper dm-crypt for disk encryption or any other application relying on the kernel crypto API.

1.1 Motivation

Some time ago processor manufactures introduced SIMD instruction sets, which allow to process multiple data with only one instruction. Many of the scalar instructions have SIMD equivalents and can be used to operate on more values in parallel. In practice SIMD instructions operate on large registers, that are for example 128 bit wide. They have the ability to fill registers from memory and to modify parts of them in a similar manner. There have been some of these SIMD instruction sets in history, namely MMX, SSE, SSE2, SSSE3, and Intel has released a new instruction set called AVX (Advanced Vector Extensions) [36] in 2011. AVX has some interesting new features. The 128 bit registers have been expanded to 256 bits, a nondestructive three-operand syntax has been added and a new extension coding scheme (VEX) has been designed. With this new features AVX is a good choice for accelerating crypto primitives.

At the beginning of this thesis, it has also been considered to speed up hash algorithms. Currently there is a competition [50] running to select a new SHA-3 function and to replace SHA-1 and SHA-2. Since theoretical attacks on SHA-1 have been carried out and SHA-1 and SHA-2 are algorithmically similar it has been decided to start a competition for a new hash standard. In this competition there are five finalists BLAKE [6], Grøstl [16], JH [59], Keccak [8] and Skein [15]. One big selection factor for the winner of this competition is, besides security, performance and that is why there were plans to accelerate one of the five SHA-3 finalists with AVX.

It has turned out, however, that symmetric ciphers can get a better performance boost, because in most modes of operation, it is possible to process many blocks in parallel, whereas with hash algorithms the blocks would have to be processed sequentially. Moreover the integration into the Linux kernel would probably not have been possible in the way, we did it with the symmetric ciphers, because the finalist has not been selected yet.

As stated above there is a demand for fast implementations of symmetric block ciphers. With AESNI [31] Intel introduced an instruction set, which implements AES [43], the most popular and widely spread block cipher, in hardware. This step made a very fast implementation of AES available to everybody, who uses x86 hardware. Other symmetric block ciphers, however, which are widely used, too, cannot benefit from the AESNI instruction set and therefore they have to be accelerated with a different approach. The current situation leads to the motivation to reimplement popular block ciphers with AVX. The ciphers related to AES, for example Serpent [4] and Twofish [53], are of special interest, because they were among the five finalists in the AES competition and have practical relevance.

As disk encryption is a very popular and the most performance critical use case of symmetric ciphers, it is a good choice to test our implementations in that field. There are different software solutions available and among these the Linux kernel itself provides support for all kinds of disk encryption. As Linux is an open source operating system, we are able to integrate our ciphers into the kernel and make meaningful speed tests in kernel space. Furthermore, integrating our ciphers into the Linux kernel makes the implementations available to many people at once and helps reproducing the results of this thesis.

1.2 Task

The goal of this thesis is to improve the performance of symmetric block ciphers using AVX. For each algorithm, Serpent [4], Twofish [53], Blowfish [52], Cast-128 [1] and Cast-256 [2], a plain assembler implementation has to be developed. All implementations are based on the concept of parallel processing sequenced blocks. The challenge is to transform the specification or the reference implementation to an implementation that makes use of the SIMD instructions and exploits the resulting parallelism. But there are operations, which cannot be transformed to equivalent SIMD operations, and in this case we have to work around the problem and make it possible to exploit AVX anyway.

AVX lacks some instructions, basically table lookups, which would be necessary to fully parallelize all implementations. Furthermore the 256 bit wide registers cannot be used efficiently with integer instructions, but instead these operations can just be applied on 128 bit wide registers. Both facts are serious drawbacks

and therefore every algorithm has also been implemented with the successor AVX2, which does not have these restrictions. For AVX2, however, there currently exists no hardware, which could be used to test the implementations, and therefore the AVX2 implementations have to be tested within an emulator [10]. Of course it is not possible to measure the exact time our implementations will take on future CPUs, but based on the instructions used, it can be guessed how fast they will be, once the hardware is available.

The integration of an AVX implementation into the Linux kernel requires some extra work. The symmetric ciphers do not need to be adjusted, but a lot of gluecode has to be written to make the integration possible. The gluecode needs to register the cipher within the crypto API of the kernel and reimplement the modes of operation, which are provided by the kernel, because the ciphers process many blocks at once and therefore cannot make use of the standard implementation of the modes of operation. It is necessary to provide new and large testvectors for the specific ciphers in all available modes to allow the kernel doing a self-test. The implementations are integrated in such a way, that they can be loaded as separate kernel module. For the Cast-128 and Cast-256 integration even the generic implementations have to be adjusted, because at the time of this writing there has no accelerated implementation been present.

Besides all this details, the goal of this thesis is simple to summarize. It should be possible to get a significant performance boost for popular symmetric ciphers, exploiting AVX and AVX2.

1.3 Related Work

There are many attempts to accelerate crypto primitives. Usually a reference implementation is shipped along with the specification of a cryptographic algorithm and the authors already give hints, which parts are able to be parallelized or optimized. Often the authors themselves provide an accelerated version of the algorithm and for most ciphers at least an assembler implementation, that does not make use of SIMD instructions, exists. Many implementations, which exploit previous instruction sets, exist as there is demand for fast ciphers.

Regarding AVX, there are already implementations of crypto primitives, that exploit the new instruction set. Among the SHA-3 candidates Blake has already been implemented with AVX and AVX2 [47] and for Grøstl there also exist implementations, that make use of AVX and AVX2 [38].

During the work on this thesis and while submitting the patches with the AVX implementations to the Linux kernel, it has been discovered that one kernel developer, Jussi Kivilinna, works on AVX2 implementations of symmetric ciphers and their integration into the kernel, too [35]. He implemented Serpent, Twofish and Blowfish in a way, which is similar to the one in this thesis, but there are differences as well. He did not provide AVX implementations, however, and at the moment no other AVX implementations of the five ciphers, which were implemented during this thesis have been found.

Nevertheless these examples prove, that providing fast implementations of cryptographic algorithms is currently a hot topic in practical IT-Security.

1.4 Results

For five different algorithms fast implementations are provided with this thesis and for every algorithm an AVX implementation, an AVX2 implementation and a Linux kernel patch has been developed. Every of the five AVX implementations is faster than the previously fastest implementation in the Linux kernel has been. The AVX2 implementations have been evaluated based on the instruction counts and are expected to be a lot faster than the AVX implementations, but no timing results can be provided for them before AVX2 CPUs will be available in 2013. With the five ciphers Serpent, Twofish, Blowfish, Cast-128 and Cast-256 we have accelerated today's most important symmetric block ciphers, aside from AES, for which with AESNI already a fast implementation in hardware is available.

Our Serpent implementation is 6.1% faster than the current SSE2 implementation and Twofish has been accelerated by 30.8% compared to the previous 3-way parallel assembler implementation. Cast-128 is

about 115.8% faster and for Cast-256 we get a speedup of 88.6%. Finally for Blowfish we get a speedup of 0.8%, which is slightly better, but too low, to submit the cipher to the Linux kernel. These timings could be reproduced in userspace, kernelspace and while measuring the data rate in disk encryption tests.

In total ten different assembler implementations have been developed, five AVX implementations and five AVX2 implementations, and ten kernel patches have been submitted. All patches have been accepted and four patches, one patch for Serpent and three patches for Twofish, have already been merged into mainline. The other six patches, three for Cast-128 and another three for Cast-256, are about to be merged. Our implementation of Serpent and Twofish will be available with the official kernel version 3.6.

1.5 Outline

In chapter 2 the background information is given, which is necessary to understand the work, which has been done in this thesis. In section 2.1 an overview of symmetric ciphers is given, the modes of operation are explained and the candidates from the AES competition will be introduced. After this the five ciphers Serpent, Twofish, Blowfish, Cast-128 and Cast-256 are explained in detail, because they will be implemented later on. The AVX instruction set is shown in section 2.2 together with the successor AVX2. Finally in section 2.3 some explanations about the Linux kernel and particularly about kernel development and the crypto API are given.

Chapter 3 explains in detail, how the five ciphers have been implemented and integrated into the kernel. For every cipher the AVX implementation will be shown first and after that the differences that lead to the AVX2 implementation are described. Last but not least the kernel integration will be shown for every of the five ciphers.

In chapter 4 we make some tests with our implementations and see them from a different angle. In section 4.1 we show the source code statistics of the different kernel patches to give an overview on what has been changed by what patch. After that we want to show possibilities how the implementations can be used (section 4.2) and make plausible why our implementations are correct (section 4.3). The most important part is section 4.4, which contains a detailed performance evaluation of every implementation. Finally there are some considerations about the security of our implementations in section 4.5.

1.6 Acknowledgments

I want to thank my advisor Tilo Müller for giving me the chance, to work on this interesting topic, and supporting me. Furthermore I want to thank the Chair for IT Security Infrastructures for lending me a computer with an Intel Core i5-2500 CPU, which supports the new AVX instruction set natively and was used heavily during this thesis.

Last but not least, I want to thank my family, first and foremost my parents, for their personally encouragement and incessant support of my entire studies.

2 BACKGROUND

This chapter provides you with the background information, which is useful or necessary to understand the implementation part of this thesis. First in section 2.1 an introduction about symmetric ciphers is given and the difference between symmetric and asymmetric ciphers will be explained. It would be beneficial to have some knowledge about this topic in advance, but the explanations given should be sufficient to understand most parts of this work. In this section the modes of operation (section 2.1.1) will be covered, which are needed to use ciphers on messages with arbitrary length. After that in section 2.1.2 the AES competition will be presented, where five different block ciphers have been chosen as finalists, which share some common properties. In section 2.1.3 and 2.1.4 two out of the five finalists will be introduced and explained in detail, because we will provide fast implementations for them. After that we will introduce three more symmetric ciphers, which were not among the five finalists, but are related to them. For every of the five presented ciphers we will provide a fast implementation and a kernel patch later on.

In section 2.2 the Advanced Vector Extensions are introduced. This is the extension, which we will use to implement our algorithms. It is no harm to have some background information about SIMD instruction sets, like MMX or SSE, in advance, but it is not necessary to understand the explanations given here. Nevertheless this section is not a guide to assembler programming in general and so you should at least be familiar with generic x86_64 assembly and have some knowledge about the general purpose registers. In section 2.2.1 the new registers are presented and in section 2.2.2 a for this thesis relevant subset of the available instructions will be shown. Section 2.2.3 gives an overview of AVX2, the successor of AVX, which is not yet available on real hardware, but can already be used to develop applications and has some huge advantages over AVX.

We want to commit our ciphers to the Linux kernel, and that is why in section 2.3 some short informations about the kernel will be provided. In section 2.3.1 the development process of the kernel is explained. There are some useful informations you should know, if you want to contribute to the kernel. Section 2.3.2 shows the source tree of the Linux kernel and points out the positions where we have to place our implementations. Finally in section 2.3.3 a high level description of the Linux crypto API is provided. This API is used for all kind of cryptographic algorithms. Besides this API we will not need much internal kernel functionality and so it is not necessary to be an expert in kernel development to understand the patches we provide with our ciphers.

2.1 Symmetric Ciphers

Symmetric ciphers use the same key for the encryption of a plaintext message and the decryption of the corresponding ciphertext. There also exist asymmetric ciphers, which in contrast to symmetric ciphers, provide a public and a private key, so that the message can be encrypted using the public key and decrypted with the private key. This has the advantage, that no secret key has to be shared between the two parties, which want to exchange secret messages. One big drawback of symmetric cryptography is that both parties have to know a common secret key and therefore need to exchange this key over a somehow secured connection that nobody is able to eavesdrop. Asymmetric cryptography, however, is build on specific algebraic properties and therefore much slower than symmetric cryptography. This one reason, why symmetric cryptography is still very important and in many cases a hybrid approach is chosen. It is common, to use asymmetric cryptography to exchange a secret key over an insecure connection and afterwards use symmetric cryptography with the shared secret key to exchange the actual information. As symmetric ciphers are used to encrypt the actual information, it is important, that they are very fast. Only symmetric cryptography is covered in this section. Asymmetric cryptography works completely different and to gain a performance boost would require a totally different approach.

When speaking of symmetric cryptography there exist in fact two types of algorithms: stream ciphers and block ciphers. Stream ciphers can take a message of arbitrary length and encrypt it on the fly, i.e. digit by digit, whereat a digit is typically a bit. This can be achieved by generating some pseudo random keystream from an initially random seed and somehow operate with this keystream on the message. Usually just an exclusive-or is used to combine the keystream with the original message. Block ciphers, in contrast, take exactly one block, i.e. a fixed length of bits specified by the cipher (typically 64 or 128 bit), as input and usually output an encrypted block with exactly the same size. Many modern block ciphers work essentially similar. They need to be initialized with a key of fixed length (typically between 64 and 512 bit) and do some key scheduling to generate round keys out of the supplied fixed length key. In the encryption or decryption routine they process the data in a cipher and key size specific number of similar designed rounds. Often the rounds only differ in the round key used from the key schedule. Common operations in these rounds are substitutions, permutations and key mixing operations. Depending on these operations the block ciphers are classified in different categories, for example as substitution-permutation-network or Feistel network. The details are discussed in the specific sections, for example in section 2.1.3 and 2.1.4, where the two algorithms Serpent and Twofish are introduced. To be able to encrypt messages of arbitrary length with block ciphers, two additional mechanisms are needed. First of all the message might have to be padded, to get a length, which is a multiple of the block size. Now there are different modes of operation, which specify how the block cipher should be used to encrypt the whole message, because the cipher itself is only able to encrypt exactly one block. The different possibilities will be shown in the next section.

2.1.1 Modes of Operation

The modes of operation, which should be used in conjunction with symmetric block ciphers, are published by the National Institute of Standards and Technology (NIST). The first modes of operation were published in 1980 to be used with the former Data Encryption Standard (DES) and are specified in FIPS 81 [41]. FIPS 81 already lists ECB and CBC, which is still in use today. In 2001, NIST renewed the specification, added the CTR mode and propagated the Advanced Encryption Standard (AES) as the default block cipher for these modes [44]. Nine years later in 2010, NIST added the XTS mode, which is used by disk encryption systems [45]. There exist more modes than the ones published by NIST, but this are the most important ones. When we want to speed up block ciphers by processing blocks in parallel, we have to know the different modes, as we have to reimplement them, to give them the ability to deal with our parallel implementation. We also have to check, whether the mode of operation actually supports parallel processing or not. In this section we want to take a closer look on three of the modes: ECB, CBC and CTR. This are three of the modes, available in the Linux kernel, and it should be possible to use them in conjunction with every block cipher.

The simplest of the modes is the *electronic codebook* (ECB) mode. For this mode the message has to have a length, which is a multiple of the block size of the underlaying block cipher. If this is not the case,

padding is required. There exist several padding schemes and one common method is to add a single one bit followed by as much zero bits, as are necessary to fill the block. This has the advantage that the actual length of the message can be reconstructed, whereas this would not be possible by just adding zeros. The disadvantages are, that a whole block has to be added, if the message ends on a block boundary and that the lengths of the plaintext and the ciphertext are different. Of course no padding is required, if you can be sure, that the input data always has the right length. With ECB the message is just divided into blocks and each block is encrypted separately with the same key. Security is only provided on a per block basis, but the overall structure may be recovered, as identical plaintext blocks lead to identical ciphertext blocks. Decryption works similarly to the encryption and as the blocks are processed separately, parallelization is perfectly possible. Nevertheless, due to security issues, this mode should no longer be used in any field of application.

Another mode is *cipher block chaining* (CBC), which is very old, but still widely used and considered secure. It is the default mode used by most of the disk encryption systems. CBC essentially requires padding as well, however there exist more sophisticated padding schemes. The disadvantage of getting a different ciphertext length than the actual plaintext length can be resolved with a method called *ciphertext stealing* [46]. With ciphertext stealing some data of the next to last block is used for padding and thus encrypted twice. In contrast to ECB, with CBC the blocks are not encrypted separately, but instead each plaintext block is exclusive-or'ed with the previous ciphertext block before being encrypted. The first plaintext block is xor'ed with an *initialization vector* (IV). Figure 2.1 visualizes the encryption process. Because of chaining the blocks this way, every block is dependent upon its previous blocks and with the IV it is even possible to encrypt equivalent messages differently with the same key. The drawback of the chaining is, that we are forced to process the blocks sequentially. Decryption, however, can be parallelized, because each block is exclusive-or'ed with the previous chiphertext block *after* being decrypted. This means decryption and the following xor operation can be done in parallel.

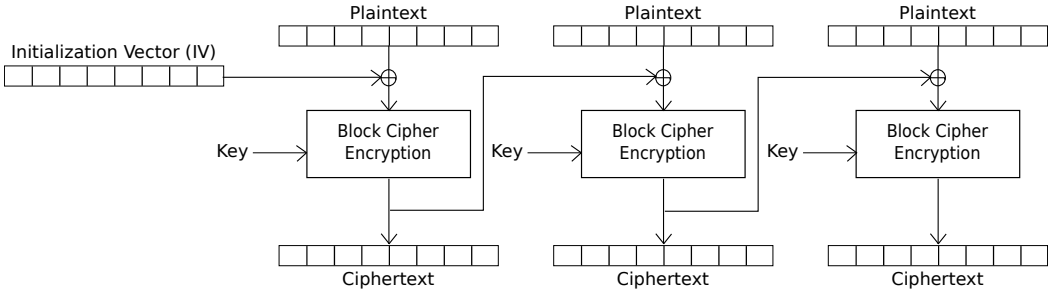

Figure 2.1: Cipher block chaining (CBC) mode encryption

The last mode presented here is *counter* (CTR). CTR does not require the message to be padded and is special, because it is able to turn the underlaying block cipher into a stream cipher. Therefore the keystream, or to be more precise, the next block of the keystream, is generated by encrypting successive values of a counter. The counter is independent from the actual plaintext message and may be a function producing a sequence. Of course CTR becomes more secure, if the sequence does not repeat, because otherwise the same problem as with ECB occurs. Most of the time, this function is just an increment-by-one, the Linux kernel uses this simple function as well, and some nonce may be added, which serves the same purpose as the initialization vector with CBC. After that, the keystream is simply xor'ed with the plaintext message. This can be done blockwise or digit by digit like an ordinary stream cipher would do it. Figure 2.2 shows an illustration of the encryption process. The decryption works exactly like the encryption, so the encryption routine of the block cipher is used for the decryption as well, to generate the identical keystream the same way. After that, the keystream is xor'ed with the ciphertext to reconstruct the original plaintext. As you can see in figure 2.2, there is no chaining between the different blocks and so both, encryption and decryption, can be easily parallelized. It might look strange, that the block cipher itself does not process the actual message, but just some counter values, however despite that, CTR is considered to be secure.

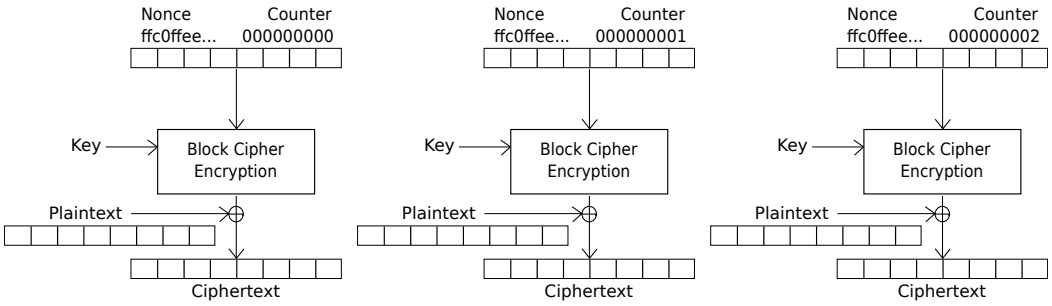

Figure 2.2: Counter (CTR) mode encryption

2.1.2 AES Competition

At this point you should have a rough overview on what a block cipher is and you also have seen how these block ciphers can be used as building blocks to encrypt or decrypt messages of arbitrary length. There are numerous block ciphers available out there and some of them are broken or considered insecure by now. The probably most popular ciphers today are finalists of the AES competition. In 1997 the National Institute of Standards and Technology (NIST) decided to start a competition to choose a successor of the Data Encryption Standard (DES) [42]. This was necessary, because DES, which uses only a 56-bit key and a block size of 64 bits, became vulnerable to brute force attacks. Consequently the new algorithms were forced to support key sizes of 128, 192 and 256 bits and process a block size of 128 bits [48]. In 2000, NIST announced, that Rijndael was chosen out of the five finalists Rijndael [43], Serpent [4], Twofish [53], RC6 [51] and MARS [9], which were rated in that order. Rijndael, now known as the Advanced Encryption Standard (AES) [43], was especially chosen, because it offered a good combination of security and performance [49].

Despite there is only one winner, none of these algorithms is clearly broken or has some kind of known security vulnerability, rather they are still in use today. Because of their importance, it seems to be a good choice to speed up algorithms out of these five finalists. MARS will not be selected, because it is the last one of the five finalists and has not that much practical relevance in today's applications. RC6 is not royalty-free and therefore will not be selected either, last but not least because an integration into the kernel would be impossible. Furthermore it is pointless trying to speed up Rijndael (or AES) with SIMD instructions, because there exists a very fast and efficient hardware implementation, called AESNI [31], which is available with every processor, supporting AVX. With AESNI it is possible, to do one round of the encryption or decryption routine with a single instruction. A round in AES consists of four steps: a substitution step, a shift step, a mix step and one step, which combines the state with the round key. In practice a software implementation would replace most of this steps by some table lookups, but even with this technique we have no chance of being faster than one single instruction. After this considerations, there are two finalists left: Serpent and Twofish. Both still have practical relevance and as there exists no specific hardware implementation on generic processors, it is a good choice to accelerate this two algorithms.

2.1.3 Serpent

Serpent [4] was the second best rated block cipher in the AES competition. As AES itself, it can be categorized as *substitution-permutation network* (SPN). A SPN is a series of mathematical operations, which consists of several rounds of substitutions and permutations. A substitution box, often referred to as S-box, is responsible for substituting a block of bits by another block of bits. As every operation in the SPN should be invertible to provide decryption, the S-box should output bits of the same length as the input and be injective and surjective. A permutation permutes all the bits, which the S-boxes give as output, and they can be used as input for the S-boxes of the next round. These two operations combined provide diffusion and confusion properties, which were already identified by Shannon in 1949 [54]. Diffusion means, that if the plaintext message changes slightly, the ciphertext should change completely. This is achieved by well chosen S-boxes. Confusion means, that changing the key slightly should change the ciphertext substantially.

$$
\begin{aligned}
\hat{B}_0 &:= IP(P) \\
\hat{B}_{i+1} &:= R_i(\hat{B}_i) \\
C &:= FP(\hat{B}_{32})
\end{aligned}
$$

where

$$
\begin{aligned}
R_i(X) &= L(\hat{S}_i(X \oplus \hat{K}_i)) && i = 0, \ldots, 30 \\
R_i(X) &= \hat{S}_i(X \oplus \hat{K}_i) \oplus \hat{K}_{32} && i = 31
\end{aligned}
$$

Figure 2.3: High-level pseudocode of Serpent

$$
\begin{aligned}
X_0, X_1, X_2, X_3 &:= \hat{S}_i(\hat{B}_i \oplus \hat{K}_i) \\
X_0 &:= X_0 <<< 13 \\
X_2 &:= X_2 <<< 3 \\
X_1 &:= X_1 \oplus X_0 \oplus X_2 \\
X_3 &:= X_3 \oplus X_2 \oplus (X_0 << 3) \\
X_1 &:= X_1 <<< 1 \\
X_3 &:= X_3 <<< 7 \\
X_0 &:= X_0 \oplus X_1 \oplus X_3 \\
X_2 &:= X_2 \oplus X_3 \oplus (X_1 << 7) \\
X_0 &:= X_0 <<< 5 \\
X_2 &:= X_2 <<< 22 \\
\hat{B}_{i+1} &:= X_0, X_1, X_2, X_3
\end{aligned}
$$

Figure 2.4: Linear Transformation L

This is done by the permutations, because they exchange the bits in a way, that the different S-boxes again are able to provide diffusion by getting slightly different input. In every round, the round key, which is obtained in the key schedule, has to be adopted to the blocks in some way. This can be done by providing key-dependent S-boxes, but in Serpent the generated round keys are simply xor'ed with the intermediate state.

As all the AES competition candidates, Serpent has a block size of 128 bits and supports key sizes of 128, 192 and 256 bits. Serpent consists of 32 rounds, operating on four 32 bit doublewords (this corresponds to the block size of 128 bits). Compared to AES, which has ten, twelve or fourteen rounds, depending on the key size, Serpent is considered more secure, but AES is faster and therefore was selected as the winner. The authors of Serpent state that they were rather conservative on security issues and specified more rounds than actually necessary, just to be sure [4]:

> 16-round Serpent would be as secure as triple-DES, and twice as fast as DES. However, AES may persist for 25 years as a standard and a further 25 years in legacy systems, and will have to withstand advances in both engineering and cryptanalysis during that time. We therefore propose 32 rounds to put the algorithm's security beyond question.

Serpent encrypts 128 bit of plaintext P to 128 bit of ciphertext C in 32 rounds under the control of 33 128 bit subkeys $\hat{K}_0, \ldots, \hat{K}_{32}$. The cipher consists of an initial permutation IP, 32 rounds, each consisting of a key mixing operation, a pass through S-boxes and a linear transformation, and a final permutation FP. In the last round an additional key mixing operation is used, instead of the linear transformation. Figure 2.3 shows the high-level pseudocode of the encryption routine. $\hat{B}_0, \ldots, \hat{B}_{32}$ are the intermediate values between the different rounds. Each round function R_i uses 32 copies of a single 4x4 S-box, which can be applied in parallel. For example in round R_0, S_0 is applied on bits 0, 1, 2 and 3 of $\hat{B}_0 \oplus \hat{K}_0$ and returns the first four bits of an intermediate vector. After that S_0 is applied on bits 4-7 of $\hat{B}_0 \oplus \hat{K}_0$, returning the next four bits of the intermediate vector and so on. The intermediate vector is then transformed by the linear transformation L, shown in figure 2.4, resulting in $\hat{B}_1$. In figure 2.4 $<<<$ donates left rotation by the given value, $<<$ donates left shift and $\oplus$ is a simple xor operation. Round R_0 is finished and the other rounds work exactly the same, as shown in the pseudocode. Serpent provides eight different 4x4 S-boxes, meaning that they map four input bits to exactly four output bits. Thus in figure 2.3, $\hat{S}_i$ is the application of the S-box $S_{i \bmod 8}$ 32 times on the subsequent 4 bit groups of the 128 bit input. The eight S-boxes $S_0, \ldots, S_7$ are generated with a deterministic and iterative algorithm that is repeated until the S-boxes satisfy specific diffusion properties. The details of this algorithm and the properties the S-boxes satisfy after being generated can be looked up in the official publication [4]. For implementation issues the generation method is not relevant as all eight S-boxes are listed in the appendix of the publication. The initial and final permutations are listed in the appendix as well.

The concrete implementation of Serpent is usually done in bitslice mode. In this mode the initial and final permutation is not needed, because it was just constructed to make the bitslice implementation very simple.

```
unsigned char s0[16] = {
        3,  8, 15,  1,
       10,  6,  5, 11,
       14, 13,  4,  2,
        7,  0,  9, 12
};
```

Figure 2.5: Serpent S-box S_0 written as array

```
#define S0(x0, x1, x2, x3, x4) ({ \
                        x4  = x3; \
  x3 |= x0;  x0 ^= x4;  x4 ^= x2; \
  x4 = ~x4;  x3 ^= x1;  x1 &= x0; \
  x1 ^= x4;  x2 ^= x0;  x0 ^= x3; \
  x4 |= x0;  x0 ^= x2;  x2 &= x1; \
  x3 ^= x2;  x1 = ~x1;  x2 ^= x4; \
  x1 ^= x2;                       \
})
```

Figure 2.6: S_0 written as logical sequence

In fact the high-level pseudocode, shown in figure 2.3, still applies to this implementation, apart from the two permutations. The linear transformation L, shown in figure 2.4, was already presented in the form, it is needed for this implementation. The major difference between the explanation given above and the implementation presented here, is the multiple application of a S-box in the function $\hat{S}_i$. On a modern processor substituting 32 times 4 bit groups of 128 bit input data would be horrible slow, as the processor is at least capable of processing 32 bit doublewords. That is why the Serpent S-boxes were designed to be able to be applied in a more sophisticated way. The 128 bit input of $\hat{S}_i$ is considered as four 32 bit doublewords. The S-box is then implemented as a sequence of logical operations (instead of 32 table lookups), which can be applied to these four words. With this logical sequence, the CPU implicitly processes the 32 table lookups in parallel. In figure 2.5 you see the Serpent S-box S_0 written as classical array, like you would use it in conjunction with an ordinary table lookup. In figure 2.6 the corresponding logical sequence of S_0 is shown. In this listing S_0 is implemented as macro and gets the four 32 bit doublewords `x0` to `x3` as input and `x4` is a temporary variable. Figure 2.6 was taken from the generic Serpent implementation in the Linux kernel, but both variants are available in the full submission package of the Serpent cipher for the AES competition as well. With these building blocks we have everything, we need to implement Serpent in an efficient way. The key mixing is just a xor of 128 bit and the linear transformation works with 32 bit doublewords as well as the substitution operation. All we have to do is to follow the pseudocode in figure 2.3 without the initial and final permutation, use the S-boxes as logical sequences, like in figure 2.6, and the linear transformation from figure 2.4. If we implement Serpent this way, like it is suggested in the official publication [4], we need no table lookups at all and the only memory operation in an encryption round is in fact the loading of the round key. Decryption has not been covered by now, but to keep things short it can be said that it works very similar, but everything has to be reversed. This is possible because the linear transformation and the eight S-boxes are invertible operations and of course the key mixing is invertible as well, because it is just a xor operation.

There is one more thing, we have not covered yet: The generation of the 33 128 bit subkeys $\hat{K}_0, \ldots, \hat{K}_{32}$, that are needed in every round and after the last round of en- or decryption, also referred to as key schedule. To generate the subkeys, we have to execute the following steps: First the user supplied key is padded to 256 bits and written as eight 32 bit doublewords $w_{-8}, \ldots, w_{-1}$. After that it is expanded to an intermediate key $w_0, \ldots, w_{131}$ by the following affine recurrence:

$$w_i := (w_{i-8} \oplus w_{i-5} \oplus w_{i-3} \oplus w_{i-1} \oplus \phi \oplus i) <<< 11$$

ϕ is the fractional part of the golden ratio $\frac{1}{2}(\sqrt{5}+1)$ or `0x9e3779b9` in hexadecimal. Now the round keys are calculated from the intermediate keys using the eight S-boxes in bitslice mode again, i.e. implemented as logical sequences. Specific S-boxes are applied to four subsequent doublewords of the intermediate key, giving four doublewords of the real key or one 128 bit subkey for the corresponding round. We do not have to reimplement the key schedule, but just the encryption and decryption routines of Serpent, and therefore the detailed process of which S-boxes are applied in which order to parts of the intermediate key will be not explained here. If you already have generated the intermediate key, it is not very complicated, tough, and you might want to have a look at the official publication [4], if you are interested in the details of the key scheduling. Concluding this section, this has been a high-level description of Serpent with some additional informations on how an actual implementation could be done. In section 3.1 our own implementation will be explained in detail.

2.1.4 Twofish

Twofish [53], as the third best rated block cipher in the AES competition, has the basic structure of a *Feistel network*. A Feistel network is a general method of transforming a function F into a permutation. One advantage of a Feistel network compared to a substitution-permutation network is that the function F does not need to be invertible, i.e. bijective. F takes $n/2$ bits of a block, consisting of n bits, and a round key as input and gives $n/2$ bits as output. The whole block of length n is divided into two parts of equal length and processed as follows: In each round the first half is the input of F and the output of F is xor'ed with the second half. After this, the two halves are swapped and processed in the next round exactly the same way. This can be repeated over a designated number of rounds. Two rounds of a Feistel network are called a *cycle*. With this technique we get an invertible cipher, possibly consisting of several rounds, even tough F is not necessarily invertible itself. Twofish is a 16-round Feistel network, i.e. takes 8 cycles, with a bijective function F. It is related to the former block cipher Blowfish [52], developed by Bruce Schneier, who amongst others also designed Twofish, as an alternative to DES. Both algorithms, Blowfish and Twofish, still play an important role in applications used today.

In contrast to Serpent, Twofish uses key-dependent S-boxes. To be more precise it uses four key-dependent 8x8 S-boxes. Additionally to the S-boxes there also exist round keys, which are xor'ed with the intermediate state in each round, and moreover Twofish uses so called *whitening keys*, which are xor'ed with the input block before the first round and with the output block after the last round. This key-whitening technique is used to hide the specific inputs to the first rounds' F function from an attacker and increases the difficulty of keysearch attacks. All these key-dependent components, the four S-boxes and 40 doublewords of subkey material (8 doublewords for key-whitening and 32 doublewords used as subkeys for the 16 rounds), are generated in the key schedule of Twofish. This makes the key schedule rather complex, but for the moment we assume, we have finished the key scheduling, generated all the key-dependent components and have a look at the encryption process. Of course Twofish has a block size of 128 bits and accepts key sizes of 128, 192 and 256 bits, because it is an AES candidate. Figure 2.7 gives a complete overview of the Twofish encryption algorithm. This illustration may look a bit complicated, but in fact it is a good visualisation of the whole process and you can see the Feistel structure of the algorithm. First of all let us explain the symbols: $<<<$ and $>>>$ donate left respective right rotations, $\oplus$ stands for a xor operation of doublewords and $\boxplus$ is the addition of two doublewords modulo 2^{32}. The input block, 128 bits of plaintext, is first xor'ed with the first four doublewords of the whitening key. After that there follow 16 rounds, in which the first half of the block is processed by the function F and afterwards is xor'ed with the second half of the block. There is one difference between this structure and a pure Feistel structure: The two one bit rotations of the doublewords in the second half of the block, one after the xor'ing with the output of F and one before it. As rotations are invertible this is no problem and the advantage of a Feistel structure still applies. Apart from that everything works the usual way. The two halves are swapped and one round is finished. After the last round has been completed, the two halves are swapped again, to undo the last swap, and the ouput is xor'ed with the second four doublewords of the whitening key. The result of this operation becomes the 128 bits of ciphertext.

Now as the macrostructure has been shown, we need to take a closer look on the function F. F takes 64 bit data as input and processes it as two 32 bit doublewords. The first doubleword is directly processed by the function g and the second doubleword is left rotated by 8 bits, i.e. one byte, and after that processed by the function g as well. On the two doublewords, now named a and b, a *Pseudo-Hadamard-Transformation* (PHT) is applied. The PHT is a simple mixing operation and defined as:

$$a' = a + b \mod 2^{32}$$
$$b' = a + 2b \mod 2^{32}$$

As you can see in figure 2.7, this operation can be realized by just two additions of doublewords modulo 2^{32}. Finally onto the two results of the PHT two doublewords, the subkey for this round, are added. Again this is done with an addition of doublewords modulo 2^{32}. So in every round two 32 bit doublewords of subkey material are needed. Now it becomes clear, why there are 40 doublewords generated in the key schedule. In every round we need 2 doublewords, adding up to 32 doublewords in 16 rounds, and 8 doublewords are needed for the key-whitening process at the beginning and end of the encryption routine. With the addition

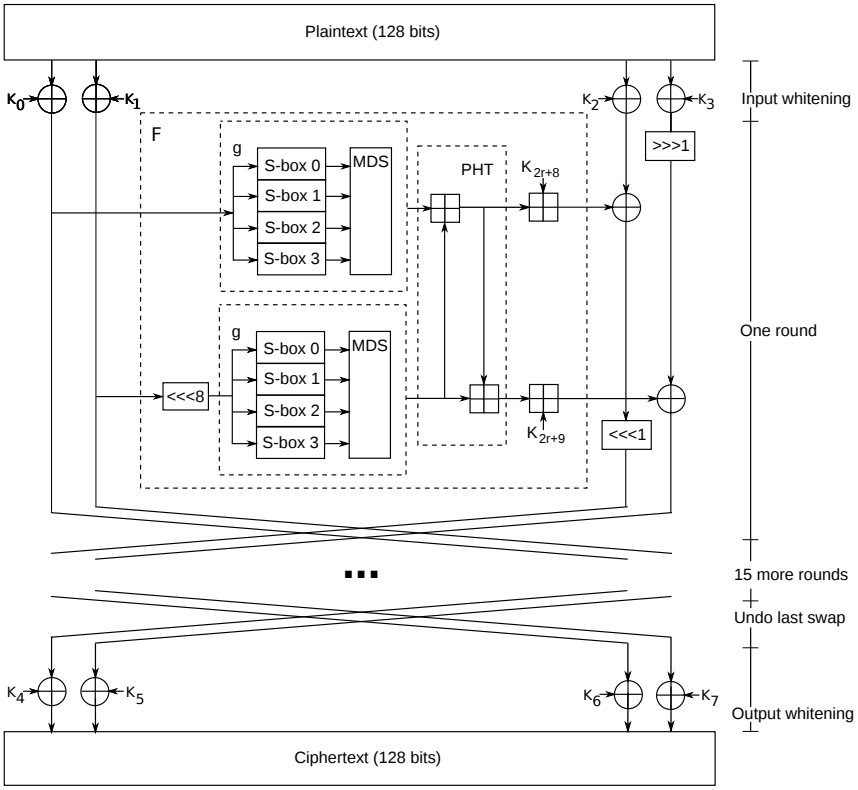

Figure 2.7: Overview of the Twofish encryption algorithm

of the round key the function F is finished.

The last component in the encryption routine, which has to be explained, is the function g. However it is the most complicated component. g takes one 32 bit doubleword as input and outputs one 32 bit doubleword. The doubleword is split into four bytes and every byte is processed by one of the four key-dependent 8x8 S-boxes. The four results are interpreted as a vector of length 4 over GF(2^8), which is multiplied by the 4x4 MDS matrix giving the result of g. MDS stands for *maximum distance separable* and a MDS matrix is a matrix operating over a finite field with certain diffusion properties. These diffusion properties are helpful, to ensure that small changes in the plaintext lead to huge changes in the ciphertext. This connection has already been pointed out in section 2.1.3 about Serpent. The MDS operation looks like this:

$$\begin{pmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} 01 & EF & 5B & 5B \\ 5B & EF & EF & 01 \\ EF & 5B & 01 & EF \\ EF & 01 & EF & 5B \end{pmatrix} \cdot \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} \tag{2.1}$$

The matrix you see is the Twofish specific MDS matrix from the official publication [53]. The values $y_0, \ldots, y_3$ are the four results of the application of the S-boxes 0 to 3 and $z_0, \ldots, z_3$, as result of the MDS operation, lead to the result of the function g by being interpreted as one doubleword again. In fact the operation (2.1) is an usual matrix-vector multiplication, but the computations are done in the field GF(2^8).

GF(2^8) is the - except from isomorphisms - unambiguous finite field with 256 elements and can be represented as GF$(2)[x]/v(x)$, where $v(x)$ is an irreducible polynomial of degree 8 over GF(2). Maybe you know GF(2) better as $\mathbb{Z}/2\mathbb{Z} = \mathbb{Z}_2$, which is the finite field with only two elements. In this field addition is the usual addition from $\mathbb{Z}$ modulo 2, which is in fact just the xor operation, and multiplication is trivial as there only exist the elements 0 and 1. Every element of GF(2^8) now can be represented as polynomial with at most degree 7 and coefficients out of GF(2), because GF(2^8) = GF$(2)[x]/v(x)$. Addition in GF(2^8) is consequently equivalent to the addition of two polynomials with coefficients out of GF(2), that is to say

the coefficients just have to be xor'ed. Addition in GF(2^8) is an efficient operation, but multiplication is non-trivial. You have to do the multiplication, like you would multiply two ordinary polynomials, but every operation is done modulo $v(x)$. This means that if you multiply, and any term with degree 8 or higher occurs, you have to remove this term by adding the terms of degree 7 or lower taken from $v(x)$ to the result. This works because $v(x)$ equals 0 in GF(2^8) and subtraction is the same operation as addition in GF(2). Elements of GF(2^8) are typically represented as exactly one byte, containing the coefficients for the corresponding polynomial of at most degree 7. This makes addition even easier, because you just have to xor two bytes, to realize an addition in GF(2^8).

After this short excursion to finite field arithmetic, let us come back to our MDS operation. The irreducible polynomial chosen for Twofish is $v(x) = x^8 + x^6 + x^5 + x^3 + 1$. All the elements listed in the matrix in (2.1) are elements of GF(2^8) = GF(2)$[x]/v(x)$ with this specific $v(x)$. The first component z_0 of the result vector is calculated like this:

$$z_0 = (01 \odot y_0) \oplus (\text{EF} \odot y_1) \oplus (\text{5B} \odot y_2) \oplus (\text{5B} \odot y_3) \tag{2.2}$$

This entire calculation is done in GF(2^8), at which $\oplus$ stands for addition (a simple byte xor) and $\odot$ for the complex multiplication in GF(2^8). The other components z_1, z_2, z_3 are calculated exactly the same way. With this operation the function g is finished and the entire encryption process has been described. The decryption routine has not been covered, but since Twofish has Feistel structure, it is not necessary to provide reversed S-boxes or even a reversed function F for the cipher. All we have to do is to apply the key-whitening in the reversed order and use the same structure as in figure 2.7. F depends on the round keys and so the 16 applications of F have to be applied in reverse order, too. The function F itself, however, does not change in this entire process. The one non-Feistel element in this cipher, the two one bit rotations, have to be reversed as well, because they are not part of the function F. This means, that if the encryption has been implemented, very few changes have to be made to implement the decryption routine. The last question is how to get the 40 doublewords $K_0, \ldots, K_{39}$ of key material for the encryption and decryption routine. Twofish has a really complex key schedule, which generates the 40 doublewords (8 doublewords whitening keys and 32 doublewords as round keys) and the four key-dependent 8x8 S-boxes from the user supplied key. It would probably take several pages to describe the key schedule in detail and as we do not need to implement it, we will not cover it here. If you are interested in the details, you should take a look at the Twofish Paper [53]. In all cases the user supplied key is padded to full 256 bits and the number of rounds is independent of the size of the supplied key.

There is one major optimization, which can be made, when implementing Twofish. Implementing the encryption routine the way it was introduced here, would result in a bottleneck, because of the function g. As already mentioned the MDS operation is a complex and expensive operation, but if you have a close look at figure 2.7, you will see that the MDS operation is applied to the four bytes *after* the processing with the four S-boxes. Moreover the MDS operation is just a matrix multiplication and therefore a linear operation over GF(2^8). This means, it is possible to combine the application of the S-boxes and the MDS operation, thus the entire function g, to four table lookups and three xor operations. However this technique needs more space, because instead of four 8x8 S-boxes, i.e. 1024 bytes, four 8x32 tables, i.e. 4096 bytes, are needed. This trick works by precalculating the MDS operation for every byte of the four S-boxes. Every byte of the first S-box is multiplied over GF(2^8) with every element of the first column of the MDS matrix. The resulting 4 bytes are saved as 32 bit doubleword in a new 8x32 lookup table at the same position from where the byte was taken out of the first S-box. The same procedure is carried out with the second S-box and the second column of the MDS matrix, and so on. After the precalculating, we now have four 8x32 lookup tables, which we can store instead of the four key-dependent S-boxes. The S-boxes are no longer needed and may be discarded. So instead of implementing g like described before, we now take the four input bytes and do four table lookups into our precalculated tables. The four resulting doublewords are xor'ed and this is the result of the function g. If you have a look at the calculation (2.2), where the first result byte was calculated the ordinary way, you will see the three xor operations and the four multiplications, which now can be replaced by our table lookups, because the lowest byte of the doubleword in a specific table corresponds to a specific term in (2.2).

There is one more small modification, which can be made. The 8 bit left rotation before one of the two applications of g can be saved, if we provide two functions g_1 and g_2, which differ in the order in which

byte of their input they use for which lookup table. This will increase code size slightly, but we save at least one instruction in every round. With this two tricks, we have significantly simplified the encryption routine, because we do not need multiplications over a finite field anymore. The only operations that are needed to implement the encryption routine are xor, rotations, additions and table lookups. In section 3.2 we will use exactly this optimizations to implement the Twofish encryption and decryption routine as fast as possible.

2.1.5 Blowfish

After having a look at the AES finalist Twofish, it is consequential to have a look at the predecessor Blowfish [52], too. Blowfish is not an AES candidate, but nevertheless it is still widely used in today's applications. Blowfish has the structure of a pure Feistel network and iterates an encryption round function 16 times. There are no exceptions, like the two one bit rotations in the Twofish encryption process. The block size of Blowfish is just 64 bits and the key size can vary between 32 and 448 bits. There are always 16 rounds, independent of the chosen key size. In the encryption routine Blowfish uses 18 32 bit subkeys and four key-dependent 8x32 S-boxes. The 18 doublewords of subkey material and the key-dependent S-boxes are generated during the key scheduling process of Blowfish. The details of the key scheduling algorithm will not be covered here, because we do not need to reimplement this part. Basically it works this way: The encryption routine itself is used to generate the subkey material and the S-boxes. First of all the 18 doublewords and the S-boxes are initialized with the hexadecimal digits of π. After that the subkeys are xor'ed with the user supplied key and an all-zero block is encrypted. The result of this encryption process replaces the first two doublewords of subkey material. The encryption routine now gets the result as input to replace the next two doublewords. This process is repeated until all values of the subkey material and the S-boxes have been replaced. In total there are 521 iterations required to finish the key schedule, making this a rather expensive operation.

By now we assume, that the key schedule has been finished successfully and the 18 doublewords of subkey material and the four key-dependent 8x32 S-boxes are available. The encryption routine gets a 64 bit block as input and outputs a 64 bit block. First of all the input block is divided into two 32 bit doublewords. Now there follow sixteen rounds, that operate on the two doublewords. In every round the left doubleword is xor'ed with the corresponding subkey doubleword and afterwards processed by the function F. The result of F is xor'ed with the right doubleword and then the two doublewords are swapped. With this operation one round is finished and the next one follows. After the sixteen rounds are finished, the left and the right doubleword are swapped again, to undo the swap of the last round, the right doubleword is xor'ed with the 17th subkey doubleword and the left doubleword is xor'ed with the 18th subkey doubleword. The recombination of the left and the right doubleword results in the output block of the blowfish encryption routine.

As you can see, the encryption routine of Blowfish is really simple to describe and by far simpler than the one for Twofish. Moreover the Feistel structure of the algorithm is clearly visible. The last thing we have to explain is the function F. F gets a doubleword as input and divides it into four bytes. For every byte a lookup into one of the four key-dependent 8x32 S-boxes is made in order, resulting in four doublewords. Now the first two doublewords are added modulo 2^{32} and the result is xor'ed with the third doubleword. The result of this operation is added to the fourth doubleword modulo 2^{32} leading to the result of F.

Now all components of the Blowfish encryption routine have been described. Basically one round in this routine consists of a xor operation with the round key and four table lookups followed by two additions and one xor operation. Although the key schedule is rather complex, the encryption routine is very simple, at least if the hardware supports operations on 32 bit doublewords. There are 4168 bytes required to store the Blowfish context among subsequent encryption calls. The decryption routine works equivalent to the encryption routine, because of the Feistel structure, and just the subkeys have to be supplied in reverse order.

2.1.6 Cast-128

Cast-128 [1], which is also called Cast5, is another block cipher, which is similar to Blowfish. It is still the default cipher in GPG according to the manpage. Cast-128 is available worldwide on a royalty-free basis and therefore it is also available in the Linux kernel. As Blowfish, Cast-128 is a Feistel network with 64 bit block size. The key may have sizes between 40 and 128 bits. Cast-128 consists of 12 or 16 rounds, depending on the size of the key. For key sizes shorter than or equal to 80 bits only 12 rounds are used and for key sizes longer than 80 bits the 16 round version is used. The two versions do not differ in anything else, but the number of the rounds and therefore we will focus on the 16 round version in this explanation. In the encryption routine Cast-128 makes use of 16 doublewords of subkey material, called masking keys, and 16 bytes of rotation values. The latter ones are used for key-dependent rotations, a new feature, which has not occurred in the ciphers that have been described by now. Furthermore four 8x32 S-boxes are needed in the encryption routine, but they are not key-dependent, but fixed, regardless of the user supplied key. The 16 masking keys and the 16 rotation values are generated during the key scheduling algorithm of Cast-128, which will not be described here. However the key scheduling is not very complicated. It uses another four S-boxes in addition to the four S-boxes needed in the encryption routine. The detailed key schedule should be looked up from the publication [1].

The encryption routine processes a 64 bit block. The block is divided into two 32 bit doublewords and then sixteen respectively twelve rounds are performed. In each round the right doubleword is processed by a round specific function and the result is xor'ed with the left doubleword. After that the two doublewords are swapped and the next round begins. When all rounds are finished the two doublewords are swapped again, to undo the swap of the last round, and the combination of the two doublewords is the resulting 64 bit output block of the encryption routine. This represents the structure of a typical Feistel network.

$$
\begin{aligned}
f_1(D,i): \quad I_0, I_1, I_2, I_3 &= ((Km_i \boxplus D) <<< Kr_i) \\
f_1(D,i) &= ((S_1(I_0) \oplus S_2(I_1)) \boxminus S_3(I_2)) \boxplus S_4(I_3) \\
f_2(D,i): \quad I_0, I_1, I_2, I_3 &= ((Km_i \oplus D) <<< Kr_i) \\
f_2(D,i) &= ((S_1(I_0) \boxminus S_2(I_1)) \boxplus S_3(I_2)) \oplus S_4(I_3) \\
f_3(D,i): \quad I_0, I_1, I_2, I_3 &= ((Km_i \boxminus D) <<< Kr_i) \\
f_3(D,i) &= ((S_1(I_0) \boxplus S_2(I_1)) \oplus S_3(I_2)) \boxminus S_4(I_3)
\end{aligned}
$$

Figure 2.8: Round-dependent functions of CAST-128

The round specific functions are listed in figure 2.8. There are three different functions, which differ just in the operations carried out on the doublewords, but the basic structure of all three functions is the same. In this figure $\oplus$ denotes xor, $\boxplus$ denotes addition modulo 2^{32}, $\boxminus$ stands for a subtraction modulo 2^{32} and $<<<$ denotes logical left rotation by the given number of bits. The functions are used in the order listed in this figure, meaning that in the first round the function f_1 is used, in the second round f_2 and in the third round f_3. After that, the function f_1 is used again in the fourth round and so on. All functions get the right doubleword D from the current encryption round as input and use the current round number i. First a specific operation with the corresponding masking key Km_i of the round is carried out and afterwards the result is left rotated by the rotation value Kr_i of that round. The result of this rotation is interpreted as four bytes, where I_0 is the most significant byte and I_3 is the least significant byte. For each byte a lookup into the fixed 8x32 S-boxes $S_1, \ldots, S_4$ is performed and the results of this lookups are combined according to the operations listed in figure 2.8.

With the round specific functions the whole encryption routine has been explained and we do not need to explain the decryption routine in detail, because of the Feistel structure. Of course the masking keys and the key-dependent rotation values have to be applied in reverse order, as well as the functions f_1, f_2 and f_3, but the rest of the decryption routine works exactly like the encryption part. To summarize things, the Cast-128 cipher uses basic operations on doublewords and table lookups. The rotations with key-dependent rotation values are not basic operations, but we will show in section 3.4, how we can implement them with SIMD instructions.

2.1.7 Cast-256

From Cast-128 it is no big step to the successor Cast-256 [2], also called Cast6, and therefore this will be the last cipher, which will be covered in this thesis. Cast-256 has been submitted to the AES Competition, but was not among the five finalists mentioned above. Consequently it has a block size of 128 bits and accepts key sizes of 128, 160, 192, 224 and 256 bits. The key sizes 128, 192 and 256 bits are necessary for an AES candidate. Cast-256 works with 48 rounds, also referred to as twelve quad-rounds, and the number of rounds is independent of the key size. The four fixed 8x32 S-boxes and the three functions f_1, f_2 and f_3 are reused from the Cast-128 algorithm. The encryption routine of Cast-256 uses 48 masking keys and 48 rotation values, which have the same meaning, as they had with Cast-128. They are generated in the Cast-256 key schedule, which will not be described here.

$$
\begin{aligned}
Q(I,i): \quad & A,B,C,D = I \\
& C = C \oplus f_1(D, 4i+0) \\
& B = B \oplus f_2(C, 4i+1) \\
& A = A \oplus f_3(B, 4i+2) \\
& D = D \oplus f_1(A, 4i+3) \\
& Q(I,i) = A,B,C,D
\end{aligned}
\qquad
\begin{aligned}
\overline{Q}(I,i): \quad & A,B,C,D = I \\
& D = D \oplus f_1(A, 4i+3) \\
& A = A \oplus f_3(B, 4i+2) \\
& B = B \oplus f_2(C, 4i+1) \\
& C = C \oplus f_1(D, 4i+0) \\
& \overline{Q}(I,i) = A,B,C,D
\end{aligned}
$$

Figure 2.9: Q and $\overline{Q}$ of CAST-256

Let us assume the 48 doublewords of masking keys and the 48 bytes of rotation values have been properly generated. The encryption routine now executes six forward quad-rounds Q and afterwards six reverse quad-rounds $\overline{Q}$, which both are shown in figure 2.9. The input of the first quad-round is the input of the encryption routine and after that, the output of the previous quad-round is taken as input for the next quad-round. In the end the output of the last reverse quad-round results in the output of the encryption routine. The reverse quad-round $\overline{Q}$ is, as the name suggests, the reverse operation of the quad-round Q. As you can see in figure 2.9 the operations are just listed in the reverse order and therefore it is sufficient to explain just the quad-round Q in detail. Q gets 128 bits of data as input and interprets them as four doublewords A, B, C and D. Now the functions f_1, f_2 and f_3 from the Cast-128 algorithm are called with the arguments shown in figure 2.9. The round argument goes from 0 to 47 and that is why all the 48 masking keys and rotation values are used by the subsequent invocations of the different functions. The whole encryption routine is described by the two quad-rounds Q and $\overline{Q}$, because both variants are just called six times. The decryption routine calls the six quad-rounds and the six reverse quad-rounds in the same order as the encryption routine, but supplies the quad-round numbers in the reverse order. So instead of supplying the values 0 to 11 the values 11 to 0 are supplied. This works, because the instructions in both variants are listed in the reverse order to each other. As you might have noticed many parts of the Cast-128 cipher can be reused in the Cast-256 implementation.

2.2 Advanced Vector Extensions

Intel's *Advanced Vector Extensions* (AVX) are a set of instructions for *Single Instruction Multiple Data* (SIMD) operations. A SIMD operation is a simple operation, like addition, multiplication or a boolean operation, which is applied on multiple, equal sized and packed data in parallel. Only one instruction is needed to process the data, whereas without SIMD multiple instructions would be used. Consequently instructions can be saved by this technique, provided that an algorithm is suitable for being parallelized this way. Figure 2.10 shows an illustration of how SIMD operations work compared to scalar operations.

AVX was developed for the continued need of vector performance in scientific applications, visual processing, recognition, gaming, physics, cryptography and other areas of applications [34]. Especially the vector performance in the field of cryptography is interesting for this thesis. The first processors supporting AVX were the Intel Sandy Bridge processors shipped in Q1, 2011 [36]. AMD provides AVX support with the Bulldozer processors shipped in Q3, 2011. At first AMD was working on its own SIMD instruction set,

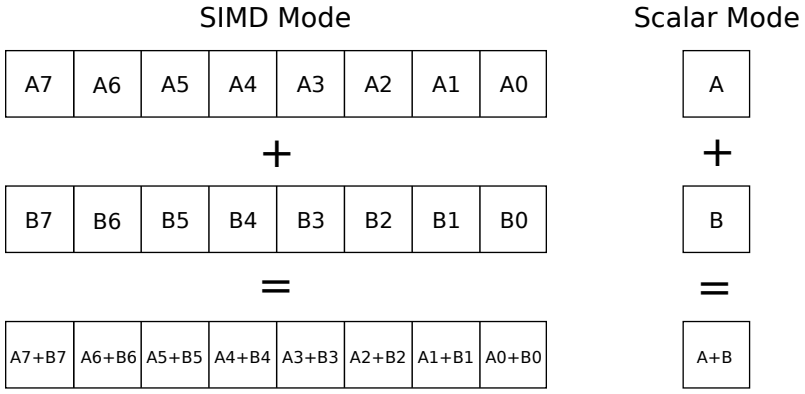

Figure 2.10: SIMD vs. scalar operations [36]

called SSE5, but in May 2009 they decided to replace SSE5 with three new extensions XOP, CVT16 and FMA4 and to stay compatible with Intel's AVX [26]. The GNU Compiler Collection (GCC) supports AVX at least since version 4.6 [27] and the Linux Kernel since version 2.6.30 [55]. Explicit operating system support is required, because the registers used by AVX have to be properly saved and restored across context switches.

The new instructions extend the previous SIMD instructions by adding some interesting new features. The registers have been expanded to 256 bits and a nondestructive, three-operand syntax has been added. The details will be discussed in the following sections. To detect the availability and support of AVX on the hardware, it is possible to use the `cpuid` instruction. If AVX is supported by the hardware, the Bits 27 and 28 in the `%ecx` register are set after executing the instruction. The operating system support has to be checked as well. Therefore the `xgetbv` instruction is needed and the Bits 1 and 2 (XMM and YMM state support) in the feature-enabled mask must be set. The whole process is shown in figure 2.11.

```
avx_supported:
        mov $1, %eax
        cpuid
        and $0x018000000, %ecx
        cmp $0x018000000, %ecx ; check OSXSAVE, AVX feature flag
        jne .Lnot_supported
        xor %ecx, %ecx         ; specify 0 for XFEATURE_ENABLED_MASK register
        xgetbv                 ; result in %edx:%eax
        and $0x06, %eax
        cmp $0x06, %eax        ; check OS has enabled XMM and YMM state support
        jne .Lnot_supported
        mov $1, %eax           ; return value true
        jmp .Ldone
 .Lnot_supported:
        xor %eax, %eax         ; return value false
 .Ldone:
        ret
```

Figure 2.11: Checking for AVX support on hardware and operating system [34]

2.2.1 Registers

AVX introduces support for 256 bit wide SIMD registers. The 8 registers YMM0 to YMM7 are available in operating modes, that are 32 bit or less and in 64 bit mode the 16 registers YMM0 to YMM15 can be used. As AVX is an extension to the previous SIMD offerings, like the Streaming SIMD Extensions (SSE), the lower 128 bits of the YMM registers correspond to the respective 128 bit XMM registers, already known from SSE. This makes it possible to use AVX and SSE instructions together in the same application, tough it is not recommended, because AVX to SSE transitions have an impact on performance. In cases where

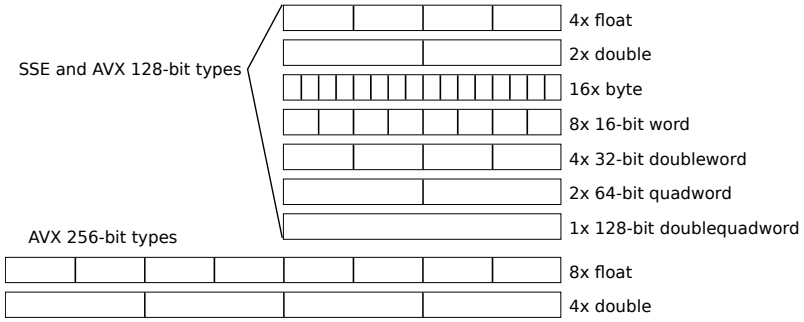

Figure 2.12: AVX and SSE data types [36]

the source code can be modified, this is not a problem, because for every SSE/SSE2 instruction there exists a new corresponding or equivalent AVX instruction. However, there might be cases, in which the code cannot be modified, for example if closed source third party libraries have to be used. In this case, mixing AVX instructions with legacy SSE instructions is not avoidable and a so called *transition penalty* is caused. The impact on performance can be minimized by using the `vzeroupper` instruction after the last AVX and before the first SSE instruction. This instruction clears the 128 upper bits of all YMM registers, which cannot be used by the legacy SSE instructions. The Intel Software Development Emulator [10] supports checking for transition penalties and shows, where `vzeroupper` instructions have to be inserted. AVX instructions, which operate on a specific XMM register, which corresponds to the lower 128 bits of the respective YMM register, always set the upper 128 bits of that YMM register to zero. This means, it is not possible to operate on just the lower part of a register using XMM operands and keep data stored in the upper part at the same time.

The data stored in the YMM and XMM registers can be interpreted in many different ways, depending on the instruction used to operate on the registers. Figure 2.12 gives an overview of the possibilities, that come with AVX. As you can see, the 256 bit wide YMM registers can only be used with 32 bit single precision and 64 bit double precision floating point types. This is a serious drawback, because most of the operations used in cryptographic algorithms are in fact integer operations. The 128 bit wide XMM registers can be used with single and double precision floating point types as well as with all integer types ranging from byte to doublequadword. The 128 bit wide integer types were already introduced with SSE2. The interpretation of the data is selected with the suffix of the operating instruction. For example `vaddps` is used to add packed single precision values whereas `vaddpd` is used to add packed double precision values. The interpretation of the data can be changed with every instruction, allowing complex modifications. Switching between integer and floating point types, however, has also an impact on the performance and as stated above switching between 256 bit wide YMM registers using floating point types and 128 bit wide XMM registers using integer types is not a good idea, because the upper 128 bits would get lost. This means that the 256 bit wide registers cannot be used efficiently in cases, where mostly integer operations are needed. The Intel 64 and IA-32 Architectures Optimization Reference Manual confirms this and states [32]:

> In case the code is mostly integer, convert the code from 128-bit SSE to 128 bit AVX instructions and gain from the Non destructive Source (NDS) feature.

This leads to the conclusion, that cryptographic algorithms cannot gain a speedup close to 2 from using AVX compared to SSE2, because the registers which are used have the same width as with SSE2. Nevertheless there are some improvements, which can be made. The NDS feature will be explained in the next section. Finally there is already a new extension, called AVX2, announced, which will be described in section 2.2.3.

2.2.2 Instruction Set

AVX not only introduces support for wider registers, but also a new instruction set. Not every detail will be covered in this section, but some instructions, that are interesting for this thesis and the major differences between SSE2 and AVX will be explained. The really useful and detailed description can be found in

the Intel Advanced Vector Extensions Programming Reference [34], which already contains information about upcoming extensions, like AVX2, or in the Intel 64 and IA-32 Architectures Software Developer's Manual [33], which is a good reference for AVX and instructions operating on general purpose registers.

Intel uses a new set of code prefixes, called *VEX coding scheme*, which makes it possible to extend the op-code map, to have space for new instructions. This instruction codes are allowed to have up to five operands compared to the original scheme used with SSE2, which only allowed two operands in the most cases. The most obvious change, which you might notice, when looking at AVX source code, is the already mentioned non-destructive three operand syntax, which is made possible with help of the VEX prefix. The VEX prefix itself is not covered here, because how the instructions are actually encoded is only interesting to people, who work on compilers, but the consequences are interesting for speeding up cryptographic algorithms. A third non-destructive source operand has been added to most instructions and so the destination register can be different from both source registers. This means that operations like $a = b + c$ can be encoded in one single instruction, while leaving b and c untouched. With SSE2, only operations of the form $a = a + b$ could have been encoded in one instruction and so encoding $a = b + c$ would have needed an additional instruction just to save the value of one of the source operands. The two-operand syntax previously expressed as

```
paddd %xmm1, %xmm2
```

can now be expressed in three-operand syntax as

```
vpaddd %xmm1, %xmm2, %xmm3
```

Of course there are cases where you do not need a third operand, because you do not need the value of both source operands in the following code. It is not a problem to use the same register as source and destination operand and so you can just write `vpaddd %xmm1, %xmm2, %xmm2` in that case. Furthermore it is perfectly legal, to set a memory operand for one of the source operands. For example this would be valid code: `vpaddd (%rsi), %xmm1, %xmm2`. The destination operand, however, has to be a register for most instructions aside from movement instructions. With this new syntax we are able to increase performance of existing code slightly, just by saving data movement instructions.

Now it is time to have a look at the instructions AVX provides. There are too many instructions, to list them all and so we concentrate on instructions, that are relevant for cryptographic algorithms. Many of the floating point operations are not relevant at all. For example there exist instructions to round floating point numbers (`vroundpd`, `vproundps`) or even to extract the square root and the reciprocal square root (`vsqrtps`, `vsqrtpd`, `vrsqrtps`, `vrsqrtpd`). All the floating point instructions, including this rather complex instructions, could operate on the 256 bit YMM registers, whereas this is not applicable on integer instructions in general. Most instructions come with a whole bunch of different suffixes to specify, which interpretation of the data in the registers used as operands should be chosen. For the floating point instructions the suffixes are `ps` and `pd` for packed single and double precision floating point types and for integer instructions the suffixes are `b`, `w`, `d`, `q` and `dq` for packed 8 bit byte, 16 bit word, 32 bit doubleword, 64 bit quadword and 128 bit doublequadword integer types. Not every instruction can be used with any suffix and to keep things simple the instructions will be listed with the `d` suffix, if there are more suffixes available. This is a good choice, because in cryptographic algorithms we often have to deal with packed data of doubleword size. Floating point instructions will only be listed, if there is no equivalent integer instruction available and if an instruction is able to operate on the 256 bit YMM registers, too, this will be denoted by a star (*) in the following listing. Here is an overview of the instructions, that will be discussed in this section:

- Movement Operations
  - Between AVX registers or memory: `vmovdqa`*, `vmovdqu`*
  - AVX registers to general purpose registers or memory: `vmovd`, `vpextrd`, `vpinsrd`
  - AVX registers to memory: `vbroadcastss`*
- Arithmetic Operations: `vpaddd`, `vpsubd`
- Logical Operations: `vpand`, `vpandn`, `vpor`, `vpxor`

- Shift Operations: `vpslld`, `vpsrld`,`vpslldq`, `vpsrldq`
- Shuffle and Pack Operations: `vpshufd`, `vpunpckhdq`, `vpunpckldq`

We begin with the movement operations. To fill the AVX registers from memory or copy data between them you can use either `vmovdqa` or `vmovdqu`. Both instructions take only two operands, one source and one destination operand, and either of them may be a memory operand. They copy 128 or 256 bit of data from source to destination, depending whether the operands are XMM or YMM registers. The `vmovdqa` instruction requires the data to be 16 or 32 byte aligned in memory and is faster, while the `vmovdqu` instruction, in contrast, requires no alignment, but might be slower. It is always advisable to use the `vmovdqa` instruction, if you are able to ensure that the data is aligned. In case of copying data between registers it does not matter, what instruction you use, but usually the `vmovdqa` variant is used, too. With `vmovd` it is possible to transfer data between general purpose registers or memory and AVX registers. One operand is a SIMD register and the other operand is either a memory operand or a general purpose register. The lowest doubleword from the AVX register is used for the operation, and if the AVX register is the destination all the upper bits, apart from the lowest doubleword, are set to zero. Sometimes this is an unwanted side effect and there the other two instructions come into play. `vpextrd` and `vpinsrd` allow to extract respectively insert a doubleword from an AVX register to a general purpose register or to memory. The doubleword can be chosen by an additional immediate operand, which is set to the position of the doubleword, counting from low to high values. `vpextrd` takes three operands (source, destination and the immediate value) and `vpinsrd` even takes a fourth operand, which decides where to get the other doublewords from, which are not loaded from memory or a general purpose register. The last movement operation discussed here is `vbroadcastss`. As the name suggests it takes a single precision floating point value from memory and broadcasts it to all 4 respectively 8 locations of an AVX register. This is a new instruction which was not available in the SSE variants. With SSE you needed to emulate this instruction with a `movd` and a following `pshufd` to broadcast the doubleword value from the lowest doubleword to the whole register. In AVX there is no integer version of this instruction available and so we have to stick with the floating point variant. The integer instruction will be introduced with AVX2 and discussed in section 2.2.3.

The next type of operations, that are needed in cryptographic algorithms, are arithmetic instructions. The name already suggests what the instructions are doing. `vpaddd` and `vpsubd` do packed addition and subtraction on their operands. As all usual AVX instructions they take three operands and thus can be used with a non-destructive source. Of course there are many more arithmetic instructions available, but they are not listed here, because on one side they are not required for our use cases and on the other side their usage should be quite clear.

By far the most frequently operations used in cryptography are logical operations and fortunately the instructions listed above are very easy to describe. `vpand`, `vpor` and `vpxor` perform a bitwise *and*, *or* and *xor* on their operands. `vpandn` performs a bitwise *and not*, meaning one operand is negated, or to be more precise the bitwise complement is taken, and after that the bitwise *and* with the second operand is performed. You can image it like this: $a = \neg b \wedge c$. There are no different suffixes for these instructions as a packed bitwise boolean operation would be independent of the size of the packed operands. Unfortunately these instructions only operate on 128 bit wide registers. There also exist floating point instructions, for example `vxorps`, `vxorpd`, that in fact do exactly the same thing and are able to take 256 bit wide operands. However it has shown, that they are way slower than the integer instructions and as already pointed out, the Intel 64 and IA-32 Architectures Optimization Reference Manual [32] states, that one should better stick with the integer instructions. But even if the performance would be better, we still had the problem that there are no equivalent floating point instructions for doing shifts and so we had to fall back to 128 bits for these instructions anyway.

The next family of instructions are shift operations. There is a big difference between the `d` and `dq` variants listed above. `vpslld` and `vpsrld` do a packed bitwise logical left or right shift on the doublewords specified in their operands, whereas `vpslldq` and `vpsrldq` do a bytewise logical left or right shift on their operands as a whole. This means the second variant is no packed operation compared to the first one. All four instructions take a destination and a source operand and the shift value may be an immediate operand or an AVX register. Intel's AVX instruction set provides no single instruction for packed rotations

and so a rotation has to be emulated by a packed bitwise logical left shift and right shift and one logical bitwise or to link the result. Consequently a packed bitwise rotation takes three instructions, instead of one, as we could do it in general purpose registers. In the most cases it is still the better choice to do it with the three instructions, because otherwise we would need to extract the data from AVX to general purpose registers and we probably could not store so much data as well. AMD provides an own extension, called XOP, which is in fact a bit more powerful than AVX and has support for packed bitwise rotations of doublewords [3]. The instruction is called `vprotd` and is capable of doing packed bitwise left and right rotations. This would probably give us some performance boost, because we could replace three instructions by just one, but as this extension is AMD specific and we want to write code, that at least is portable on newer x86_64 systems, we do not exploit this feature.

The last type of operations are shuffle and pack operations. The `vpshufd` instruction is a rather powerful instruction allowing the permutation of doublewords in the source operand to be stored in the destination operand. The mode of shuffling is specified with a third immediate operand, where for every of the four possible positions in the destination operand two bits in the immediate operand define, which doubleword from the source operand will be taken. Duplicates are possible and very useful in some cases. A broadcast operation can be emulated by supplying four times the same value for all four 2 bit pairs of the immediate operand. For example the lowest doubleword can be simply broadcasted with this instruction:

```
vpshufd $0, %xmm0, %xmm0
```

Other variants of this instruction take a third AVX register, instead of an immediate operand. This has the advantage of being adjustable at runtime, but on the other hand, if the value is known statically at compile time, a register has to be wasted. Another interesting use case is to change the endianness of packed doublewords. Therefore the `vpshufb` instruction is required as endianness differs on byte granularity. This instruction takes the mode of shuffling as a register operand, and so the mask has be loaded into a free register first. Figure 2.13 shows the full example. The `vpunpckhdq` and `vpunpckldq` instructions both take three operands and interleave the high-order or low-order doublewords from the first and second source into the destination register. This is useful when you want to process data in parallel, but the data is not prepared in the order, you would like to process it. To save instructions and especially avoid using too much memory operations, it is possible to load the data linear from memory and then do some interleaving, to get it in the order, you need it for the processing. For example a matrix transposition can be done this way. You will see a good example in section 3.1.1, when we are using these instructions to restructure the data, we get as input for the parallel block ciphers.

```
.align 16
.Lbswap_mask:
  .byte 3, 2, 1, 0, 7, 6, 5, 4, 11, 10, 9, 8, 15, 14, 13, 12

  vmovdqa .Lbswap_mask, %xmm15
  vpshufb %xmm15, %xmm0, %xmm0
```

Figure 2.13: Changing the endianness of packed doublewords with AVX

This were the most important instructions or at least the ones, we are using in this thesis. Some instructions, which are new with AVX, but have not been covered yet, are the `vinsertf128` and `vextractf128` instructions, which are similar to the `vpinsrd` and `vpextrd` instructions, but operate with 128 bit wide floating point values. Consequently they only can be used on the YMM registers, because inserting a 128 bit wide value into a 128 bit wide register is in fact just a movement operation. Another interesting operation is the `vmaskmovps` instruction. It can conditionally write any number of single precision floating point elements from an AVX register to memory, while leaving the remaining elements in memory untouched. The `vperm2f128` instruction does shuffling with 128 bit wide floating point elements on the YMM registers. It is a so called *cross lane* operation, because it can transfer data between the lower and upper 128 bit of a YMM register. More explanation about the lane concept follows in the next section. Last but not least new instructions are also the already mentioned `vzeroupper` instruction, which clears the upper half of the YMM registers, and the `vzeroall` instruction, which simple sets all YMM registers to zero and tags them as unused.

2.2.3 AVX2

In the last section you may have noticed, that there are still some drawbacks, while using AVX, especially regarding the integer operations, which are important to us. AVX has support for 256 bit floating point operands, but is lacking the support for 256 bit integer operands. To equalize this diversity, Intel announced the new extension AVX2, which comes with full support for the 256 bit integer operations and some more interesting new instructions. The first processor generation, which will fully support AVX2, is the Haswell microarchitecture planned to be shipped for 2013 [11]. The specification of the new instruction set, however, is available by now and the Intel Advanced Vector Extensions Programming Reference [34] has all the information which is needed to create applications, that exploit AVX2. The Gnu Compiler Collection (GCC) supports AVX2 since version 4.7 [28] and there are binary packages available, that it is not necessary to build GCC 4.7 yourself. Binary packages of GCC 4.7 are for example included in the upcoming release *Wheezy* of the Debian Distribution, which has currently testing state [12]. This makes it very easy to build applications using some AVX2 elements. You just have to use the new compiler version and everything works out of the box.

Testing your application is not that easy, because at the moment there exists no real hardware, which is capable of running AVX2 code. Fortunately Intel provides a Software Development Emulator [10], which is qualified for emulating the new instructions, introduced with the Haswell architecture. It can emulate SSE4, the AES instructions, AVX and AVX2. Of course it is not possible to make realistic benchmarks with this emulator, but at least it gives us the chance to verify the correctness of our implementations. The Software Development Emulator, now referred to as SDE, does not emulate the whole application, but instead does some sort of static analysis of the binary and decides for each instruction, whether the instruction should be executed directly on the machine or be emulated with the SDE. In the latter case the SDE skips over the instruction and jumps to the appropriate emulation routine. To decide whether to emulate the instruction or not, the `cpuid` instruction is executed on the machine to determinate, which features the machine actually supports. From the emulated applications point of view, the `cpuid` instruction is modified by the SDE in such a way, that for the application it looks like all features exist on the real machine. Supposed the SDE is properly installed, it is enough to use the following line to run your application within the emulator:

```
path-to-kit/sde -- path-to-yourapp/yourapp
```

Everything before the two dashes is considered as argument for the SDE, whereas everything after them is directly passed to the emulated application. It is no problem to test an application, which uses `stdin` and `stdout` with this setup, because the SDE redirects everything and acts transparent. This allows using pipes and even makes it possible to run a whole shell like `bash` within the emulator. Verifying the correctness of an algorithm that uses AVX2 with the SDE is no big deal, exact time measurements, however, are not possible and testing code, which runs only in kernelspace is not feasible either. Nevertheless at least for userspace issues the SDE is a helpful tool and besides just running code is has a few additional features. One simple, but effective, feature is counting instructions. It gives a first hint, which algorithm is the better one, or how much we can benefit from AVX2 compared to AVX. A bit more details offers the mix histogram tool, which prints exactly which instructions in which functions are used and how big the ratio is compared to the instructions of the whole binary. This might be important for spotting points, which later should be optimized. Last but not least there is the AVX to SSE transition checker included, which already was discussed in section 2.2.1.

This was a lot of information on how to compile and execute applications using AVX2. So here comes the part, why AVX2 offers a huge benefit compared to AVX. The probably most important and remarkable difference is the support for integer operations with 256 bit wide operands. To make things simple, it can be said, that any operation listed in section 2.2.2 without a star (*) now supports the 256 bit wide YMM registers as operands. This makes porting of applications from AVX to AVX2 in general an easy task, because with most instructions it is enough to substitute for example `%xmm1` with `%ymm1` and the instruction operates on 256 bit instead of 128 bit. Of course it is necessary to prepare the data in a different manner, change the reading and writing to memory parts and maybe use different bitmasks, but the overall effort is moderate. However it must be taken account of the lane concept already introduced with AVX. The lane concept is discussed in this section, because it only gets relevant when operating with 256 bit wide operands and we are not using the 256 bit wide floating point operations AVX provides, but we are indeed

using the 256 bit wide integer operations AVX2 provides. A *lane* refers to a 128 bit wide part of a 256 bit wide register, so every YMM register has two lanes, one for the upper 128 bit and one for the lower 128 bit. Now there exist two kind of instructions, so called *in-lane* and *cross-lane* instructions. Both types operate on the whole 256 bit of their operands, but the in-lane instructions modify the upper and lower 128 bit in a similar manner and do not transfer data between the two lanes or operate on one lane dependent on the state of the other lane, whereas cross-lane instructions operate on the 256 bit as one entity and therefore are able to transfer data between the lanes or to operate lane-dependent. A in-lane instruction operating on a 256 bit wide YMM register leads to the same result as would the execution of two times the same instruction on the upper and lower 128 bit of that YMM register. Cross-lane instructions often do not have a corresponding instruction, which operates only on the lower 128 bit or on a XMM register. Most AVX2 instructions are in-lane instructions. Aside from the movement operations, which either support already 256 bit wide operands with AVX or have no corresponding instruction, which supports 256 bit wide operands at all, all instructions listed in section 2.2.2 are in-lane instructions. For most of them it does not really matter, whether we call them in-lane or cross-lane, because they operate on packed data anyway and so the lanes are not dependent onto each other. This applies to all arithmetic and logical operations as well as to the first half of the shift operations. It should be clear, that for example a bitwise xor leads to no different result, if we carry out the instruction separately on the lower and upper half, or if we look at the operand as one entity. The second half of the shift operations and the shuffle and pack operations, however, would behave different if they were not in-lane instructions. The `vpslldq` instruction for example does not do a packed bytewise logical left shift on the 256 bit wide operand as a whole, as you might suspect first, but instead does a packed bytewise logical left shift on the upper and lower 128 bit part of its operand. This can be seen as some sort of packed operation with 128 bit wide data elements. The same applies to `vpshufb` used in the example, shown in figure 2.13, to change the endianness of packed doublewords. It is not necessary to adjust the mask used in this example, because `vpshufb` works in-lane and therefore applies the shuffle mask to the lower and upper half respectively. This means, the operands have to be changed to be YMM registers and the mask has to be copied to the upper and lower half of the YMM register containing the mask. Therefore the `vbroadcasti128` instruction, which broadcasts a 128 bit integer value to the upper and lower part of a YMM register, can be used. Figure 2.14 shows the full example and as you can see the mask is untouched and the XMM registers are just replaced by their YMM equivalents. This has the advantage that porting existing AVX code to AVX2 code is rather easy, because the instructions stay the same and just operands and the data preparation has to be adjusted. The disadvantage, however, is that the instructions cannot be used on 256 bit as an entity. The `vpshufb` instruction used in this example is not capable of shuffling bytes across the lanes, for example bytes 31 and 0 cannot be swapped. For such purposes you have to use one of the few cross-lane instructions, which are generally considered a bit more expensive than in-lane instructions, but under certain circumstances might be more powerful.

```
.align 16
.Lbswap_mask:
  .byte 3, 2, 1, 0, 7, 6, 5, 4, 11, 10, 9, 8, 15, 14, 13, 12

  vbroadcasti128 .Lbswap_mask, %ymm15
  vpshufb        %ymm15, %ymm0, %ymm0
```

Figure 2.14: Changing the endianness of packed doublewords with AVX2

At this point you already know most of the instructions, which come with AVX2, because they are the same as the ones coming with AVX, but operate on 256 bits with respect to the lane concept. There are some more instructions that do not fit in this scheme and some of them are cross-lane instructions, which will be denoted by a dagger (†) in the following listing. As in section 2.2.2 the `d` suffix will be used if there are more suffixes available. Here is an overview of AVX2 specific instructions discussed in this section:

- Movement Operations: `vpbroadcastd`, `vbroadcasti128`
- Permutation Operations: `vpermd`†, `vperm2i128`†
- Vector Shift Operations: `vpsllvd`, `vpsrlvd`
- Gather Operation: `vpgatherdd`

There is not much to say about the movement operations. The `vpbroadcastd` instruction is the integer equivalent of `vbroadcastss` and does exactly the same. It should always be used instead of `vbroadcastss`, because mixing integer and floating point instructions has impact on the performance and we are dealing with integer instructions most of the time. The `vbroadcasti128` instruction takes two operands and copies a 128 bit wide integer from memory into the lower and upper part of a YMM register. You have seen it already in figure 2.14. Just for the sake of completeness it should be mentioned that the instructions introduced at the end of section 2.2.2 got a integer equivalent with AVX2, too. Instead of `vinsertf128` and `vextractf128` you should now use `vinserti128` and `vextracti128` and the `vmaskmovps` instruction may be replaced by the `vpmaskmovd` instruction. They have not changed their behaviour, except being the integer equivalent of the respective floating point instructions.

The `vpermd` instruction works similar to the `vpshufd` instruction by doing packed doubleword shuffling, but differs from `vpshufd` in the fact that it is a cross-lane instruction. `vpermd` takes one source and destination operand, the mode of shuffling is given as a register operand containing the indices and the source operand may be a memory location. It is a very powerful operation and also often referred to as any-to-any permutation. There also exists a `vpermq` instruction, which does packed quadword shuffling and takes the mode of shuffling as an immediate value, an any-to-any permutation of bytes, however, is not possible with a single instruction. The `vperm2i128` instruction is the integer equivalent of `vperm2f128` and takes four operands. It shuffles 128 bit integer values from two potential different source operands, in which one may be a memory operand, and stores the result in the destination register. The mode of shuffling is controlled by the fourth immediate operand, which also leaves the possibility of setting parts of the destination register to zero.

The vector shift operations are new since AVX2 and allow the packed bitwise shift of doublewords with a variable shift value per doubleword. Both instructions take three operands, one source and destination register operand and a third operand, which holds the shift values and might be a memory operand. `vpsllvd` does a packed bitwise variable logical left shift of its operands, whereas `vpsrlvd` does the right shift. There also exists an instruction `vpsravd`, which does an arithmetic right shift with variable shift values.

The most interesting new instruction introduced with AVX2 is the gather operation. It allows to conditionally gather packed doublewords from memory using signed doubleword indices and to merge them into a destination register. The `vpgatherdd` instruction takes three operands, a destination register, a memory operand with the indices and a mask to conditionally select, which elements to gather. The mask has to be a register operand and is set to zero after executing the instruction, therefore you have to set the mask every time you want to use the `vpgatherdd` instruction. The second memory operand specifies a general purpose register as the common base, an AVX register for an array of indices relative to that base and a constant scale factor. It is also possible to additionally supply a displacement value. Using `vpgatherdd` might look like this:

```
vpcmpeqd   %ymm15, %ymm15, %ymm15
vpgatherdd %ymm15, 16(%rsi, %ymm1, 4), %ymm0
```

The first instruction is used to set every bit of the mask register to one. It is a so called *dependency breaking idiom*, the *ones idiom*, which can be executed faster than other instructions, because it does not depend upon its source operands, as it is known, that regardless of the input data, the output data is always *all ones* [32]. Another dependency breaking idiom would be `vpxor`, which sets every bit to zero. The second instruction now does the gathering and gets supplied the mask with every bit set to one, so it will gather 8 doublewords from the locations specified in the second operand and store the elements in `%ymm0`. The syntax of the second operand may look familiar to you, because it is the same, which is used when writing x86_64 assembler code, that deals with general purpose registers. So what happens here is that the `%rsi` register is used as base, the 8 doubleword indices are taken from `%ymm1` and multiplied by the scale factor 4 and then there is a displacement of 16 added to the result. So for the 8 doublewords the addresses are calculated like this while i is running from 0 to 7:

```
%rsi + %ymm1[32*i+31:32*i]*4 + 16
```

We can benefit a lot from the gather operation, because it is perfect for doing table lookups. In cryptographic algorithms there are often complex algebraic operations, which then get, to gain performance, transformed in lookups into precalculated tables. Now if we are processing data parallel, we can do 8 table lookups

in parallel as well. Without the `vpgatherdd` instruction we need to do the 8 table lookups sequentially and even worse we need to extract the data to general purpose registers first, then do the lookup and after that insert the data again into some AVX register. It is obvious that this causes a big performance overhead compared to the gather operation and even if we currently do not know, how fast this instruction will be executed on real hardware, we can say that just by saving instructions it should give us a good performance boost. It is also possible to gather quadwords instead of doublewords with the `vpgatherqq` instruction. The first `q` stands for the width of the indices, the second `q` for the width of the gathered data and there are also mixed forms available, for example `vpgatherdq`. However the index and data width is restricted to doublewords and quadwords and it is not possible to gather, for example, bytes with one operation. At the latest with the possibilities this powerful instruction provides, it can be concluded that AVX2 offers a huge advantage over plain AVX, when implementing cryptographic algorithms that use integer operations most of the time.

2.3 The Linux Kernel

Linux refers to an unix-like operating system assembled under the model of free and open source software development. The Linux kernel itself was first released in 1991 by Linux Torvalds, at that time a 21-year-old Finnish computer science student. There exists this one very famous quote of him announcing, that he currently started working on his own operating system [56]:

> I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. [...]

It soon turned out, that he was wrong with this statement. Tough in the first days being restricted to the 80386 architecture, Linux today runs on almost everything ranging from small embedded devices to large mainframes and supercomputers and is said to be ported to more hardware platforms than any other operating system available. It is still not the dominant desktop operating system, but leading on servers and used in highly tailored systems, such as mobile phones and routers. The Linux kernel also serves as base for the Android system, which is currently a popular operating system on smartphones.

On 21 July 2011 Linus Torvalds announced the release of Linux 3.0 [58], which actually was not expected in the first place as there always was the agreement to stick with the 2.6 versioning scheme for the next time. Linux is still licensed under version 2 of the GPL [57] and currently it looks like the license will not be changed to GPLv3, at least Linus himself is not a huge fan of the GPLv3 and wants to stick with the GPLv2. The choice to select the GPLv2 as license for the kernel, however, certainly helped the kernel growing so fast as everyone is able to read and modify the code, contribute to the current kernel and on the other hand is forced to publish the source code of the modifications he or she makes along with the binary blob. The license is good for us as well, because we were not able to write a patch, if we had to deal with a closed source kernel. There would be a lot to say about the Linux kernel in general and the following sections shall give a rather short overview on specific topics that are relevant for this thesis.

2.3.1 Kernel Development

Kernel development is a challenging task. The downside of being ported to so many architectures, shipping drivers for a whole bunch of different devices and trying to satisfy the requirements of different user groups is the constantly increasing maintenance effort. The current development model is such that Linus Torvalds still releases the new versions, called the *mainline* kernels, which contain the generic branch of development. This branch is officially released approximately every three months. When starting a new series, usually there is a two-week merge window, to integrate major changes into mainline. After that there follow several iterations to fix bugs in the submitted major changes or different locations in the kernel. During this phase the kernel gets an additional suffix, for example `-rc1`, to show that it is not released yet. The kernel will finally be released and from this point still gets security fixes. A third digit is used to indicate this. For the last few major releases there exists a stable release, which gets only the security fixes

but not the new features submitted to mainline. The current kernel releases are provided as tarball on the official website [30] and the development branches can be accessed via git.

Of course Linus Torvalds, as a single person, cannot take care of the huge source base on his own and so there is at least one maintainer for every subsystem or architecture the kernel contains, who often manages his own repository and is responsible for keeping everything in the specific subsystem together. So it usually happens that Linus just gets all the changes from the maintainers in the time of the merge-window and merges them together in his official development branch. For normal persons submitting patches still works via email and it is advisable to write to the right maintainer to avoid your mail is just getting lost. Most of the patches are actually sent in by developers of large companies such as Intel or Red Hat, because often they provide code to support their own hardware devices, but in fact everyone is able to send in a patch. Besides just writing to the maintainer, for every subsystem there exists a mailinglist, where a copy should be sent to. There also exists a list for the kernel as a whole, the *Linux Kernel Mailing List* (LKML), which is used by everyone contributing to the kernel. The volume on this list is consequently very high, about 200-300 messages per day. An overview over all lists is provided on `vger.kernel.org` [29]. The advantage of using this lists is, that everyone can read the suggested patches and discuss them on the list. After there is enough agreement, the patch might be integrated into mainline.

When you want to submit a patch to the Linux Kernel there are a few things you have to consider. First of all Linux is often referred to as a *moving target*, because there exists no stable API inside the kernel and so it is important that your patch is cleanly applicable to the currently available development version. The patch should not be too big, so if you have to patch a lot of things, it is better to split the patch. A rule of thumb is one patch per logical feature. Besides just working correctly there are some style issues you have to respect. A very detailed description of how to format your code is located in `Documentation/CodingStyle` within the Linux kernel source tree. If you come from the GNU side you probably should read it as the guidelines substantially differ from the GNU coding standards, as this quotation from the beginning of the Linux Coding Guidelines makes clear:

> First off, I'd suggest printing out a copy of the GNU coding standards, and NOT read it. Burn them, it's a great symbolic gesture.

If you have formatted your code this way, you can generate a patch using the standard unix-tool `diff` or `git format-patch`, which is probably easier, if you are already using version control. It is important that the patch is in the unified diff format and has a meaningful commit message attached to it, which contains the line "Signed-off-by: <yourmail>", to confirm that you are the author of this patch. There are a lot more things to consider and `Documentation/SubmittingPatches` gives you a full list of things you should check before submitting. To get sure that you made no mistake there exists a script located in `scripts/checkpatch.pl`, which is able to check your patch against styling issues. Before sending the patch you should at least check it with this script. If you do not know the maintainer of the code you are working on, there is another script `scripts/get_maintainer.pl`, which tells you who is responsible for a specific file. After these checks you are ready to send your patch to the maintainer and a copy to the LKML and the mailinglist of the specific subsystem. You should send the patch inline, as plain text and not as attachment, because this makes it is easy to comment on the patch. For this thesis most of the modifications happen in `arch/x86/crypto/` and so we find out, that the maintainer of the crypto subsystem is Herbert Xu and there also exists a subsystem specific list `linux-crypto@vger.kernel.org`. Therefore we will send our patches to Herbert Xu, the crypto list and finally the LKML.

2.3.2 Source Tree

This section shall give a brief overview of the source code structure of the kernel and point out the parts, which we need to modify, if we want to submit a patch, which adds a cryptographic algorithm. We do not need to make modifications in many parts of the tree and this keeps things rather simple. In the following listing only directories, which contain kernel relevant source code are listed, whereas build and script files are removed. So here is like the kernel tree is structured:

- **arch/x86**: x86_32 and x86_64 specific source code
  - **crypto**: x86 specific implementation of ciphers
  - **include/asm**: x86 specific kernel headers
- block: Block I/O layer
- **crypto**: Crypto API
- drivers: Device drivers
- firmware: Device firmware
- fs: Filesystem implementations
- **include**: Kernel headers
  - **crypto**: Crypto API headers
- init: Kernel boot and initialization code
- ipc: Interprocess communication
- kernel: Core subsystems (e.g. scheduling)
- lib: Helper routines
- mm: Memory Management subsystem
- net: Networking subsystem (Ethernet, IPv4, IPv6, ...)
- security: Linux Security Module
- sound: Sound subsystem
- virt: Virtualization infrastructure

As we are using AVX, our implementation is x86-specific and to be precise it is x86_64-specific. It is not a big surprise that most of the code will go into arch/x86/crypto. In fact the whole implementation and also the gluecode goes there. Some modifications have to be made directly in crypto, because the Crypto API provides a testmanager, which is located there. Finally there are the architecture specific header files which need to be adjusted in arch/x86/include/asm and the generic header files in include/crypto, which need to be adjusted for the Cast-128 and Cast-256 implementation.

2.3.3 Cryptographic API

As already mentioned in section 2.3.1, the maintainer of the crypto subsystem is Herbert Xu. He takes care of the crypto API in the kernel and decides, whether a patch will be accepted or not. Unfortunately the crypto API is not very well documented, there exists only a brief introduction [14], which explains the features of the crypto API. The API supports five types of transformations: AEAD (Authenticated Encryption with Associated Data), block ciphers, ciphers, compressors and hashes. The names are a bit confusing. Ciphers means in fact block ciphers, i.e. a cipher, which exactly processes one block such as AES, Blowfish, Twofish or Serpent. Block ciphers are all types of ciphers including stream ciphers and ciphers combined with a specific mode of operation, e.g. AES with ECB. Compressors are algorithms like Deflate or LZO, which can be used to save space, and hashes are, as you would think, algorithms like MD5 and SHA1. Originally the crypto API was completely synchronous but by now there also exists an asynchronous interface, which was primarily designed for hardware crypto devices. Since multi core systems can benefit from the asynchronous interface as well, it is used all over the crypto subsystem. Unfortunately this interface is even worse documented, but at least the code already written with this interface is readable and can be adjusted to our needs. However, as stated in section 2.3.1 there exists no stable API within the kernel and so the crypto API is not stable either. The patches we write can be applied to version 3.4 of the kernel. There were some changes from 3.3 to 3.4 in the crypto part and the patches are not easily applicable to version 3.3, but they can be rather easy applied to version 3.5. Every patch has to be adjusted between major kernel

versions, but fortunately only the gluecode has to be changed as the tuned assembler implementation of an algorithm itself is developed and tested in userspace anyway.

The crypto API makes it possible to abstract in the cases where the cryptography is actually needed. Cryptoloop or dm-crypt, which is part of the device mapper infrastructure, use the crypto API of the kernel and do not implement cryptographic routines on their own. So if we add algorithms to the crypto API of the kernel, we can use them, for example with dm-crypt, to encrypt our harddisk. This is very comfortable, as we have to patch only one specific component of the kernel. For many components of the Linux kernel you have the choice to either build it into the kernel or compile it as a loadable module. This applies to algorithms within the crypto API as well and has the advantage that they are only loaded, when they are needed, for example by the device mapper. Every algorithm compiled as module can be loaded by its module name or alias and by the priority of the specific algorithms it is decided, which one is actually used. The algorithms from the x86 specific directory `arch/x86/crypto` usually have a higher priority than the generic ones from the `crypto` directory and so the faster algorithms are used, when both are available.

Regarding symmetric ciphers, the crypto API allows to separate the code of the block cipher itself from the mode of operation, with which it can be used. Actually we needed only to implement our specific assembler tuned implementation of the cipher and the crypto API would make it possible to use it with all available and compatible modes of operation. In our case, however, it is not that easy, because to speed things up we want to process more than one block in parallel and this is not supported by the generic implementation of the modes of operation, because they expect the block cipher to process exactly one block. Consequently we are forced to reimplement every mode of operation with some gluecode and call our parallel n-way block cipher whenever it is possible. In cases where it is not possible, for example in the encryption part of CBC, we have to fall back to a 1-way block cipher, which is usually already implemented. This breaks with the design of the crypto API and the separation between block cipher and mode of operation is no longer possible, but for the sake of speed we have to go this way and reuse as much of the functionality, the crypto API provides, as possible.

One more feature the crypto API provides is a test module called `tcrypt`. It can be loaded with various parameters, which have to be looked up from the source code, and is capable of validating the correctness of cryptographic algorithms with known test vectors. It is able to check every mode of operation and also provides functionality for making benchmarks. `tcrypt` can for example count cycles for different key and block sizes of specific ciphers or measure how much data a specific cipher can process in a given time. For newly registered algorithms with different names, a test routine should be provided to allow the crypto API doing a self-test and to verify the correctness. For n-way parallel algorithms it might be necessary to provide larger test vectors, to avoid the legacy 1-way algorithm getting called. Thus structured rather simple, the `tcrypt` module gives us suitable benchmark results, which can be used for the evaluation of our implementations in kernelspace.

3 IMPLEMENTATION

In this chapter we will describe the specific implementations, which have been produced in the context of this thesis. All implementations have been developed in userspace and after verifying, that they work as intended, they have been integrated into the kernel and a patch has been generated. Therefore the source code of the generic implementation, existing x86-specific implementations and the common parts, for example key scheduling definitions and the context declarations in the header files, have been copied from the Linux kernel tree. The source files have been changed in a trivial way, to make the present implementations running in userspace. For example the data types have to be changed from the Linux kernel internal types to types defined in `stdint.h`, header files have to be removed or replaced and the gluecode, which calls the specific implementations has to be adjusted. Most of the gluecode can simply be removed, because we do not need all the code, which registers the algorithms within the crypto API of the kernel. Moreover we do not need all available modes of operation, because we have to write specific gluecode for our implementation along with the kernel integration anyway. In userspace we are just developing the AVX and AVX2 implementations of the block ciphers, which are capable of processing blocks in parallel, and testing them in ECB mode. This makes it easy to compare the results of the different implementations and verify whether an implementation is correct. It is also not very difficult to make some first time measurements to decide, whether our implementation is faster than the ones, that are currently present. To compile the algorithms the GCC is used and an own Makefile has been written, capable of building the different implementations in userspace. A new enough version of GCC is able to compile AVX and AVX2 implementations. The AVX implementations are tested on real hardware and the AVX2 implementations have to be tested within the Intel Software Development Emulator [10]. The algorithms themselves are written in plain assembler, but make use of the C-Preprocessor. GCC automatically applies the CPP before assembling, if the source files have a capital "s" as suffix (`.S`).

In section 3.1 the implementation of the Serpent cipher will be presented. First the AVX implementation will be described, the differences or improvements that come with AVX2 are documented and finally the kernel patch is shown. In section 3.2 we introduce the implementation of Twofish. For Twofish we provide an AVX and AVX2 implementation as well. The kernel patch of Twofish is presented in section 3.2.3. In the sections 3.3, 3.4 and 3.5 we will have a look at Blowfish, Cast-128 and Cast-256. For each algorithm the AVX and the AVX2 implementation will be shown, as well as a kernel patch.

3.1 Serpent

In this section the implementation of the Serpent block cipher will be presented. We will implement the cipher analogous to the approach in section 2.1.3, i.e. in bitslice mode with the S-boxes implemented as logical sequences. The block size of Serpent is 128 bit and we will provide an AVX implementation, which processes eight blocks at once, i.e. 128 bytes of data, and an AVX2 implementation, which processes sixteen blocks at once, i.e. 256 bytes of data. With AVX only four blocks and with AVX2 only eight blocks are processed in parallel, but the implementations process two four respectively eight block chunks sequentially and therefore are able to save the loading of the round key from memory one time, because the key can be kept in a register during the processing. There exists already a SSE2 implementation of the Serpent cipher in the Linux kernel and the AVX implementation does not differ very much from the SSE2 implementation, because of the drawbacks mentioned in section 2.2. The S-boxes as logical sequences can be looked up from the generic implementation and simply be transformed to SIMD operations. Some speedup with AVX, compared to SSE2, can be achieved by exploiting the non-destructive three operand syntax and with AVX2 we can gain performance by using the 256 bit wide registers.

```
#define SERPENT_EXPKEY_WORDS  132

struct serpent_ctx {
    u32 expkey[SERPENT_EXPKEY_WORDS];
};
```

Figure 3.1: Serpent context

```
void __serpent_enc_blk_8way_avx(
  struct serpent_ctx *ctx, u8 *dst,
  const u8 *src, bool xor);
void serpent_dec_blk_8way_avx(
  struct serpent_ctx *ctx, u8 *dst,
  const u8 *src);
```

Figure 3.2: Serpent function prototypes

The key schedule is still implemented the generic way, because the affine recurrence used in the key scheduling has to be done sequentially anyway. So we assume the 33 subkeys (132 doublewords of key material) are properly generated and saved in the Serpent specific context, used by the encryption and decryption routine. The context is defined as shown in figure 3.1 and the function prototypes for our assembler implementation look like the ones in figure 3.2. In this figure the prototypes for the AVX implementation are shown, but the ones for AVX2 have the same signature and just a different function name. The functionality of the two functions should be self explaining. They get the current context, which contains just the generated subkeys, a pointer to the source and one to the destination and encrypt respectively decrypt exactly eight blocks (or sixteen blocks with AVX2) and write the result to the position of the destination pointer. The encryption routine gets a fourth boolean parameter and xor's the result of the encryption with the destination instead of just writing to memory, if this parameter is set to true. Some modes of operation, for example CTR, would xor the result anyway and this way the xor operation is done directly by the encryption routine and a memory loading and storing operation can be saved.

3.1.1 AVX Implementation

Now let us have a look at the AVX implementation. To make things simpler the registers have been renamed with the help of macros. The definition is shown in figure 3.3. First the eight blocks have to be read from memory and therefore we use eight registers and read sequentially into the registers. A block is 128 bit wide and fits exactly into an AVX register. All Serpent operations work with the four doublewords of one block and that is why the input data has to be reordered to be ready for parallel processing. In figure 3.4 you see how the data is loaded from memory and transformed. You can see that the data is processed in two independent four block chunks. In this listing `%rdx` is the source pointer, which matches the function prototype in figure 3.2 according to the x86_64 calling convention. The registers `RA1` to `RD1` and `RA2` to `RD2` take the input data and `RE1` and `RE2` are additional registers, which are needed in the processing later on. The registers `RK0` to `RK3` are actually used to store the doublewords of the corresponding round key, but in the reading and transforming process in figure 3.4 they just act as temporary registers. The register `RNOT` contains a mask, which is used to negate 128 bits at once and `tp` is a special temporary register, which will later save instructions in the application of the S-boxes. The macro `read_blocks` reads and transforms one four block chunk from memory. The reading is pretty straight forward, the four registers are filled

```
#define RA1 %xmm0
#define RB1 %xmm1
#define RC1 %xmm2
#define RD1 %xmm3
#define RE1 %xmm4

#define tp  %xmm5

#define RA2 %xmm6
#define RB2 %xmm7
#define RC2 %xmm8
#define RD2 %xmm9
#define RE2 %xmm10

#define RNOT %xmm11

#define RK0 %xmm12
#define RK1 %xmm13
#define RK2 %xmm14
#define RK3 %xmm15
```

Figure 3.3: Register definitions

```
#define transpose_4x4(x0, x1, x2, x3, t0, t1, t2) \
    vpunpckldq          x1, x0, t0; \
    vpunpckhdq          x1, x0, t2; \
    vpunpckldq          x3, x2, t1; \
    vpunpckhdq          x3, x2, x3; \
    \
    vpunpcklqdq         t1, t0, x0; \
    vpunpckhqdq         t1, t0, x1; \
    vpunpcklqdq         x3, t2, x2; \
    vpunpckhqdq         x3, t2, x3;

#define read_blocks(in, x0, x1, x2, x3, t0, t1, t2) \
    vmovdqu (0*4*4)(in),    x0; \
    vmovdqu (1*4*4)(in),    x1; \
    vmovdqu (2*4*4)(in),    x2; \
    vmovdqu (3*4*4)(in),    x3; \
    \
    transpose_4x4(x0, x1, x2, x3, t0, t1, t2)

leaq (4*4*4)(%rdx), %rax;
read_blocks(%rdx, RA1, RB1, RC1, RD1, RK0, RK1, RK2);
read_blocks(%rax, RA2, RB2, RC2, RD2, RK0, RK1, RK2);
```

Figure 3.4: Reading and transforming input blocks

with four sequential blocks, but the interesting part is the transformation done in `transpose_4x4`. In fact `transpose_4x4` can be seen as a 4x4 matrix transposition. After the transformation x_0 contains the lowest, x_1 the second lowest, x_2 the third lowest and x_3 the highest doublewords of all four blocks. This is achieved by first interleaving doublewords with `vpunpckldq` and `vpunpckhdq` and after that interleaving quadwords with `vpunpcklqdq` and `vpunpckhqdq`. This way we need just four memory accesses and everything else is done in registers. If we would implement this transformation with SSE2, instead of AVX, it would take several instructions more, because we could not benefit from the three operand syntax. For example the first `vpunpckldq` in `transpose_4x4` would destroy the value of x_0 and therefore this value had to be saved first by an additional movement operation. The same applies to all instructions were both source operands are reused in a following instruction. Now we are able to do the operations, we would do with the four doublewords of one block with the four doublewords of four blocks in parallel by substituting the scalar operations with equivalent SIMD instructions, like shown in figure 2.10. After the encryption routine itself, we have to reverse the transformation before writing to the destination in memory. Since a transposition is an invertible operation, we can use the same transformation again before writing the result to memory.

The implementation of the actual Serpent encryption routine is not really complicated, because the whole algorithm consists just of basic logical operations and can be implemented really close to the pseudocode listed in figure 2.3. The S-boxes can be taken directly from the generic implementation and be transformed to corresponding SIMD instructions. This is almost always possible, however the negation has to be replaced by a xor operation with a register containing all ones. The register `RNOT` has been reserved for this purpose. In figure 3.5 you see the S-box S_0 implemented with SSE2 and in figure 3.6 the same S-box S_0 is implemented with AVX. The SSE2 implementation executes exactly the same instructions as the S-box S_0 form the generic implementation, shown in figure 2.6. Every instruction can be directly transformed to its SIMD equivalent and the negation is replaced by xor with `RNOT`. It is no big step from SSE2 to AVX, because it is possible to duplicate the destination operand and use it as second source operand and destination operand simultaneously. With the introduction of the new register `tp` and the three operand syntax we can now save the first `movdqa` instruction by moving the result to a different register and keeping the value of both source operands. At the end of the S-box $x_0, \ldots, x_4$ have the same values because of some subtle substitutions. By this technique we can save one movement instruction in every of the eight S-boxes, and therefore are able to save 64 instructions when encrypting eight blocks, just by doing this simple trick.

The linear transformation L is implemented close to figure 2.4. L operates on doublewords and so the transformation can be implemented with SIMD instructions very easily. The rotations have to be replaced

```
#define S0(x0, x1, x2, x3, x4) \
    movdqa  x3,   x4; \
    por     x0,   x3; \
    pxor    x4,   x0; \
    pxor    x2,   x4; \
    pxor    RNOT, x4; \
    pxor    x1,   x3; \
    pand    x0,   x1; \
    pxor    x4,   x1; \
    pxor    x0,   x2; \
    pxor    x3,   x0; \
    por     x0,   x4; \
    pxor    x2,   x0; \
    pand    x1,   x2; \
    pxor    x2,   x3; \
    pxor    RNOT, x1; \
    pxor    x4,   x2; \
    pxor    x2,   x1;
```

Figure 3.5: S-box S_0 implemented with SSE2

```
#define S0(x0, x1, x2, x3, x4)  \
    \
    vpor     x0,   x3, tp; \
    vpxor    x3,   x0, x0; \
    vpxor    x2,   x3, x4; \
    vpxor    RNOT, x4, x4; \
    vpxor    x1,   tp, x3; \
    vpand    x0,   x1, x1; \
    vpxor    x4,   x1, x1; \
    vpxor    x0,   x2, x2; \
    vpxor    x3,   x0, x0; \
    vpor     x0,   x4, x4; \
    vpxor    x2,   x0, x0; \
    vpand    x1,   x2, x2; \
    vpxor    x2,   x3, x3; \
    vpxor    RNOT, x1, x1; \
    vpxor    x4,   x2, x2; \
    vpxor    x2,   x1, x1;
```

Figure 3.6: S-box S_0 implemented with AVX

by two shift operations and an or operation, because AVX does not offer packed rotations. If the linear transformation L is implemented with SSE2 eight movement instructions are required, because the packed shift operations destroy their source operand (6 rotations and 2 shifts). With AVX we can save all eight movement instructions the same way we have done it with the S-boxes. The linear transformation L is applied 31 times in the encryption process of a four block chunk and so we can save $8 \cdot 31 \cdot 2 = 496$ more instructions, by substituting the instructions in the linear transformation, in the process of encrypting eight blocks. We will not show the whole implementation of L at this point, because it is not very interesting and straight forward to implement. In every round the four doublewords of the round key are loaded into the registers `RK0` to `RK3`. The lowest doubleword of the round key is broadcasted, i.e. copied to every position, into `RK0`, the second lowest doubleword is broadcasted into `RK1`, the third lowest doubleword into `RK2` and the highest doubleword into `RK3`. With SSE2 it is necessary to load the specific doubleword with `movd` and broadcast it explicitly with a shuffle operation like `pshufd`, whereas with AVX the two instructions can be replaced with `vbroadcastss`, which loads the doubleword from memory and broadcasts it to every position in the destination register. In the encryption process the 33 round keys have to be loaded and that is why we can save another $4 \cdot 33 = 132$ instructions compared to SSE2. The round keys are used for both four block chunks and that is why we do not multiply by two in this calculation. So to sum things up, we can save $64 + 496 + 132 + 20 = 712$ instructions per encryption of eight blocks compared to SSE2 by removing the `movdqa` and `movd` instructions. The 20 instructions are saved in the input transformation. Of course this brings not a gigantic speedup compared to the SSE2 implementation, but it is measurable in applications using this improved cipher. The detailed performance evaluation will be shown in section 4.4.1.

3.1.2 AVX2 Implementation

The AVX2 implementation of the Serpent cipher has been developed based on the AVX implementation and is a very good example on how easy it is, to migrate an existing algorithm from AVX to AVX2. The reason for this is the in-lane behaviour of most AVX2 instructions explained in section 2.2.3. The most modifications need to be done in the reading from and writing to memory part. First the register definitions have to be adjusted. It is sufficient to replace the XMM registers with YMM registers, e.g. `%xmm0` with `%ymm0`, in the macro definitions, because this changes all instruction operands in the algorithm itself. Instead of using `vbroadcastss` to broadcast the doublewords of the round key, we will now use the `vpbroadcastd` instruction. They both work exactly the same way, but the latter one is an integer instruction and therefore preferable compared to `vbroadcastss`. Now let us have a look at the reading and writing parts. An AVX2 register is 256 bit wide and capable of taking two 128 bit blocks. Of course we need to adjust the displacement in the reading and writing operations as shown in figure 3.7 for the output operation. This has to be done with every instruction in the input and output parts, but basically you just have to substitute 4 for 8. After these small changes the whole algorithm is working with AVX2. The reading process

```
vmovdqu     x0, (0*4*4)(out); \
vmovdqu     x1, (1*4*4)(out); \
vmovdqu     x2, (2*4*4)(out); \
vmovdqu     x3, (3*4*4)(out);
```

```
vmovdqu     x0, (0*8*4)(out); \
vmovdqu     x1, (1*8*4)(out); \
vmovdqu     x2, (2*8*4)(out); \
vmovdqu     x3, (3*8*4)(out);
```

Figure 3.7: Writing to memory with AVX and AVX2

shown in figure 3.4 still works the same way. Instead of a four block chunk an eight block chunk is read sequentially from memory and stored in four 256 bit wide AVX2 registers, each taking two blocks. After this the transformation is applied without any changes on the YMM registers and because all the operations in the transformation are in-lane operations, the upper and the lower half get transformed like they would if they were standalone XMM registers. This results in four registers containing respectively eight doublewords from eight blocks. So x_0 contains the lowest doubleword from each block, x_1 the second lowest and so on. Now as we have verified that the input and output routines still work as intended, we need to think about the encryption process itself. All operations we use in the encryption process are operations on packed doublewords and therefore they work exactly the same way, because the data is provided correctly. Instead of doing the SIMD operation on four doublewords in parallel, now eight doublewords in parallel are processed. Because we are just using packed operations, we do not need to adjust the algorithm in a more sophisticated way. This improvement brings probably a huge speedup, because we can approximately cut in half the number of instructions, we need to encrypt a long message. We will take a more detailed look on the instructions that can be saved in section 4.4.1, but by now we can say that AVX2 gives us a good performance boost compared to a rather moderate implementation effort.

3.1.3 Kernel Integration

You have seen how the parallel block cipher itself is implemented and in this section we will discuss the integration of this block cipher into the Linux kernel. Only for the AVX implementation a kernel patch [21] has been developed, because currently there is no support for AVX2 in the Linux kernel and moreover we could not test it on real hardware. However there is not a big difference between the integration of the AVX2 and the AVX implementation and so it should be easily possible to write a patch for the AVX2 implementation, too, once the hardware and the kernel is ready. The AVX block cipher implementation is provided for the kernel without modifications compared to the implementation tested in userspace and explained in section 3.1.1. Particularly no assembler code has to be modified to integrate the cipher, but instead we have to write a lot of gluecode to reimplement the modes of operation, that make use of our block cipher.

We will provide the accelerated block cipher as loadable kernel module with an own entry in Kconfig, the build configuration system of the Linux kernel. This way the user can decide, whether he or she wants to build the cipher at all, as loadable module or directly into the kernel. Therefore we do not offer the cipher as an external module, but provide a patch that can be applied against the current crypto development tree. To achieve this, we have to modify the Makefile in `arch/x86/crypto/Makefile` and create a new entry in the Kconfig file located in `crypto/Kconfig`. The actual cipher and the corresponding gluecode, which contains the module information, is located in `arch/x86/crypto`. Last but not least the testmanager of the crypto API has to be adjusted in `crypto/testmgr.c`, to allow the crypto API doing a self-test with the new cipher. The most interesting part is the gluecode in `serpent_avx_glue.c`, because you have already seen the implementation in `serpent-avx-x86_64-asm_64.S` and the change in the Makefile just adds a rule that compiles those two files dependent on the Kconfig selection and links them together to a loadable module or directly into the kernel image.

The module, which is generated from the gluecode, does not register the block cipher itself, because our cipher processes eight blocks in parallel and is not able to process just one block, but instead ten algorithms, which combine our parallel cipher with a specific mode of operation. We provide support for five different modes of operation (ECB, CBC, CTR, LRW, XTS) and for each mode a synchronous block cipher is registered within the crypto API with an unique name. After that the five modes are registered as asynchronous block ciphers with the common name of the mode of operation combined with our block cipher. The latter

```
static int __init serpent_init(void)
{
    u64 xcr0;
    if (!cpu_has_avx || !cpu_has_osxsave) {
        printk(KERN_INFO "AVX instructions are not detected.\n");
        return -ENODEV;
    }
    xcr0 = xgetbv(XCR_XFEATURE_ENABLED_MASK);
    if ((xcr0 & (XSTATE_SSE | XSTATE_YMM)) != (XSTATE_SSE | XSTATE_YMM)) {
        printk(KERN_INFO "AVX detected but unusable.\n");
        return -ENODEV;
    }
    return crypto_register_algs(serpent_algs, ARRAY_SIZE(serpent_algs));
}

static void __exit serpent_exit(void)
{
    crypto_unregister_algs(serpent_algs, ARRAY_SIZE(serpent_algs));
}

module_init(serpent_init);
module_exit(serpent_exit);

MODULE_DESCRIPTION("Serpent Cipher Algorithm, AVX optimized");
MODULE_LICENSE("GPL");
MODULE_ALIAS("serpent");
```

Figure 3.8: Serpent AVX module initialization

five algorithms make use of the first five registered and compete with different implementations of the same cipher. The winner is selected by the priority given to the ciphers and is used by the application asking for it. Because of the five different modes and the implementation of the asynchronous cipher support, we need to provide a lot of gluecode to support our cipher within the Linux kernel. Currently there is not an API for this purpose and that is why we have to copy a lot of existing gluecode. There already exists good gluecode for the SSE2 implementation of Serpent and the AESNI implementation of AES and we can adopt most of the generic stuff from this files. Duplicating code, however, is not very nice and it seems that in the future with more SIMD implementations showing up there will be a reasonable API to unify the different gluecodes to a generic gluecode implementation. For the moment we will stick with copying existing gluecode and adjusting it to our needs. Basically we need to provide implementations for the five different modes of operations, code for the asynchronous cipher implementation and we have to detect the availability of AVX and register the algorithms in the module initialization routine.

The module initialization is shown in figure 3.8. The detection of the availability of AVX is in fact the same as in figure 2.11, but makes use of some Linux kernel specific functions and macros to avoid writing assembler code. It is checked whether the CPU supports AVX and whether the operating system has enabled the extension, i.e. saves and restores the new registers across context switches. After that the function `crypto_register_algs` is called with an array as parameter, which contains the descriptions of the algorithms we provide with this module. The array consists of structures, which are initialized with the informations for the corresponding algorithms. In figure 3.9 you see the initialization of the structure of the synchronous Serpent block cipher operating in ECB mode. This is just a helper algorithm to be used by the asynchronous block cipher, which initialization is shown in figure 3.10. Both figures have been changed in some trivial manner to fit on the page, so if you want to know the details, you may want to have a look at the source code. Most of the options are available with both types, synchronous and asynchronous algorithms. Here is an explanation of the options, which have to be set to register an algorithm within the crypto API of the kernel:

- `.cra_name` is the common name of the algorithm. The algorithm is requested by this name. For the synchronous algorithm this name is just used by the asynchronous algorithm, but for the asynchronous algorithm the name has to correspond to the convention of the crypto API, to be selected by applications. The convention is to write the mode of operation followed by the block cipher used

```
.cra_name        = "__ecb-serpent-avx",
.cra_driver_name = "__driver-ecb-serpent-avx",
.cra_priority    = 0,
.cra_flags       = CRYPTO_ALG_TYPE_BLKCIPHER,
.cra_blocksize   = SERPENT_BLOCK_SIZE,
.cra_ctxsize     = sizeof(struct serpent_ctx),
.cra_alignmask   = 0,
.cra_type        = &crypto_blkcipher_type,
.cra_module      = THIS_MODULE,
.cra_list        = HEAD_INIT(algs[0].cra_list),
.cra_u = {
    .blkcipher = {
        .min_keysize = SERPENT_MIN_KEY_SIZE,
        .max_keysize = SERPENT_MAX_KEY_SIZE,
        .setkey      = serpent_setkey,
        .encrypt     = ecb_encrypt,
        .decrypt     = ecb_decrypt,
    },
}
```

Figure 3.9: Synchronous Serpent description (ECB)

```
.cra_name        = "ecb(serpent)",
.cra_driver_name = "ecb-serpent-avx",
.cra_priority    = 500,
.cra_flags       = CRYPTO_ALG_TYPE_ABLKCIPHER,
.cra_blocksize   = SERPENT_BLOCK_SIZE,
.cra_ctxsize     = sizeof(struct async_ctx),
.cra_alignmask   = 0,
.cra_type        = &crypto_ablkcipher_type,
.cra_module      = THIS_MODULE,
.cra_list        = HEAD_INIT(algs[5].cra_list),
.cra_init        = ablk_init,
.cra_exit        = ablk_exit,
.cra_u = {
    .ablkcipher = {
        .min_keysize = SERPENT_MIN_KEY_SIZE,
        .max_keysize = SERPENT_MAX_KEY_SIZE,
        .setkey      = ablk_set_key,
        .encrypt     = ablk_encrypt,
        .decrypt     = ablk_decrypt,
    },
}
```

Figure 3.10: Asynchronous Serpent description (ECB)

by this mode in brackets. Usually a generic mode of operation implementation can work with any generic block cipher implementation, but we need to provide both at once. `.cra_driver_name` is an unique name for the driver to be registered.

- `.cra_priority` is the priority, which is used by an application to decide, which of the ciphers with the same name to load. The synchronous implementation has priority 0, because it is not intended to be used directly by an application, whereas the asynchronous implementation has priority 500, which is very high. The SSE2 implementation has priority 400, so an application would choose the AVX implementation, if both get loaded at the same time.
- `.cra_blocksize` specifies the block size in bytes. `.min_keysize` and `.max_keysize` specify the minimal respectively maximal key size in bytes. There is no possibility to specify what steps between the two limits are allowed, so this has to be indicated by an error, when using the cipher.
- `.setkey`, `.encrypt` and `.decrypt` specify which functions the crypto API should use to generate the round keys, encrypt or decrypt a message. In the synchronous case the functions are the straight implementations of the key scheduling or the mode of operation in combination with the block cipher, but in the asynchronous case the functions are part of the asynchronous crypto API, which delegates the calls to the synchronous implementation.

The other options listed in figure 3.9 and 3.10 have a rather technical meaning. The options `.cra_init` and `.cra_exit` are specific for the asynchronous implementation and are pointers to functions for the initialization and uninitialization of the asynchronous interface. The reason why we just provide the asynchronous implementation to applications is that the crypto API can provide them as synchronous algorithms to applications as well and so we can cover both cases with one implementation. The function `serpent_setkey` is taken from the generic implementation and the functions `ecb_encrypt` and `ecb_decrypt` implement the ECB mode and call the parallel AVX implementation of the Serpent block cipher. The implementation of these two functions is very short, because it just calls the block cipher sequentially on successive parts of the message. The implementation of different modes than ECB, for example CBC and CTR, is a little bit more complicated, but not very difficult. The figures 2.1 and 2.2 already suggest how to implement the specific mode. In all cases the generic implementation, which processes just one block, is called, if the remaining message is too small for being processed by the parallel AVX implementation. The asynchronous functions are wrappers around the functions, which implement the modes of operation. These functions all contain a lot of code, but are not worth to be presented here, because they do not introduce new ideas related to our block cipher and it would take much space to describe them here.

The last component, which has been adjusted by our patch, is the testmanager of the crypto API. The

testmanager already contains tests and large enough testvectors for Serpent with all the modes of operation we use. However these tests are registered with the common name, e.g. `ecb(serpent)`, and because we are registering algorithms with new names, for example `__ecb-serpent-avx`, we have to provide a test for these, too. The algorithm itself is already tested with the common name, because the common name refers to the asynchronous implementation and this implementation calls the synchronous implementation, which calls the algorithm. It is useless to add another test for the same algorithm, but if we do not add a test, we get warnings in the kernel log and that is why we just add a so called null test for the five synchronous algorithms to the testmanager. After doing this, our algorithm is still tested by the existing tests for the common names, but we do not get warnings anymore.

In this section you have seen, how the cipher is integrated into the Linux kernel [21]. Of course it is not possible to discuss all parts of the code, which are necessary to integrate it properly, but this section should have given you a good overview. In section 4.3, where we show how to verify the correctness of an algorithm, you will see how the algorithms, we have registered within the crypto API, are actually used by an application.

3.2 Twofish

In this section we will present the implementation of the Twofish block cipher. For the implementation we will make use of both tricks, mentioned in section 2.1.4. The first one was to precalculate the MDS operation and consequently replace the four applications of the S-boxes and the MDS operation by four table lookups and three xor operations. The second trick was to provide two functions g, which do the table lookups in different orders, to save the 8 bit rotation before the second application of the function g (see figure 2.7). We provide an AVX and an AVX2 implementation of the cipher, which are able to process eight respectively sixteen blocks at once, but as in section 3.1 for the Serpent cipher, the Twofish implementations work on four respectively eight block chucks, too. Only the round key loading is saved by processing two chunks at once. There is no SSE or SSE2 implementation in the Linux kernel available and so we have to write more code from scratch than in the Serpent case, but at least we can get informations from the generic implementation, which is close to the official reference implementation. However there exists a so called 3-way parallel x86_64 assembler implementation, which processes three blocks at once, but in fact they are processed sequentially, i.e. as one block chunk.

```
struct twofish_ctx {
    u32 s[4][256];
    u32 w[8];
    u32 k[32];
};
```

Figure 3.11: Twofish context

```
void __twofish_enc_blk_8way(
  struct twofish_ctx *ctx, u8 *dst,
  const u8 *src, bool xor);
void twofish_dec_blk_8way(
  struct twofish_ctx *ctx, u8 *dst,
  const u8 *src);
```

Figure 3.12: Twofish function prototypes

Again we assume, that the key scheduling process has been successfully finished and the Twofish context is filled with the correct information, needed for the encryption and decryption process. The definition of the context is shown in figure 3.11 and you can see that it already contains all the results from the rather complex key schedule. `s` contains the four 8x32 lookup tables, which are generated from the key dependent 8x8 S-boxes by precalculating the MDS operation like explained in section 2.1.4, `w` contains the eight doublewords of whitening keys, which are used at the beginning and end of the encryption routine and `k` contains the round key material. In each round two doublewords of key material are needed, summing up to 32 doublewords for sixteen rounds. In figure 3.12 you see the function prototypes of the Twofish block cipher. The parameters have exactly the same meaning as they have in the Serpent implementation, and that is why they will not be explained here again.

3.2.1 AVX Implementation

For the AVX implementation of Twofish the registers have been renamed. The definition is shown in figure 3.13. As you can see, we need a lot of general purpose registers compared to the implementation of Serpent. This is because the encryption and decryption routine needs to do the four table lookups and with AVX it is not possible to do packed table lookups. The index used in the lookup has to be stored in a general purpose register. Later you will see how we extract the data from the SIMD registers, do the lookup with general purpose registers and reinsert the data into a SIMD register. `CTX` contains a pointer to the context structure from figure 3.11. The registers `RA1` to `RD1` and `RA2` to `RD2` contain the actual data between the rounds and at the beginning the two four block chunks. `RX` and `RY` are temporary registers used within one round and `RK1` and `RK2` contain the subkey material for the current round. All other registers are used during the table lookup process and are either needed to extract the data from a SIMD register or directly as index register during the lookup.

```
#define CTX %rdi

#define RA1 %xmm0
#define RB1 %xmm1
#define RC1 %xmm2
#define RD1 %xmm3

#define RA2 %xmm4
#define RB2 %xmm5
#define RC2 %xmm6
#define RD2 %xmm7
```

```
#define RX %xmm8
#define RY %xmm9

#define RK1 %xmm10
#define RK2 %xmm11

#define RID1  %rax
#define RID1b %al
#define RID2  %rbx
#define RID2b %bl
```

```
#define RGI1   %rdx
#define RGI1bl %dl
#define RGI1bh %dh
#define RGI2   %rcx
#define RGI2bl %cl
#define RGI2bh %ch

#define RGS1  %r8
#define RGS1d %r8d
#define RGS2  %r9
#define RGS2d %r9d
#define RGS3  %r10
#define RGS3d %r10d
```

Figure 3.13: Twofish register definitions

All operations in the encryption or decryption process are operations on doublewords. The only exception are the four table lookups in the function g, which operate on the single bytes of a doubleword, but we need to handle lookups specially anyway. Therefore it is feasible to substitute the operations on doublewords with packed SIMD instructions on doublewords. This means we are able to do the same input transformation as for the Serpent algorithm, i.e. `RA1` takes the lowest doublewords of all blocks `RA2` the second lowest and so on. The only difference is that we merge the application of the whitening key inside the reading and writing process right before the transposition, which itself is not changed. In the reading process we just have to substitute

```
vmovdqu (0*4*4)(in), x0
```

by

```
vpxor   (0*4*4)(in), wkey, x0
```

to apply the whitening key for a whole block. In the writing process we need two instructions, because the destination of `vpxor` has to be a register operand. Of course we can use the same transformation, we used after reading from memory, just before writing to memory after the encryption or decryption process.

Now as the data is prepared sufficiently, let us have a look at the encryption routine. We have to do sixteen rounds, that means eight cycles. After each round the registers are swapped, because of the Feistel structure. This can be simply achieved by calling the round macro with swapped arguments and so no additional instructions are required for swapping. In figure 3.14 you see the implementation of the encryption cycle and round. The macro `encrypt_cycle` is called directly from the encryption routine with a cycle number between 0 and 7. The macro `encrypt_round` is then called with the current round number and as you can see the arguments are swapped between two subsequent rounds. This way the Feistel structure is implicitly implemented. In every round the two doublewords of round key are loaded with `vbroadcastss` from the Twofish context, described in figure 3.11. After that the actual encryption process is done, i.e. the function F, the two one bit rotations and the xor operations. The two four block chunks are processed sequentially by two subsequent calls to the macro `encround`. The key is used by both chunk operations,

```
#define encrypt_round(n, a, b, c, d) \
    vbroadcastss (k+4*(2*(n)))(CTX),   RK1;           \
    vbroadcastss (k+4*(2*(n)+1))(CTX), RK2;           \
    encround(a ## 1, b ## 1, c ## 1, d ## 1, RX, RY); \
    encround(a ## 2, b ## 2, c ## 2, d ## 2, RX, RY);

#define encrypt_cycle(n) \
    encrypt_round((2*n), RA, RB, RC, RD);       \
    encrypt_round(((2*n) + 1), RC, RD, RA, RB);
```

Figure 3.14: Twofish encryption cycle and round

but the working registers are independent of each other. The macro `encround` implements the function g, applies the Pseudo-Hadamard-Transformation to the first two arguments, adds the round key and then xor's the result to the second two arguments with respect to the two rotations. The registers `RX` and `RY` are used as temporary registers within the macro and do not need to be different for the two block chunks. As the Pseudo-Hadamard-Transformation can be implemented with two packed additions and the addition of the round key and the xor operations are not pretty difficult either, we will just show the implementation of the function g.

As already pointed out, we have to do the table lookups in general purpose registers, because with AVX we have no support of doing packed lookups. The function g is shown in figure 3.15 and takes six arguments. The first argument is the register with the packed input doublewords. For each doubleword we have to do four lookups for each byte of the doubleword. In x we store the result of the combined lookup and MDS operation. The arguments $t_0 \dots t_3$ are parameters, which contain the offsets to the key-dependent precalculated lookup tables in the Twofish context. This way we can permute the tables used for the lookup and save the eight bit rotation, like explained in section 2.1.4. We do not need to provide a second function g, because we can call this generic implementation of g with a different order of the arguments. At the beginning of g we extract the lower and upper 64 bits of the AVX register into the registers `RGI1` and `RGI2`. For each of the two doublewords in `RGI1` and `RGI2` we need to do four lookups on each byte. This is done by the macro `lop`, shown in figure 3.16. The result of the lookup is saved in the registers `RGS1`, `RGS2` and `RGS3`. The registers `RGI1` and `RGI2` are shifted successive by sixteen bits, to extract all the bytes, one after another. After each `lop` call the register has been shifted by 32 bits, i.e. one doubleword, and a new lookup can be started. The results of the operation `lop` are combined to two 64 bit results with simple shift and or operations and finally inserted into the AVX result register x. Therefore the `vpinsrq` operation is used. This operation is rather expensive, but it does not touch the remaining contents of the destination register and is therefore required here.

The operation `lop` does not use any SIMD instructions at all, but just operates on general purpose registers. In fact this operation, together with the g operation, is the core of this implementation. In `lop` we make use of the concatenation feature of the CPP. Because of the general purpose register definitions, shown in figure 3.13 we can access just parts of the 64 bit wide registers by appending a specific suffix with the CPP. `lop` gets a doubleword as input and outputs a doubleword as result. At first the two lowest bytes are extracted and a lookup into the precalculated table is done. You can see that the order of the table arguments decides, in which order the lookups are done. Instead of just doing the lookups and operating on the result, the xor operations have been combined with the lookup. After looking up the first two bytes a 16 bit right shift is done and the same operations are carried out on the two highest bytes. The `movl` instruction sets the upper 32 bit of the destination register implicitly to zero and therefore we just have 32 bit set in the result register and are able to link the result by a simple or operation in the function g.

With this operations the whole encryption routine has been described and we do not need to explain the decryption routine, because due to the Feistel structure exactly the same operations are carried out, just the round keys and the whitening keys have to be applied in different order and the one bit rotations have to be reversed. Of course we are producing a lot of sequential code this way, because for any of the four doublewords in the AVX register, which g gets as input, we call the macro `lop`, which itself does four table lookups. This means in total we are doing sixteen table lookups for one AVX register within the function g and we have no benefit compared to a strict sequential implementation, if we just look at g. However we

```
#define G(a, x, t0, t1, t2, t3)     \
  vmovq           a,    RGI1;       \
  vpsrldq $8,     a,    x;          \
  vmovq           x,    RGI2;       \
  \
  lop(t0, t1, t2, t3, RGI1, RGS1); \
  shrq $16,       RGI1;            \
  lop(t0, t1, t2, t3, RGI1, RGS2); \
  shlq $32,       RGS2;            \
  orq             RGS1, RGS2;      \
  \
  lop(t0, t1, t2, t3, RGI2, RGS1); \
  shrq $16,       RGI2;            \
  lop(t0, t1, t2, t3, RGI2, RGS3); \
  shlq $32,       RGS3;            \
  orq             RGS1, RGS3;      \
  \
  vmovq           RGS2, x;         \
  vpinsrq $1,     RGS3, x, x;
```

Figure 3.15: Twofish function g with AVX

```
#define lop(t0, t1, t2, t3, src, dst)   \
  movb      src ## bl,        RID1b;    \
  movb      src ## bh,        RID2b;    \
  movl      t0(CTX, RID1, 4), dst ## d; \
  xorl      t1(CTX, RID2, 4), dst ## d; \
  shrq $16, src;                        \
  movb      src ## bl,        RID1b;    \
  movb      src ## bh,        RID2b;    \
  xorl      t2(CTX, RID1, 4), dst ## d; \
  xorl      t3(CTX, RID2, 4), dst ## d;
```

Figure 3.16: Twofish AVX table lookup

can parallelize at least the Pseudo-Hadamard-Transformation, the adding of the round keys and the rotations outside of g. Moreover we have a lot of space in registers, when using AVX and can save some memory operations. Because of this advantages, we still get a good speedup compared to an implementation, which does not use SIMD instructions at all, even if the core itself is still implemented sequentially. In section 4.4.2 you will see the detailed evaluation of the speedup we can gain with this implementation and in the next section we will show, how we are able to do it much better and probably faster with the use of parallel table lookups coming with AVX2.

3.2.2 AVX2 Implementation

In the last section you have seen, that the core of the AVX implementation still works sequentially and does not use SIMD registers at all. The AVX2 implementation now overcomes this disadvantages by exploiting the `vpgatherdd` instruction. Moreover the AVX2 implementation processes sixteen blocks at once in two eight block chunks. First of all the register definitions have to be changed. Basically the XMM registers are replaced by the corresponding YMM registers and all the general purpose register definitions have been removed, because they are not necessary any more. There are three new SIMD registers required and we call them `RIDX`, `RLOW` and `RFULL`. `RIDX` will be the new index register for packed table lookups, `RLOW` is a register containing a bitmask with the lowest byte of every doubleword set to all ones and the remaining bits set to zero and `RFULL` is a register that contains just all ones. We will need them all in the new function g. Of course the input and output transformation needs to be adjusted slightly, because now we are reading into the 256 bit wide YMM registers, but as in the Serpent case we do not need to change very much compared to the AVX implementation. The input and output transformation can be left untouched, because it is built out of in-lane operations and the in the reading and writing part it is sufficient to substitute 4 by 8 in the memory operands. The whitening key is xor'ed the same way, but because we now have two blocks in one register, the whitening key has to be doubled as well. Therefore the whitening key is loaded with `vbroadcasti128` to the upper and lower 128 bit of the destination register, instead with `vmovdqu`, like it is done with plain AVX. The remaining parts of the encryption and decryption routine can be left untouched, because they contain operations on packed doublewords and therefore still work with 256 bit wide registers.

So the only big change between AVX and AVX2 is the implementation of the function g. Figure 3.17 shows the AVX2 implementation and as you can see there are no operations on general purpose registers left and the function g is applied to eight doublewords in parallel. Of course the lookups still need to be done with bytes and therefore the mask `RLOW` has been designed. So in the first step the lowest byte of each doubleword is extracted by setting all the upper bytes to zero with the help of `RLOW`. Afterwards the all ones idiom is applied to `RFULL` with `vpcmpeqd`. The register `RFULL` has to be reseted after every

```
#define G(a, x, t0, t1, t2, t3) \
    vpand           RLOW, a, RIDX;                  \
    vpcmpeqd        RFULL, RFULL, RFULL;            \
    vpgatherdd      RFULL, t0(CTX, RIDX, 4), x;     \
    vpsrld $8,      a, RIDX;                        \
    vpand           RLOW, RIDX, RIDX;               \
    vpcmpeqd        RFULL, RFULL, RFULL;            \
    vpgatherdd      RFULL, t1(CTX, RIDX, 4), RIDX;  \
    vpxor           RIDX, x, x;                     \
    vpsrld $16,     a, RIDX;                        \
    vpand           RLOW, RIDX, RIDX;               \
    vpcmpeqd        RFULL, RFULL, RFULL;            \
    vpgatherdd      RFULL, t2(CTX, RIDX, 4), RIDX;  \
    vpxor           RIDX, x, x;                     \
    vpsrld $24,     a, RIDX;                        \
    vpcmpeqd        RFULL, RFULL, RFULL;            \
    vpgatherdd      RFULL, t3(CTX, RIDX, 4), RIDX;  \
    vpxor           RIDX, x, x;
```

Figure 3.17: Twofish function g with AVX2

`vpgatherdd` instruction, because it is cleared by this instruction. After all this preparations the actual table lookup is done by the instruction `vpgatherdd`. It uses the bytes extracted with the help of `RLOW` to do eight lookups in parallel and stores the result in x. After the first step this step is repeated three times for the remaining three lookups. The higher bytes are extracted by first doing a packed logical right shift and applying the mask `RLOW` afterwards. Moreover the result of the lookup is directly xor'ed on the previous results as it has been done in the `lop` macro in the AVX implementation. By the order of the table parameters it is still possible, to determine the order of the lookups. This implementation is much easier, much shorter and much better than the plain AVX implementation.

You have seen that it is not a big step from the AVX implementation to the AVX2 implementation, but the speedup should be quite large. Firstly the AVX2 implementation processes eight block chunks compared to the four block chunks with AVX. Secondly we save a lot of instructions by doing the lookup with one `vpgatherdd` instruction in parallel rather than using a complex macro for each lookup and thirdly we do not need to copy data between SIMD registers and general purpose registers any more. We will have a look at the instruction savings in section 4.4.2, but at this point we can say, that even if the `vpgatherdd` instruction is horribly slow, we should get a significant speedup compared to the AVX implementation.

3.2.3 Kernel Integration

For the AVX implementation of Twofish a kernel patch [25] has been developed as well. The integration into the kernel is very similar to the integration of Serpent and therefore this section will be kept rather short. If you are interested in the details, you may want to have a look at section 3.1.3. We provide the Twofish block cipher as a loadable kernel module with an own entry in Kconfig. The cipher itself has not been modified compared to the AVX implementation, tested in userspace, and is located in `arch/x86/crypto/twofish-avx-x86_64-asm_64.S`. The Makefile had to be adjusted and a rule has been added, which compiles the cipher itself and the gluecode and links the result to the new module. The Kconfig entry was made in `crypto/Kconfig` and the testmanager of the crypto API, located in `crypto/testmgr.c`, has been extended by null tests for the specific synchronous names of Twofish, that are called by the asynchronous implementation, e.g. `__cbc-twofish-avx`.

The gluecode, located in `arch/x86/crypto/twofish_avx_glue.c` looks very similar to the Serpent gluecode and in fact the Serpent gluecode was first designed and then modified to fit for Twofish. The initialization and registering of the algorithms is no different from Serpent. However there is a 3-way parallel implementation of Twofish in the current kernel and therefore we call this implementation, if the message, that has to be encrypted, is too short for our 8-way parallel, but long enough for the 3-way parallel implementation. Only if the message is too short for both implementations the generic implementation will be called. We had to modify the 3-way gluecode in `arch/x86/crypto/twofish_glue_3way.c` as

well, to export the function prototypes of this module, which makes us able to call them from our implementation.

There is one more thing we have done, which we have not done in the Serpent integration. There already has been a 8-way parallel Serpent implementation present in the kernel and therefore the testvectors were large enough, to test 8-way parallel implementations. In the case of Twofish, however, there has just been the 3-way parallel implementation and therefore we had to add tests to `crypto/tcrypt.c`, to be able to test the new code paths with the `tcrypt` module of the crypto API. Moreover we provided new testvectors in `crypto/testmgr.h` that are large enough to test our new code. They were provided in an extra patch [24], because this patch adds many lines and is logical independent of the AVX implementation.

3.3 Blowfish

For Blowfish we provide an AVX and an AVX2 implementation. Currently there are a generic implementation and two assembler implementations provided by the Linux kernel. One assembler implementation processes just one block, like the generic implementation, and the second implementation processes four blocks at once. Our AVX implementation will be able to process sixteen blocks at once in four four block chunks. The AVX2 implementation processes 32 blocks in four eight block chunks. As the block size of Blowfish is just 64 bits this results in a input size of 128 bytes for AVX and 256 bytes for AVX2. This is the same size we got for Serpent and Twofish.

```
struct bf_ctx {
    u32 p[18];
    u32 s[1024];
};
```

Figure 3.18: Blowfish context

```
void __blowfish_enc_blk_16way(
  struct bf_ctx *ctx, u8 *dst,
  const u8 *src, bool xor);
void blowfish_dec_blk_16way(
  struct bf_ctx *ctx, u8 *dst,
  const u8 *src);
```

Figure 3.19: Blowfish function prototypes

In figure 3.18 you see the Blowfish context, which should be filled by the key scheduling algorithm. In `p` the 18 subkeys are stored and `s` contains the four key-dependent 8x32 S-boxes. Figure 3.19 shows the function prototypes for our parallel AVX implementation with the well known arguments. The name indicates, that we want to process sixteen blocks at once.

3.3.1 AVX Implementation

If you have a look at the Blowfish algorithm in section 2.1.5, you can see, that there are not so much operations, which can be done in parallel with AVX. The biggest part of the encryption routine is done by the function F, which consists almost exclusively of table lookups. These lookups have to be done sequentially and only the xor'ing with the round specific subkey can be done in parallel. Moreover we loose speed by copying the data between AVX registers and general purpose registers. The only advantage of implementing Blowfish with AVX, is that we can store a lot of data within registers and therefore save memory operations.

As Blowfish has a block size of 64 bits, the input and output parts have to be modified compared to the previous implementations, but in fact they get easier with the smaller block size. The operations in the Blowfish cipher are all done in big endian mode and therefore the endianness has to be changed during the input and output parts. In section 2.2.2 you have already seen how this is possible with AVX. In this implementation we will make use of the this technique and combine it with the input routine. Figure 3.20 shows the process of transforming the output blocks and writing to memory. The endianness is changed by the `vpshufb` instruction with the register `RMASK`, which is initialized like it has been presented in section 2.2.2. The registers `RR1` to `RR4` and `RL1` to `RL4` contain the actual data. Every register pair contains four blocks, but the blocks are divided into two doublewords and the left and right halves are taken by the RL and RR registers respectively. The transformation, which is necessary, is done by the macro

```
#define transpose_2x4(x0, x1, t0, t1) \
    vpunpckldq            x1, x0, t0; \
    vpunpckhdq            x1, x0, t1; \
    \
    vpunpcklqdq           t1, t0, x0; \
    vpunpckhqdq           t1, t0, x1;

#define outunpack(out, x0, x1, t0, t1) \
    transpose_2x4(x0, x1, t0, t1)      \
    \
    vpshufb RMASK, x0, x0;             \
    vpshufb RMASK, x1, x1;             \
    vmovdqu        x0, (0*4*4)(out);   \
    vmovdqu        x1, (1*4*4)(out);
```

```
outunpack(%rsi, RR1, RL1, RK, RX);
leaq (2*4*4)(%rsi), %rax;
outunpack(%rax, RR2, RL2, RK, RX);
leaq (2*4*4)(%rax), %rax;
outunpack(%rax, RR3, RL3, RK, RX);
leaq (2*4*4)(%rax), %rax;
outunpack(%rax, RR4, RL4, RK, RX);
```

Figure 3.20: Transforming and writing output blocks

`transpose_2x4`, which interleaves the blocks the same way it has already been presented for 128 bit wide blocks. Basically it is the same operation, but because there are just two doublewords, which have to be interleaved, we need just four instructions instead of eight instructions. The registers `RK` and `RX` are used as temporary registers for the transformation. Later on `RK` will store the current round key and `RX` will remain a temporary register. The actual writing to memory is done by the macro `outunpack`, which is called four times, because we want to process four four block chunks. The `leaq` instruction is used to load the address of the position, where we want to write to. The reading part works very similar to the writing part presented here.

In the encryption and decryption routine of the Blowfish algorithm we need to do a lot of table lookups. The general purpose registers used for the lookups are defined exactly like they have been defined for Twofish in section 3.2.1. After the data has been read in and transformed correctly the 16 rounds are performed. Encryption and decryption differ only in the order in which the round keys are used and therefore we can provide one macro, which is used by both methods. In figure 3.21 you see the round macro which is used in the encryption and decryption routine. The registers are supplied as the first two arguments and swapped after each call of the macro. The third parameter is the round number, which goes from 0 to 15 in the encryption routine and from 17 to 2 in the decryption routine. In figure 3.22 the last round of Blowfish is shown, which consists just of the xor'ing with the remaining round keys. In the encryption routine the parameters 16 and 17 are supplied and in the decryption routine the parameters 1 and 0. These macros take care of processing the data in four chunks and load the round key just one time for these four operations.

```
#define round(r, l, n) \
  vbroadcastss (p + 4*(n))(CTX), RK; \
  subround(r ## 1, l ## 1, RX);      \
  subround(r ## 2, l ## 2, RX);      \
  subround(r ## 3, l ## 3, RX);      \
  subround(r ## 4, l ## 4, RX);
```

Figure 3.21: One round of Blowfish

```
#define last(i, j) \
  vbroadcastss (p + 4*(i))(CTX), RK; \
  vpxor           RK, RL1, RL1;      \
  vpxor           RK, RL2, RL2;      \
  vpxor           RK, RL3, RL3;      \
  vpxor           RK, RL4, RL4;      \
  vbroadcastss (p + 4*(j))(CTX), RK; \
  vpxor           RK, RR1, RR1;      \
  vpxor           RK, RR2, RR2;      \
  vpxor           RK, RR3, RR3;      \
  vpxor           RK, RR4, RR4;
```

Figure 3.22: Last round of Blowfish

The macro `subround` does the xor'ing with the round key, processes one half by the function F and xor's the result with the second half. The function F looks in fact exactly like the function g for Twofish and therefore we will not show the implementation here. As we have just one function F, we do not need to change to order of the S-boxes, like we did it for Twofish and therefore we do not need the four S-box parameters. There are just two differences between the implementation for Twofish and the one for Blowfish. First of all the lookups for Blowfish have to be done in big endian order, i.e. the most significant byte has to be processed first. Therefore we add another shuffle instruction at the beginning of the function F, which changes the endianness of all doublewords at once. The second difference is that we need different

operations in the macro `lop`. So instead of using three times `xorl`, we now need to use first `addl` then `xorl` and finally `addl`. Both differences are just two small changes, but the rest of the already existing implementation can be reused for this purpose.

You have seen that it is not very difficult to implement Blowfish, if an implementation for Twofish is already existing. Most of the parts get easier, because of the smaller block size and the pure Feistel structure. However there are not much parts, which get done in parallel and you will see in section 4.4.3, that consequently the speedup is not very good either. Nevertheless the AVX implementation of Blowfish is a good basis for the AVX2 implementation, which should do things much better.

3.3.2 AVX2 Implementation

Implementing Blowfish with AVX2 is similar to the AVX2 implementation of Twofish. The data reading and writing parts have to be adjusted, but as it was the case for the larger block size, the input and output transformation can be reused because of the in-lane operations. Basically it is sufficient for the reading and writing parts, to substitute the current displacement values by the appropriate new values, e.g. to exchange 4 by 8. The changing of the endianness has to be adjusted as well, but this works exactly like it was presented in section 2.2.3, i.e. the `vpshufb` instruction can be left untouched and the mask has to be broadcasted to both lanes of the register using the `vbroadcasti128` instruction.

The big change is in fact the function F. As it was the case for the Twofish implementation, the table lookups are now done in YMM registers and no general purpose registers are needed. The new function F looks very similar to the function g presented in section 3.2.2. The same registers are used and the only differences are, that the bytes are looked up in the reverse order, i.e. the register is first shifted by 24 bits and not by 8 bits, and that the operations have been replaced. Instead of using three times `vpxor`, we now use first `vpaddd`, then `vpxor` and finally `vpaddd` again. The operations, which were done sequentially in the AVX implementation are now transformed to their SIMD equivalents.

As the registers are now 256 bit wide, instead of just 128 bit, the AVX2 implementation processes 32 blocks at once in four eight block chunks. This are in total 256 bytes. We save a lot of instructions compared to the AVX implementations, because the lookups are not sequential anymore. Consequently this implementation should be faster than the plain AVX implementation. The detailed instruction evaluation will be shown in section 4.4.3.

3.3.3 Kernel Integration

Even tough the speedup for the AVX implementation of Blowfish is not really worth the work, a kernel patch has been developed, just to test the implementation in kernel space. This patch has not been submitted, however, because probably no one would use this implementation, but is has some educational meaning and parts of this patch can be reused by the patch for Cast-128. This section will be kept rather short, because once the parallel assembler implementation of a cipher is ready the patching of the kernel always is similar for different ciphers.

The architecture specific Makefile has to be adjusted and an entry for the new cipher has been added. A new entry in the crypto Kconfig file has been added, too. In the `tcrypt` module new functions for testing the different modes have been added and the null tests have been added to the testmanager. The gluecode of the existing assembler implementations has been modified to export the functions, these modules provide and to be able to use them within our new implementation, if the supplied block is too small for being processed by our 16-way parallel implementation. Besides the actual implementation, which has been added to the architecture specific crypto directory, most changes go to the new gluecode needed to integrate our cipher. It is by far fewer code, than for Serpent and Twofish, because only the modes ECB, CBC and CTR are supported. The modes LRW and XTS just work with 128 bit wide block ciphers and therefore cannot be used with Blowfish. So the new gluecode provides support for these three modes of operation and calls the 4-way parallel or the one block implementation, if the block is too small for our implementation.

3.4 Cast-128

After Blowfish another cipher with a block size of 64 bits has been implemented. For Cast-128 an AVX and an AVX2 implementation is provided as well. The AVX implementation processes sixteen blocks at once in four four block chunks and the AVX2 implementation takes 32 blocks at once in four eight block chunks. This are the same values as for the Blowfish cipher. In the Linux kernel there is currently just a generic implementation of the Cast-128 cipher, written in C, and therefore we have to do a little bit more work for the kernel integration. The speedup compared to the generic implementation, however, is not bad in return.

```
struct cast5_ctx {
    u32 Km[16];
    u8  Kr[16];
    int rr;
};
```

Figure 3.23: Cast-128 context

```
void __cast5_enc_blk_16way(
  struct cast5_ctx *ctx, u8 *dst,
  const u8 *src, bool xor);
void cast5_dec_blk_16way(
  struct cast5_ctx *ctx, u8 *dst,
  const u8 *src);
```

Figure 3.24: Cast-128 function prototypes

Figure 3.23 shows the Cast-128 context, which is initialized by the key scheduling algorithm. `Km` takes the sixteen masking keys and `Kr` contains the sixteen key-dependent rotation values, which are used in each round. The field `rr` is a flag, which indicates how many rounds should be processed. If `rr` equals one, the cipher does only twelve rounds, and if `rr` equals zero, sixteen rounds are performed. The value `rr` is initialized in the key schedule according to the length of the user supplied key and has to be checked in each encryption or decryption routine. In figure 3.24 the well known function prototypes of our parallel AVX implementation are listed.

3.4.1 AVX Implementation

The reading from and writing to memory part as well as the transformation in the AVX implementation is identical to the description from the Blowfish cipher above. After the data has been read in and transformed, the twelve respectively sixteen rounds in the encryption routine follow according to the flag `rr`. If `rr` has a non-zero value, the last four rounds are simply skipped. In the decryption routine the first four rounds are skipped, if a non-zero value is supplied.

The `round` macro is invoked with four parameters. In figure 3.25 you see the implementation of this macro. The registers containing the two halves of the blocks are supplied as first parameters and swapped between each invocation of the `round` macro. The third parameter is the current round number and the fourth parameter is the number of the function, which should be used in this round. The functions have already been shown in figure 2.8 and the numbers 1 to 3 are supplied circularly. The first thing which is done in a new round is to broadcast the round specific masking key into the register `RKM`. Next the rotation value is loaded to the lowest byte of the register `RKRF`, which is zero initialized at the beginning of the encryption routine. The rotation value is one byte wide, but just the lowest 5 bits are allowed to have a non-zero value. Nevertheless we need to store this value in a 128 bit wide register, because this way we are able to do packed rotations on four doublewords at once. Because there is no instruction, which is able to do packed rotations, we have to emulate this rotation by a logical left shift, a logical right shift and one bitwise or operation. Therefore we need the value $32 - Kr_i$ for the right shift. This value is computed and stored in the register `RKRR`, with the help of the mask register `R32`, which just contains the value 32 in the lowest byte of the register and is otherwise set to all-zero. The subtraction performed is actually a packed subtraction on quadwords, and so the upper half of the register `RKRR` contains the value 32 instead of zero after this operation. However for the shift operation only the lower 64 bits of a XMM register are used according to the Intel 64 and IA-32 Architectures Software Developer's Manual [33] and therefore this wrong value does not matter.

After initializing the masking key register and the values needed for the key-dependent rotation the macro `subround` is called for every of the four block chunks. This macro invokes the corresponding function

```
#define F(a, x, op0, op1, op2, op3) \
    op0     a,      RKM,  x;    \
    vpslld  RKRF,   x,    RTMP; \
    vpsrld  RKRR,   x,    x;    \
    vpor    RTMP,   x,    x;    \
 .....

#define F1(b, x) F(b, x, vpaddd, xorl, subl, addl)
#define F2(b, x) F(b, x, vpxor,  subl, addl, xorl)
#define F3(b, x) F(b, x, vpsubd, addl, xorl, subl)

#define subround(a, b, x, n, f) \
    F ## f(b, x);  \
    vpxor a, x, a;

#define round(l, r, n, f) \
    vbroadcastss    (km+(4*n))(CTX), RKM;        \
    vpinsrb $0,     (kr+n)(CTX),     RKRF, RKRF; \
    vpsubq          RKRF,            R32,  RKRR; \
    subround(l ## 1, r ## 1, RX, n, f);          \
    subround(l ## 2, r ## 2, RX, n, f);          \
    subround(l ## 3, r ## 3, RX, n, f);          \
    subround(l ## 4, r ## 4, RX, n, f);
```

Figure 3.25: Cast-128 round processing

`F1`, `F2` or `F3` and xor's the result to the left halves of the blocks. As you have already seen in section 2.1.6 the three functions do not differ in structure, but only in the operations used and therefore for all three functions there exists just one macro, which gets the operations, that need to be performed, as arguments. In figure 3.25 you also see the first part of this generic function F. First the specific operation is carried out on the input doubleword and the masking key and afterwards the rotation is performed by the two shifts and the or operation. For this one temporary register `RTMP` is needed and instead of immediate values the registers `RKRF` and `RKRR` are supplied as shift values. The rest of the function F looks like the function F from the Twofish implementation, but the macro `lop` gets the remaining three operations as arguments. Instead of the three `xorl` operations, `lop` now uses the instructions, specified by `op1`, `op2` and `op3`. The remaining parts of the `lop` macro, however, are identical to the definition in the Twofish implementation, because there are also four 8x32 table lookups performed, as you have seen in the high level description of f_1, f_2 and f_3 in section 2.1.6. The four S-boxes, however, are not part of the context, like it was the case for Twofish and Blowfish, but instead are fixed and exported from the generic implementation. That is why the first operand of a table lookup operation has to be changed. So instead of supplying

```
s1(CTX, RID1, 4)
```

we now just have to supply

```
s1(, RID1, 4)
```

because the displacement value s1 is now an absolute value and the context must not be used as base for this operation. Of course the name `s1` has not been used to export the first S-box from the generic module, because exporting this generic name for the whole kernel would be bad, but instead the name `cast5_s1` has been exported and `s1` has just been defined for this assembler file.

With the explanation of the three functions `F1`, `F2` and `F3` the encryption routine is finished and the decryption routine uses the same macros as the encryption routine, because it is a pure Feistel network. The round numbers are supplied in the reverse order as well as the circular numbers 1 to 3 to the `round` macro, but everything else remains the same. Of course there are not very much operations, which are done in parallel either, because the table lookups are still sequential, but it are more operations, than it were for Blowfish. Not only the xor'ing with the result of the function F, but also the operation with the round specific masking key and the key-dependent rotation is done in parallel on the four doublewords. In addition there are two values which are preserved across the invocation of the macro `subround` for all four chunks. In section 4.4.4 you will see how this implementation performs compared to the generic implementation, which is already present in the Linux kernel.

3.4.2 AVX2 Implementation

After the detailed description of the AVX implementation, the AVX2 implementation can be described in few words. Again the reading and writing parts have to be modified according to the larger register sizes, but there is nothing special about this change, which would be worth to be mentioned here. The overall structure of the AVX2 implementation remains the same as well, but it has to be noted that the registers, used as shift operands remain XMM registers, while all other registers are converted to YMM registers. This is no problem, since only the lowest 64 bit are taken as shift value anyway.

The only big difference between AVX and AVX2 is again the implementation of the generic function F. The conversion is done exactly the same way, like it has been done for Twofish, meaning that the table lookups now are done in YMM registers instead of general purpose registers. Of course the operands for the `vpgatherdd` instruction do not have the `CTX` base, for the same reason it has been shown in the AVX implementation of Cast-128. The converted function F takes the same four additional parameters as in the AVX case, but every parameter is converted to its SIMD equivalent, because all operations are now done in parallel on four doublewords at once. So instead of defining

```
#define F1(b, x) F(b, x, vpaddd, xorl, subl, addl)
```

we now define

```
#define F1(b, x) F(b, x, vpaddd, vpxor, vpsubd, vpaddd)
```

With this conversion every operation in the Cast-128 algorithm is done in parallel and that is why the AVX2 implementation should perform better than the AVX implementation. We are able to save the usage of the general purpose registers and the copying between SIMD and general purpose registers. In section 4.4.4 we will have a look at the instructions that can be saved by this technique.

3.4.3 Kernel Integration

The kernel integration of the Cast-128 cipher requires a little bit more work to be done, because at the moment there is just a generic implementation provided by the Linux kernel, which is written in C. The kernel integration of Cast-128 is done with three separate patches. The first patch [18] prepares the generic module for optimized implementations. The second patch [22] adds larger testvectors for the Cast-128 algorithm and modifies the `tcrypt` module to support new tests for the accelerated cipher. The third and final patch [17] adds the actual parallel AVX implementation and integrates it properly.

The module, which contains the generic Cast-128 implementation, has been renamed from `cast5` to `cast5-generic` and the alias `cast5` has been set for the new module name. After that the module may still be loaded by the name `cast5`, but additional modules may be defined, which share the same alias. The context declaration and the definition of the relevant constants of the algorithm, i.e. block size, minimal key size and maximal key size, as well as the function prototypes for the generic encryption, decryption and key scheduling routines have been moved to a new header file in `include/crypto/cast5.h`. Furthermore the four fixed 8x32 S-boxes needed in the encryption and decryption routine have been exported out of the module, as well as the generic encryption, decryption and key scheduling functions. All this components are used by our module, which we will provide with the third patch.

The second patch is very huge, because of the large testvectors, that are introduced with this patch. At the moment there are just testvectors for ECB, taken from the official RFC [1], present in the testmanager and therefore we have to generate new and long enough testvectors for ECB, CBC and CTR. To achieve this, an own standalone kernel module has been written, called `testv`, which processes a given input by a specific cipher with a specific mode and writes the result to the kernel log. The operation of this module is very similar to the operation of the `tcrypt` module itself. After that all the large existing testvectors from Twofish have been taken and processed by `testv` with the generic implementation of Cast-128. The resulting data together with the known inputs has been supplied as new testvectors for the Cast-128 algorithm to the testmanager. In addition to the testvectors this patch also adds the tests for the modes CBC and CTR to the testmanager, because just the ones for ECB were present. Furthermore the performance

tests for the synchronous and asynchronous version of the algorithm with all supported modes were added to the `tcrypt` module.

The third and last patch contains the actual implementation. The architecture specific Makefile is adjusted to be able to compile the new module and the crypto Kconfig has been modified with a new entry for the implementation. The implementation itself is created in the architecture specific crypto directory together with the gluecode, which is necessary to register the new cipher within the crypto API. During this work the crypto API changed slightly and there are new helper modules available, which simplify the integration a little bit. The new gluecode makes use of this helper modules and is therefore shorter, than the gluecode for the three algorithms, already presented. However there are no fundamental, but rather syntactic changes. In addition to the real tests for the common name, which have already been added with the second patch, this patch also adds the nulltests for the specific names of the new implementation to the testmanager.

3.5 Cast-256

The last cipher, for which we provide an implementation, is Cast-256, the successor of Cast-128. In contrast to Blowfish and Cast-128, Cast-256 has a block size of 128 bit again. The AVX implementation will be able to process eight blocks at once in two four block chunks and the AVX2 implementation will be able to process sixteen blocks at once in two eight block chunks. Again this results in sizes of 128 bytes for the AVX implementation and 256 bytes for the AVX2 implementation. The Linux kernel contains just a generic implementation of Cast-256 and that is why as much work for the kernel integration is necessary, as it has been for Cast-128. Fortunately the whole process is somewhat analogous to the integration process of Cast-128.

```
struct cast6_ctx {
    u32 Km[12][4];
    u8 Kr[12][4];
};
```

Figure 3.26: Cast-256 context

```
void __cast6_enc_blk_8way(
  struct cast6_ctx *ctx, u8 *dst,
  const u8 *src, bool xor);
void cast6_dec_blk_8way(
  struct cast6_ctx *ctx, u8 *dst,
  const u8 *src);
```

Figure 3.27: Cast-256 function prototypes

In figure 3.26 you see the context of Cast-256, which is initialized by the key scheduling algorithm. `Km` contains the masking keys for the 48 rounds, also referred to as six quad-rounds and six reverse quad-rounds, and `Kr` is filled with the 48 rotation values. There is no `rr` flag value, like in the Cast-128 algorithm, because the number of rounds is independent of the user supplied key size and so no flag has to be checked in the encryption and decryption routine. In figure 3.27 the function prototypes for the AVX implementation are listed and the name indicates that the implementation just processes eight blocks and not sixteen blocks.

3.5.1 AVX Implementation

The reading and writing parts of this implementation are similar to those of the Twofish implementation, because both implementations have the same block size and work on 32 bit doublewords. The only difference is that the endianness has to be changed in the Cast-256 reading and writing part. This is achieved with the `vpshufb` instruction, which is applied on the four registers that are read from or written to memory. The remaining parts, e.g. the transformation of the input or output blocks, are exactly the same, as the ones for the Twofish implementation.

After the blocks have been read in and transformed, the six quad-rounds and the six reverse quad-rounds are performed. The only parameter which is supplied to the `Q` or `QBAR` macro is the current quad-round number. In the encryption routine the number goes from 0 to 11 and in the decryption routine the number goes from 11 to 0. In both cases first the macro `Q` is called six times and afterwards the macro `QBAR` is called six times. The high-level description of the Q and $\overline{Q}$ function has already been shown in figure 2.9. The implementation is really close to this high level description. Figure 3.28 shows the first part of the macro `Q`,

```
#define qop(in, out, x, f) \
    F ## f(in ## 1, x);          \
    vpxor out ## 1, x, out ## 1; \
    F ## f(in ## 2, x);          \
    vpxor out ## 2, x, out ## 2; \

#define Q(n) \
    vbroadcastss    (km+(4*(4*n+0)))(CTX), RKM;         \
    vpinsrb $0,     (kr+(4*n+0))(CTX),     RKRF, RKRF; \
    vpsubq          RKRF,                  R32,  RKRR; \
    qop(RD, RC, RX, 1);                                \
 .....
```

Figure 3.28: Part of the Cast-256 macro `Q`

which implements the function Q. The round specific masking key and the rotation value is loaded the same way, they were loaded in the Cast-128 implementation. Again we need the value $32 - Kr_i$ to be able to emulate the key-dependent rotation by two shift operations and one or operation. In every invocation of the macro `Q`, four masking keys and four rotation values are needed. That s why it is called a quad-round. After these two values have been loaded correctly the macro `qop` is called, which is responsible for processing both four block chunks with the corresponding function. The functions f_1, f_2 and f_3 are implemented like it has been shown for Cast-128 and therefore their implementation is not listed in figure 3.28. After the call to `qop` the same procedure is repeated three more times according to the definition in figure 2.9.

The macro `QBAR` looks similar to the macro `Q`, but the order of the loading of the masking keys and the rotation values is exchanged as well as the arguments to the `qop` macro. This becomes quite clear, if you have a look at the definition of the function $\overline{Q}$ in figure 2.9, as the operations are just reversed. After the six quad-rounds and the six reverse quad-rounds the encryption or decryption routine is finished. Unfortunately there remain the parts of the functions f_1, f_2 and f_3, which have to be processed sequentially, but nevertheless we get a good speedup compared to the generic implementation with this approach. The detailed evaluation will be shown in section 4.4.5.

3.5.2 AVX2 Implementation

The AVX2 implementation processes sixteen blocks at once in two eight block chunks. They are read in like in the AVX2 implementation of Twofish. The changing of the endianness is done like in the AVX implementation, just the mask has to be broadcasted to both lanes of the register `RMASK`. The rest of the implementation does not change very much compared to the AVX implementation as well. Of course all registers, except from the registers responsible for the key-dependent rotations, are converted to YMM registers. In the encryption and decryption routine the macro `Q` and `QBAR` is still called six times and the implementation of these two macros does not change very much either.

As it was already explained for Cast-128 the generic function F, which is exactly the same implementation as for Cast-128, is converted to do the table lookups in YMM registers instead of general purpose registers. The definition of the macros `F1`, `F2` and `F3` is done the same way, it was done in the AVX2 implementation of Cast-128. To keep things short it can be said that relevant changes from the AVX implementation to the AVX2 implementation of Cast-256 are just needed in the parts, which are shared between the Cast-128 and the Cast-256 implementation and therefore it does not make sense, to show them again. With this changes we eliminate all sequential parts and replace them by parallel operations with SIMD registers. The AVX2 implementation is therefore able to save a lot of instructions.

3.5.3 Kernel Integration

It has already been mentioned that the kernel integration of Cast-256 is somewhat analogous to the kernel integration of Cast-128. Another three patches are needed to integrate Cast-256 into the kernel. In fact the integration of Cast-128 and Cast-256 has been provided as one six patches large patchset, because there are

files, which need to be modified for both implementations and therefore the patches are kind of dependent onto each other. For example we need to add the testvectors for both ciphers to the testmanager or provide new tests in the `tcrypt` module. So the first three patches of the patchset are regarding the Cast-128 implementation and have already been covered in section 3.4.3 and the second three patches are regarding the Cast-256 implementation and are covered in this section.

The first patch [20] prepares the generic implementation of Cast-256 to be able to work with optimized implementations. The kernel module `cast6` is renamed to `cast6-generic` and the S-boxes are exported as well as the generic encryption, decryption and key scheduling functions to be able to use them in our new implementation. The context declaration has been moved to a new header file, which is located at `include/crypto/cast6.h`, together with the definition of the relevant algorithm constants and the function prototypes of the generic implementation. The cipher priority of the generic implementation has been set to 100, because our implementation will use priority 200 and is therefore preferred by an application using the crypto API.

The second patch [23] adds the long enough testvectors for the Cast-256 algorithm as well as the tests for the testmanager and the performance tests for the `tcrypt` module. As Cast-256 has a block size of 128 bits we need to provide testvectors for all five modes of operation, i.e. ECB, CBC, CTR, LRW and XTS. All these testvectors were generated from the Twofish testvectors with our standalone kernel module `testv`. This way it is possible to test the parallel code paths, which our new implementation introduces.

The third and final patch [19] adds the actual implementation. The architecture specific crypto Makefile as well as the crypto Kconfig file have been adjusted to be able to select and compile the new cipher. Furthermore some more nulltests had to be added to the testmanager in addition the real tests added with the last patch. The actual implementation has been copied to the architecture specific crypto directory and a new gluecode file has been created. The gluecode now is able to work with all five modes and registers the synchronous and asynchronous implementation within the crypto API. As the new helper modules are available now, the gluecode is simpler than the one for Serpent and Twofish. The details however should be looked up in the official patch, because on the one side this is a lot of code, but on the other side there are no new concepts that are introduced. The code is just necessary for a proper kernel integration.

4 EVALUATION

In this chapter we want to take a different look on the implementations, which were produced during this thesis. In chapter 3 you have seen, how the ciphers are actually implemented, and in this chapter we want to discuss the impact of these implementations. In the last chapter many details of the specific implementations have been presented and now we will give at first in section 4.1 an overview on the changes on a larger scale. For example you will see what parts of the kernel have been changed in what matter. In section 4.2 we will give some short explanations on what architectures and with which software our implementations can be used.

Up to this point we have just claimed that our implementations work correct and as intended, but in section 4.3 we will explain why this claim is plausible and show some methods to verify the implementations. We will work with methods provided by the crypto API of the Linux kernel, but do userspace tests as well as automated filesystem tests. Once the implementation is registered within the crypto API of the kernel, there are many ways to use and test it.

The most important section of this chapter is section 4.4. As the goal of this thesis is to accelerate cryptographic primitives, the performance evaluation plays an important role. We will show how fast our implementations are and give real time measurements, whenever possible. For AVX2 we cannot do real measurements by now, but at least we can count instructions and make some educated guess about the possible speedup. Once again the Intel Software Development Emulator will help us with this task. The AVX implementation, however, can be tested in user- and kernelspace. We will show the different modes of operation and compare our implementation to the present implementations in the Linux kernel. Moreover we provide filesystem tests with the device mapper, to test, whether we can get a higher data rate with our accelerated implementations. Of course the data rate is not just influenced by the chosen cipher, but by a whole bunch of other factors. Nevertheless we try to get as accurate data as possible by minimizing the other influences.

Last but not least some considerations about the security of the implementations and ciphers in general are made. Of course security is an important issue besides correctness and performance and that is why we should at least have a short look at this topic. As our implementations are just accelerated versions of already existing ciphers, they should be as secure or as vulnerable as the existing implementations.

4.1 Source Code Statistics

In this section we want to give an overview on what files have changed and how extensive the changes are. We will focus on the kernel patches, which have been submitted, because the userspace setup, which has been used to develop the ciphers itself is not very interesting. Moreover the implementation of the cipher itself has not been changed between the userland implementation and the kernel integration.

Table 4.1 gives an overview of the Serpent kernel patch [21]. For every file, which has been modified with this patch, it is listed, whether the file was newly created and how many lines have been changed. Furthermore you see, what percentage, compared to the whole patch, the changes in a specific file have. You see, that the changes in the Makefile and the Kconfig file are very simple and short. The null tests added to the testmanager are not a big change either. So most of the changes come from the two new files, which are the actual implementation of the parallel block cipher and the gluecode, needed to register the cipher within the crypto API. Unfortunately the gluecode is in fact more code than the actual implementation, although of course not so complicated as the implementation, because writing C is in fact easier than writing assembler.

File	New	Lines	%
`arch/x86/crypto/Makefile`		2	0.11
`arch/x86/crypto/serpent-avx-x86_64-asm_64.S`	✓	704	40.58
`arch/x86/crypto/serpent_avx_glue.c`	✓	949	54.70
`crypto/Kconfig`		20	1.15
`crypto/testmgr.c`		60	3.46
Total		1735	100.00

Table 4.1: Source code statistics of the Serpent kernel patch

Table 4.2 shows you the statistics of the Twofish kernel patches [25]. The situation is similar to the Serpent patch, but some more files had to be adjusted. The changes in the Makefile and Kconfig file are rather short again, just as the null tests added to the testmanager. The new and large enough testvectors for an 8-way parallel Twofish implementation have been added to `crypto/testmgr.h` and were provided as an extra patch [24]. This is a rather big change, but not very complicated, because actually there has been no code added to the file, but just data. The new tests have been added to the `tcrypt` module, to be able to use that module with the new cipher and this change is rather short as well. Another small change was made in the 3-way parallel implementation to make it possible to use this implementation within our implementation. So apart from the adding of the testvectors, which is an extra patch, the biggest changes are again the actual implementation and the gluecode. The implementation of Twofish is much shorter than the Serpent implementation, because a lot of stuff is done with table lookups and there is no need to provide many S-boxes as logical sequences. In this patch you clearly see how much gluecode is necessary to be able to use the accelerated cipher. The gluecode is even more code compared to the Serpent gluecode, because in corner cases we call the 3-way parallel implementation and therefore we have to write some more lines.

File	New	Lines	%
`arch/x86/crypto/Makefile`		2	0.08
`arch/x86/crypto/twofish-avx-x86_64-asm_64.S`	✓	301	12.41
`arch/x86/crypto/twofish_avx_glue.c`	✓	1086	44.77
`arch/x86/crypto/twofish_glue_3way.c`		2	0.08
`crypto/Kconfig`		24	0.99
`crypto/tcrypt.c`		23	0.95
`crypto/testmgr.c`		60	2.47
`crypto/testmgr.h`		928	38.25
Total		2426	100.00

Table 4.2: Source code statistics of the Twofish kernel patches

In table 4.3 you see the statistics of the Blowfish kernel patch. This patch has not been submitted, because the speedup is not very good as you will see in section 4.4.3. This is the reason why the testvectors have

not been expanded with an extra patch. As you can see most of the lines are needed for the gluecode and the implementation itself. The existing gluecode for the 1-way and 4-way assembler implementation had been adjusted slightly to export the functions this module provides. The remaining changes are just used to integrate the implementation into the build system and the `tcrypt` module.

File	New	Lines	%
`arch/x86/crypto/Makefile`		2	0.17
`arch/x86/crypto/blowfish-avx-x86_64-asm_64.S`	✓	295	25.37
`arch/x86/crypto/blowfish_avx_glue.c`	✓	768	66.04
`arch/x86/crypto/blowfish_glue.c`		4	0.34
`crypto/Kconfig`		19	1.63
`crypto/tcrypt.c`		15	1.29
`crypto/testmgr.c`		60	5.16
Total		1163	100.00

Table 4.3: Source code statistics of the Blowfish kernel patch

Table 4.4 is a summary of the three kernel patches [18] [22] [17], that integrate the AVX implementation of the Cast-128 cipher. The testvectors added to the testmanager make up the biggest part of the changes. The gluecode now needs fewer lines, because of the new helper modules that have been introduced, while working on this thesis, but it is still the biggest code change in the whole patchset. You see the renaming of the generic implementation as well as the changes to the `tcrypt` module. In addition to the new implementation and the gluecode a new header file has been created, that takes the common declarations and definitions, which can be shared between all implementations. You see that there are some more changes needed, if just a generic implementation is available.

File	New	Lines	%
`arch/x86/crypto/Makefile`		2	0.11
`arch/x86/crypto/cast5-avx-x86_64-asm_64.S`	✓	323	16.96
`arch/x86/crypto/cast5_avx_glue.c`	✓	530	27.82
`crypto/Kconfig`		14	0.73
`crypto/Makefile`		2	0.11
`crypto/{cast5.c -> cast5_generic.c}`		79	4.15
`crypto/tcrypt.c`		32	1.68
`crypto/tcrypt.h`		1	0.05
`crypto/testmgr.c`		90	4.72
`crypto/testmgr.h`		810	42.52
`include/crypto/cast5.h`	✓	22	1.15
Total		1905	100.00

Table 4.4: Source code statistics of the Cast-128 kernel patches

The source code statistics for Cast-256, the last cipher, for which we provide an implementation, are shown in table 4.5. They look similar to the statistics for Cast-128. As now testvectors for all five modes have to be submitted, this gets a very huge change. In fact one half of all lines, that have been changed, are related to testvector data. The actual implementation and the gluecode are the biggest code changes, but if you compare the gluecode size to the gluecode size of Twofish, you can see that the helper modules are really useful in reducing gluecode size. The rest of the changes is pretty standard. The `tcrypt` module and the testmanager is modified and the new header file for the Cast-256 algorithm is created. If we look at these changes as one entity, these are the biggest changes, that have been required to integrate a cipher into the kernel so far [20] [23] [19].

Concluding this section one can say that adding a cipher to the Linux kernel is in fact a local change. All changes go either to the common crypto part or to the architecture specific crypto part. Integrating a cipher into the Linux build system is very easy, because there are only very small changes needed. However currently it is necessary, to add a lot of gluecode, if you want to add a new cipher. This causes code

File	New	Lines	%
arch/x86/crypto/Makefile		2	0.07
arch/x86/crypto/cast6-avx-x86_64-asm_64.S	✓	336	12.55
arch/x86/crypto/cast6_avx_glue.c	✓	648	24.21
crypto/Kconfig		17	0.64
crypto/Makefile		2	0.07
crypto/{cast6.c -> cast6_generic.c}		67	2.50
crypto/tcrypt.c		50	1.87
crypto/testmgr.c		120	4.48
crypto/testmgr.h		1412	52.75
include/crypto/cast6.h	✓	23	0.86
Total		2677	100.00

Table 4.5: Source code statistics of the Cast-256 kernel patches

duplication, increases object size and makes maintenance more difficult. As it is conceivable that at the latest with AVX2 more accelerated ciphers will be added to the crypto API, there currently are efforts to extract common parts of the gluecode. During the work on this thesis there have already been some helper modules introduced, which make it possible to reduce the gluecode size. This is clearly visible if we compare the integration patch of Twofish with the patchset for Cast-256. Of course there still needs to be a cipher specific gluecode file, but by changing the API slightly, it would probably be possible to decrease the lines of code necessary for this specific gluecode file by another few lines.

4.2 Compatibility

In this section we want to make some considerations under what circumstances it is possible to use our accelerated implementations. There are two major requirements that need to be met. Of course AVX is architecture specific and so it is not possible to use these implementations with every processor on the market. As already mentioned in section 2.2 the first processors, that support AVX were Intel's Sandy Bridge processors, first shipped in Q1, 2011. Since 2012 Intel provides a new microarchitecture, called Ivy Bridge. All processors based on this architecture are capable of working with AVX instructions as well. For example Core i3, Core i5 and Core i7 are processors that are based either on the Sandy Bridge or the Ivy Bridge microarchitecture. AMD provides the Bulldozer processors since Q3, 2011 and they are able to deal with AVX instructions, too. They have also been already introduced in section 2.2. So if you want to use the accelerated implementations, produced during this thesis, you currently need a processor that falls in one of the three categories mentioned here.

Besides the hardware requirement, there is also a software requirement. Although the implementations have been developed at first in userspace, you have seen in the last section, how much gluecode is necessary to actual use the implementations. Of course much of this code is Linux specific, but if you would want to use this implementations directly in an userspace application, there would be still some gluecode to write. Currently the implementations are provided as Linux kernel patch and so the implementations can be used with the crypto API of the kernel. This means, it is possible to use the implementations everywhere inside the kernel and with all components supporting the crypto API. The most notable use case is probably disk encryption. For example the device mapper dm-crypt supports all ciphers, the crypto API provides, to create crypto containers upon block devices or simple files. This way it is possible to use our implementations for disk encryption out of the box just by installing a new kernel.

Since kernel version 2.6.38 it is possible to use algorithms, the crypto API of the kernel provides, from userspace in standard applications. Therefore a netlink-based interface, that adds a new interface family, called AF_ALG, has been introduced [13] and integrated into the kernel [60]. Now it is possible to use at least hash algorithms and ciphers from the crypto API in standard applications with the help of sockets. However this interface has been mostly designed to give applications the possibility to exploit the performance, which is provided by hardware crypto devices, which would otherwise be just accessible by the

kernel itself. Some people might see it as bad practice to force the kernel to do rather expensive computations, if the algorithm is just a software implementation and could be done in userspace as well. There exists another interesting patch [39], which makes it possible to use algorithms from the crypto API in userspace with the help of a block device called `/dev/crypto`. Therefore the application needs to do specific `ioctl` calls on this block device to fore the kernel to encrypt or decrypt data. The idea was taken from OpenBSD's cryptodev userspace API and is similar to that API. The maintainers of this patch, called CryptoDev, claim, that it is faster than the current implementation of `AF_ALG` in the kernel, but nevertheless this patch is not available in mainline.

Concluding this section, it can be said, that our implementations can be used on any new enough consumer processor with x86_64 architecture, as they probably will have support for AVX. The implementations can be used anywhere inside the kernel with components using the crypto API and with the `AF_ALG` interface it is even possible to use them from userspace without patching the kernel.

4.3 Correctness

Before we have a look at the performance of the implementations, we want to show that our implementations work correct and as intended. We will not give a real proof for correctness here, because this would not be trivial. Instead we rely on black box testing, which is an accepted method in cryptography. Basically there are two possibilities, how we can test our implementations. On the one side we have existing implementations of the ciphers, which made it into the official Linux kernel, and we are able to compare the results of this implementations with the results from our implementations. Of course, if we are choosing this method, we implicitly suppose the existing implementations to be correct. On the other side, there are usually official testvectors published along with the specification and we can use this testvectors, which consist of a specific key, a defined input and a defined output, to test our implementations and verify that they produce the same output on the defined inputs.

First the block ciphers were developed in userland and tested in ECB mode with the help of a small program, which takes data on `stdin`, encrypts or decrypts the data according to a flag, supplied at compile time, and writes the result to `stdout`. With this setup it is easy to check, whether the cipher produces a specific result. Let us assume, that we have a file called `in` with some plaintext and a file called `out` with the corresponding ciphertext, produced with some different implementation. A line like the following one is sufficient for testing the new implementation:

```
sde -- ./serpent-avx2 < in | diff - out
```

As you can see, with the help of the Intel Software Development Emulator it is possible to check with this simple line the correctness of the AVX2 implementation. For implementations, that are able to run directly on the hardware, it is of course possible to run them without the `sde` command. If the files `in` and `out` are large enough chances are good that the cipher is correct.

```
name         : __cbc-twofish-avx
driver       : __driver-cbc-twofish-avx
module       : twofish_avx_x86_64
priority     : 0
refcnt       : 1
selftest     : passed
type         : blkcipher
blocksize    : 16
min keysize  : 16
max keysize  : 32
ivsize       : 0
geniv        : <default>
```

```
name         : cbc(twofish)
driver       : cbc-twofish-avx
module       : twofish_avx_x86_64
priority     : 400
refcnt       : 1
selftest     : passed
type         : ablkcipher
async        : yes
blocksize    : 16
min keysize  : 16
max keysize  : 32
ivsize       : 16
geniv        : <default>
```

Figure 4.1: Entries for Twofish in CBC mode in `/proc/crypto`

If we boot our patched kernel and load one of our modules, for example `twofish-avx-x86_64`, it does

an self-test on startup with the testvectors provided by the testmanager and shows up in `/proc/crypto`. In figure 4.1 you see the entries for Twofish in CBC mode. Again there are two entries, one for the synchronous implementation on the left, which is called by the asynchronous implementation, which is referred to by the second entry on the right. You see that the synchronous implementation has priority zero and a non-common name, whereas the asynchronous implementation has priority 400 and the typical name combination, which is used by applications searching for the cipher. We do not explain every field here, because in fact this was done already in section 3.1.3 for the Serpent cipher. One interesting field is the field `selftest`. As you can see both variants have passed their self-test, but remember that for the synchronous implementation this is just a null test, which always succeeds. To ensure that the self-test has been passed correctly the field has to be set to `passed` for the asynchronous implementation on the right side. In figure 4.1 this is the case and so we have evidence, that the cipher works as intended. The test can also be done manually with the help of the `tcrypt` module. Tcrypt does the tests with the same testvectors in the module initialization function and after that the module gets unloaded again. The ciphers, which should be tested, can be specified by module parameters, which have to be passed when loading the module. We will also use this module to make performance tests in section 4.4.

Now we have evidence that the block cipher implementations work correct from the tests in userspace and we also may assume that the algorithms the module provides work as intended, because of the self-tests with the given testvectors. Nevertheless we want to do one more test, to ensure our assumptions. We use our module in conjunction with the device mapper dm-crypt. To make it even more plausible, that our implementation is correct, we set up a crypto container with an already available implementation and write data to this container. After that we take our newly introduced implementation, open the container with the same options and an identical encryption key and read the data we have previously written. We then compare the read data to the original data, and if they are identical we can be quite sure that our implementation is correct. Figure 4.2 shows a listing of commands that need to be executed to achieve the just described process. The listing is an example for testing the Serpent CBC reading routine. Dm-crypt takes always the cipher with the highest priority, which is currently loaded. If none is loaded they get all loaded by common name and the one with the highest priority is chosen. That is why we have to manually load and unload the modules in this listing. We can verify, that the module has been taken, which we actually wanted to test, by looking at the reference count for the cipher in `/proc/crypto`. In the example we create a crypto container called `serpent` with the key file `keyfile` and write 1MB data to it, taken from the file `in`. In the second step this data is read with our module and compared to the original data. If all the commands can be successfully executed without printing stuff, the cipher works as intended. The mode `cbc-essiv:sha256` means to use standard CBC mode with a special initialization vector, the ESSIV (Encrypted salt-sector initialization vector). The ESSIV is a hash of the combination of the sector number and the disk encryption key. As the mode name suggests the hash function taken for this process is SHA256.

```
> modprobe serpent-generic
> cryptsetup --cipher serpent-cbc-essiv:sha256 --key-file keyfile create serpent disk
> dd if=in of=/dev/mapper/serpent bs=1024 count=1024 2> /dev/null
> cryptsetup remove serpent
> rmmod serpent-generic
>
> modprobe serpent-avx-x86_64
> cryptsetup --cipher serpent-cbc-essiv:sha256 --key-file keyfile create serpent disk
> dd if=/dev/mapper/serpent of=/dev/stdout bs=1024 count=1024 2> /dev/null | diff in -
> cryptsetup remove serpent
> rmmod serpent-avx-x86_64
```

Figure 4.2: Testing the Serpent CBC reading routine with dm-crypt

With the approach from figure 4.2 we can test many different situations. Of course the lines from the listing have not been typed by hand, but instead a script has been written, which is capable of checking all five modes of operation for a given pair of a trusted old and a newly introduced cipher. In the listing only the checking of the reading routine is shown, but of course the writing routine has been checked as well. Furthermore different data sizes have been tried as well as different keys. Instead of just writing data to the crypto container it is also possible to generate a filesystem within the container and mount this

filesystem. After that some files can be created and later be retrieved, when mounting the filesystem again with a different implementation. If the implementation would not be correct, changes are very high, that even the mounting fails. This can be simulated by supplying a different key in the second step. However using filesystems for the testing process does not bring any new information on the correctness of the cipher and so it should be enough to make tests similar to the listing in figure 4.2.

4.4 Performance

In this section we will make a detailed performance evaluation of the implementations, we have introduced in chapter 3. For every implementation we will show a summery of the instructions needed, measure the computing time in userspace and make a whole bunch of tests in kernel space. The different modes of operation are compared to each other as well as the new implementations to the already existing implementations. Last but not least we do some disk encryption tests with the device mapper to see, whether we can achieve a higher data rate with our implementations. All tests have been performed on an Intel Core i5-2500 CPU, which supports AVX natively.

For the userspace tests the implementations have been compiled with GCC version 4.7, because this version includes support for AVX2. The following flags were used for the compilation process:

```
-std=c99 -Wall -Werror -O2 -fexpensive-optimizations
```

For most implementations the flags do not matter, because they are written in plain assembler, but for the C implementations they make a difference and are used to be able to make fair comparisons between C and assembler implementations. The instruction counts include only the instructions, which are actually used within the parallel implementation of the block cipher itself. Instructions, which are needed for the key scheduling and for the repeated calling of the block cipher are not counted. The small overhead caused by the ECB mode implementation is not listed within the instruction count. The instructions were counted with the Intel Software Development Emulator [10] based on a test application that encrypts exactly 1MB of data. The timings are based on an application that encrypts exactly 1GB of data in ECB mode. The data is read from memory and written to memory, thus no disk I/O operations are performed. Of course the time the application takes has been measured multiple times and an average value has been calculated out of the results. For the AVX2 implementation we are unable to make time measurements, because we can just run the implementation within the SDE. For all other implementations, existing ones and the ones we wrote, we give the absolute time the implementation takes to encrypt 1GB and the percentage that the implementation is faster than the previous one. This makes it very simple to decide, whether an implementation is faster than another one. Of course this is not possible either for AVX2 and so we have to stick with the instruction count in that case.

After the userspace results we will present results, we have measured in kernel space. Therefore the `tcrypt` module has been used. To make things more accurate, tcrypt has been patched to perform more iterations for a specific configuration, than it usually would do within the standard kernel. Furthermore the output of the testing routine has been modified to make it easier to parse the data. It turned out that the results do not change for more input data than 8192 bytes, but it has been tested up to 65536 bytes. That s why we just show results for the input data between 16 and 8192 bytes in the graphs below.

Last but not least we will show the results we got from the device mapper, when using our implementations for disk encryption. To get accurate results the measurements were taken within a ramdisk, because otherwise the measured results are fluctuating very heavy. The tool `hdparm` was used for this purpose and the ramdisk had a size of 2.5 GB. The reading speed has been measured for three seconds and the reading rate has been calculated out of the data, which has been read during this period. This process was repeated several times and the average of the results from the different runs has been taken as result for a specific module. To give a reference value the same procedure has been applied to the assembler and the AESNI implementation of AES. As mode `cbc-essiv:sha256` has been used, because it is a common mode for disk encryption and the default value for a dm-crypt setup with LUKS.

4.4.1 Serpent

Let us have a look at our Serpent implementation. Table 4.6 shows the results we got in userspace. All implementations are 64 bit implementations. The generic implementation is written in C and the other three are written in assembler. You can see that the speedup from the generic implementation to the SSE2 implementation is very huge. This becomes clear if you show yourself quite plainly, that the generic implementation only processes one block whereas the SSE2 implementation processes eight blocks at once in four block chunks. In the ideal case the speedup would be approximately four, but as there are parts, which cannot be done in parallel, for example memory operations, the speedup is in this case approximately 3.6, which is quite good. Our AVX implementation uses similar techniques as the SSE2 implementation, and therefore we cannot get such a high speedup compared to SSE2. However we are at least 6.1% faster than the SSE2 implementation. With the instruction count we can check that our calculations from section 3.1.1 have been correct. We have saved $(32456704 - 26624000)/((1024 \cdot 1024)/128) = 712$ instructions per encryption of eight blocks. This is the same result we got, when we implemented the cipher. The AVX2 implementation needs exactly half of the instruction the AVX implementation needs. This simply comes from the fact, that we can substitute the instructions one by one and as the AVX2 implementation processes sixteen blocks at once it is only called half as much as the AVX implementation. Of course we will not reach a speedup of 2 with the AVX2 implementation, but nevertheless it should be a quite huge speedup.

Implementation	Instructions	Time (s)	Speedup (%)
`generic`	87031808	12.605	-
`sse2`	32456704	3.514	258.68
`avx`	26624000	3.312	6.10
`avx2`	13312000	-	-

Table 4.6: Serpent instruction and timing results in userspace

Figure 4.3 shows the results of the tcrypt module for different modes of operation with the SSE2 implementation and our AVX implementation. For all measurements the key size is fixed to 256 bit and we use 8192 bytes of input data. Actually the key size does not matter, because the Serpent cipher needs always 32 rounds, independent of the key size. You see that our implementation is slightly faster than the SSE2 implementation and the speedup is almost independent of the specific mode. The value might be a bit lower than the 6.1% we got in userspace, because of the overhead caused by the particular mode, but the dimension remains the same. Thus the speedup is still measurable in kernel space with all the different modes. Of course the CBC encryption is an exception, because it has to be done sequentially. The SSE2 and AVX implementation just call the generic implementation and therefore are both very slow. The graph has been cropped, because the real value was just too large (about 735000 cycles).

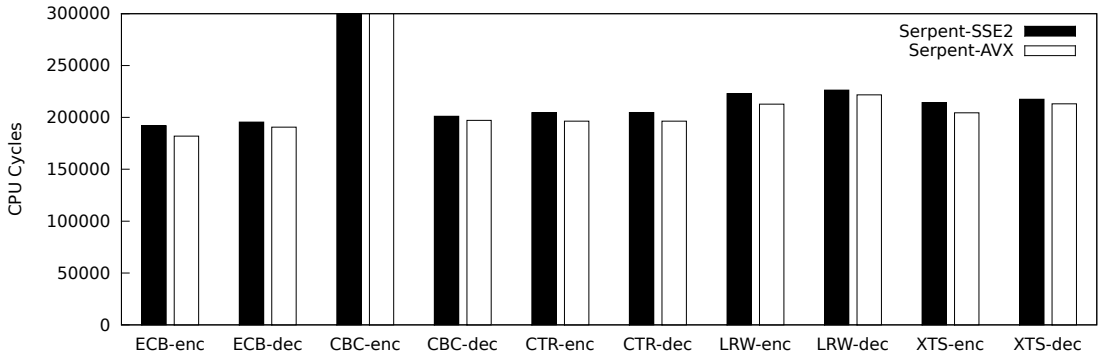

Figure 4.3: Tcrypt results for different modes with Serpent (256 bit key, 8192 input bytes)

In figure 4.4 you see two graphs based on the same measurement results. Both just show the CBC decryption routine for the SSE2 and AVX implementation of Serpent and the AESNI and x86_64 assembler implementation of AES as reference. For all implementations the key size is fixed to 256 bit and the graphs

show the cycles needed for a different amount of input data. Of course more cycles are needed if more data has to be processed and that is why in the second graph on the right the ratio of cycles and bytes is shown, instead of just the cycles. Furthermore the scale in the right graph is logarithmic to show the behaviour of the implementations for small input data sizes. Again you see that the AVX implementation is slightly faster than the SSE2 implementation, but of course the difference is very small if we compare it with AES. For large data sizes both Serpent implementations are way faster than the assembler implementation of AES, which is an interesting result, because the Serpent cipher is considered to be very slow compared to AES. This is confirmed by the graph on the right, where Serpent is slower for small data sizes. As Serpent uses no table lookups, it can be parallelized very efficiently even compared to AES. However we have no chance to compete with the AESNI implementation of AES, which runs in hardware and is simply faster than every implementation of Serpent, independent of the data size.

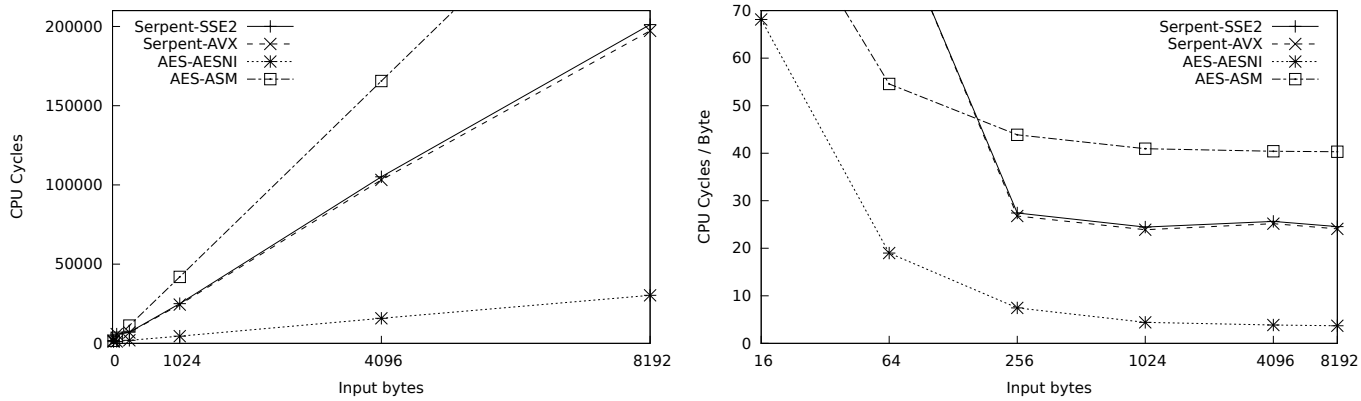

Figure 4.4: Tcrypt results of CBC decryption for Serpent and AES (256 bit key)

Table 4.7 shows the results with the device mapper, which we get with the different kernel modules, that provide a Serpent implementation. Basically the results we already pointed out in this section still apply for this application of the cipher, but of course the speedup is a little bit lower than in userspace. Again you see that Serpent is in general a very slow cipher, because the assembler implementation of AES is much faster than the generic Serpent implementation. Nevertheless the parallelized implementations are very fast and the speedup for the AVX implementation compared to the SSE2 implementation is still measurable when using the cipher with the device mapper.

Kernel Module	Disk Speed (MB/s)
`aes-x86_64`	318.68
`aesni-intel`	1055.75
`serpent-generic`	155.89
`serpent-sse2-x86_64`	435.57
`serpent-avx-x86_64`	445.24

Table 4.7: Serpent disk reading speed results (cbc-essiv:sha256)

4.4.2 Twofish

Table 4.8 shows the results, we got in userspace for the different implementations of Twofish. Every implementation is 64 bit and written in assembler, except from the generic implementation, which is written in C. The speedup from the generic implementation to the assembler implementation is not so huge as in the Serpent case, because this is just a one way parallel implementation. The 3-way parallel implementation is about 23.0% faster than the plain assembler implementation, although it needs a lot more instructions. This is because the 3-way implementation does more computations in registers and therefore minimizes memory accesses and is optimized for out-of-order CPUs. Our AVX implementation is about 30.8% faster

than the 3-way parallel implementation, which is quite a good result, and we need less instructions than the generic assembler implementation. There is a very huge decrease in instructions from AVX to AVX2. This is because in the AVX2 implementation we can eliminate all the sequential code, which is still present in the AVX implementation. We should get a very huge speedup with AVX2, because the whole code is ready to process sixteen blocks at once, which would cut in half the instructions, and the AVX2 implementation itself needs lesser instructions than the AVX implementation, because of the gather instructions.

Implementation	Instructions	Time (s)	Speedup (%)
`generic`	35913728	6.215	-
`asm_64`	28788575	5.800	7.15
`asm_64-3way`	34493255	4.714	23.03
`avx`	28622848	3.605	30.79
`avx2`	6426624	-	-

Table 4.8: Twofish instruction and timing results in userspace

The results for the different modes of operation with the Twofish implementations are shown in figure 4.5. Again the key size is fixed at 256 bit and the input data size is 8192 bytes. You can see clearly the speedup between the 3-way and the AVX implementation with all modes of operation. In userspace we got a result of about 30.8% and the dimension of this speedup remains in the measurements made in kernel space. The CBC encryption is again implemented sequentially, but is not so much slower than the other modes, as it was the case for Serpent. Firstly Twofish is considered to be a faster cipher than Serpent and secondly Twofish cannot be parallelized as good as Serpent.

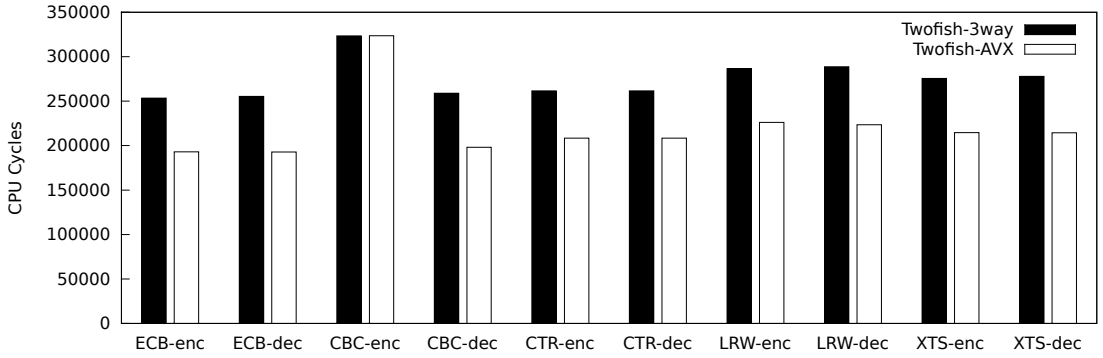

Figure 4.5: Tcrypt results for different modes with Twofish (256 bit key, 8192 input bytes)

In figure 4.6 you see the two graphs of the CBC decryption routine for the two implementations of Twofish and the two implementations of AES. The speedup between the 3-way and the AVX implementation is still clearly visible in both graphs and of course the AESNI implementation of AES is still by far the fastest implementation. In the right graph you see that Twofish is a faster cipher, because the assembler implementation of AES is now always slower, even for small data sizes. For extremely small data sizes even the AESNI implementation is a little bit slower. Furthermore it is visible that the graphs of the Twofish implementations converge with increasing data size and stay proportional to each other in the right diagram. The absolute speed of the AVX implementation is approximately the same as the speed of the AVX implementation of Serpent (about 24 cycles per byte).

The disk reading speed results of the different Twofish implementations are shown in table 4.9. The assembler implementation of Twofish is already able to compete with the assembler implementation of AES and the 3-way and AVX implementation is much faster than the assembler implementation of AES. Of course the AESNI implementation of AES is still faster than every other implementation. The speedup between the 3-way implementation and our AVX implementation is visible and has practical impact on applications using our cipher for disk encryption.

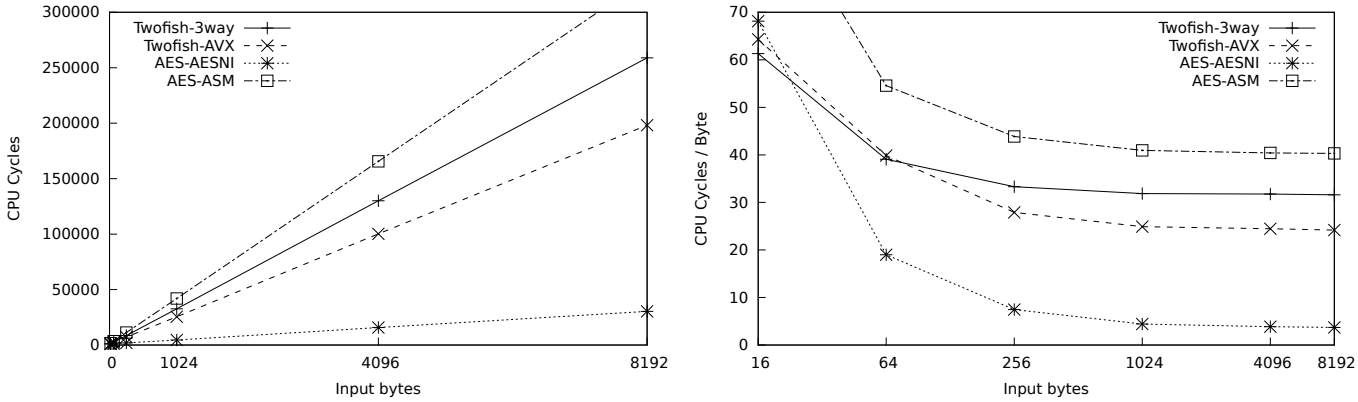

Figure 4.6: Tcrypt results of CBC decryption for Twofish and AES (256 bit key)

Kernel Module	Disk Speed (MB/s)
aes-x86_64	318.68
aesni-intel	1055.75
twofish-generic	282.15
twofish-x86_64	314.98
twofish-x86_64-3way	390.15
twofish-avx-x86_64	467.49

Table 4.9: Twofish disk reading speed results (cbc-essiv:sha256)

4.4.3 Blowfish

In table 4.10 you see the results, we got in user space for the different Blowfish implementations. The generic version is written in C and all other implementations are written in plain assembler or in assembler with AVX extensions. The first thing you might notice is that the generic implementation is about 7.1% faster than the plain assembler implementation, which processes just one block at once. This result has been observed multiple times on our testing machine. On a different machine with an Intel Xeon E31220 processor the two implementations took approximately the same time. With no other implementation such big deviations have been discovered. It seems that the compiler does a really good job and for some reason the plain assembler version is slowed down on our test machine. The 4-way parallel assembler implementation, however, brings a huge speedup on all machines, which it were tested on. It is more than two times faster than the generic implementation.

Implementation	Instructions	Time (s)	Speedup (%)
asm_64	27787264	8.451	-
generic	29229056	7.888	7.14
asm_64-4way	25591808	3.629	117.36
avx	27082752	3.600	0.81
avx2	5398528	-	-

Table 4.10: Blowfish instruction and timing results in userspace

If we look at our implementations we see that we cannot save much instructions with the AVX implementation. We even need more instructions than the 4-way implementation. The AVX2 implementation, however, needs really few instructions compared to all other implementations. This is because no sequential parts remain in the AVX2 implementation, apart from the reading from and writing to memory parts. The speedup should therefore be very huge. Our AVX implementation is about 0.8% faster than the 4-way parallel implementation, which is already present in the kernel. The speedup is so low, that it is not worth to submit this implementation to the kernel, because it would add a lot of code, but almost nobody would

benefit from this implementation. The AVX implementation has also been evaluated in kernel space and the speedup has been as low as these userspace results already show. The kernel space measurements do not bring any new information and as the patch has not been submitted anyway, the results from the `tcrypt` module are skipped here.

4.4.4 Cast-128

The structure of the Cast-128 algorithm is related to the structure of the Blowfish algorithm and therefore the results are not completely different. The timing and instruction results from userspace are listed in table 4.11. As it was the case for Blowfish, we save a lot of instructions with the AVX2 implementation, because we are able to do the table lookups in parallel. With the AVX implementation we cannot save so much instructions, but the difference between the generic implementation and the AVX implementation is bigger than it has been for Blowfish. The AVX implementation is more than two times faster than the generic implementation, i.e. we get a speedup of about 115.8%. It might be possible to write an assembler implementation, that uses just general purpose registers and that is much faster than the generic implementation as well. However we are able to do more things in parallel, in contrast to the Blowfish implementation, and the 4-way parallel assembler implementation of Blowfish was not able to beat our AVX implementation. Therefore it might be suspected, that even if someone writes such an implementation, our AVX implementation of Cast-128 is still faster.

Implementation	Instructions	Time (s)	Speedup (%)
`generic`	35389440	8.767	-
`avx`	28876800	4.062	115.83
`avx2`	6361088	-	-

Table 4.11: Cast-128 instruction and timing results in userspace

In figure 4.7 the results we got for Cast-128 in kernel space are shown. As for the userspace tests, a key of 128 bits has been taken, i.e. sixteen rounds are performed by the cipher. The performance is listed for the different modes and every mode has been tested with an input size of 8192 bytes by the `tcrypt` module. Only the modes ECB, CBC and CTR are available for Cast-128. It is clearly visible that the speedup is greater than two and is still present, regardless of the mode used. The exception is of course CBC in encryption mode, because in this case the same implementation is called for both modules. You can see that the overhead of calling the generic implementation slows down the encryption process a little bit in that case.

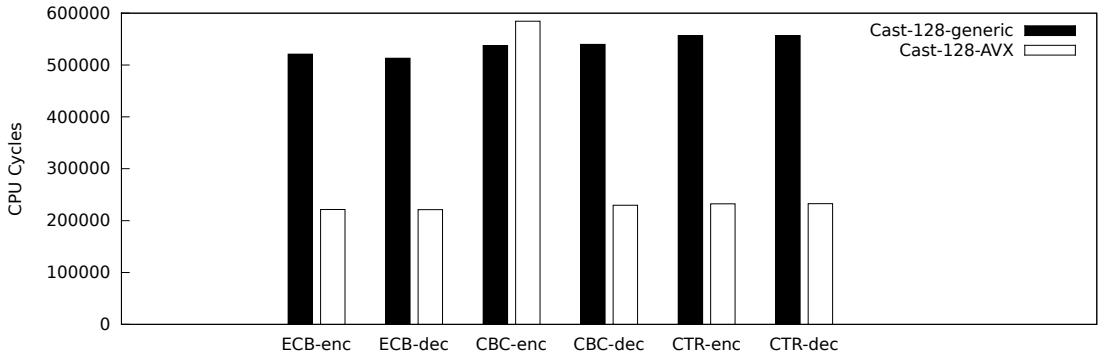

Figure 4.7: Tcrypt results for different modes with Cast-128 (128 bit key, 8192 input bytes)

As Cast-128 has only a block size of 64 bits we will not show the comparison with AES and instead discuss the successor Cast-256 a bit more in detail. We have seen that the speedup remains visible in kernel space and therefore it can be exploited by applications, that make use of the crypto API of the kernel.

4.4.5 Cast-256

The structure of the Cast-256 algorithm is closely related to Cast-128, but if we just look at the instructions used in the algorithm and the parts that can be done in parallel we should compare it with Twofish for two reasons. Firstly the block size of both ciphers is 128 bits and secondly there are approximately as much operations, which can be done in parallel compared to the ones which have to be done sequentially. However Twofish in general needs fewer instructions, i.e. is a faster cipher. With the AVX implementation we can save approximately a third of the instructions from the generic implementation, which is quite good. Of course the AVX2 implementation needs again way fewer instructions than all other implementations. With our AVX implementation we get a speedup of approximately 88.6%. It is no surprise that this speedup is quite good, because there exists just the generic implementation. Nevertheless it should not be easy to write an assembler implementation, that uses just general purpose registers and is faster than our AVX implementation. One reason is that there are probably not enough registers to write a 4-way parallel implementation and a 3-way parallel implementation would be faster than the generic implementation, but probably not faster than our AVX implementation. This is the same situation, which occurred for the Twofish implementations.

Implementation	Instructions	Time (s)	Speedup (%)
`generic`	61931520	11.522	-
`avx`	43499520	6.109	88.61
`avx2`	9740288	-	-

Table 4.12: Cast-256 instruction and timing results in userspace

After the userspace evaluation, the Cast-256 implementations have been tested in kernel space. Figure 4.8 shows the results for all five modes of operation, which are supported by Cast-256. The speedup is clearly visible for all five modes and with the CBC encryption mode you see the overhead that is caused by calling the generic implementation indirectly.

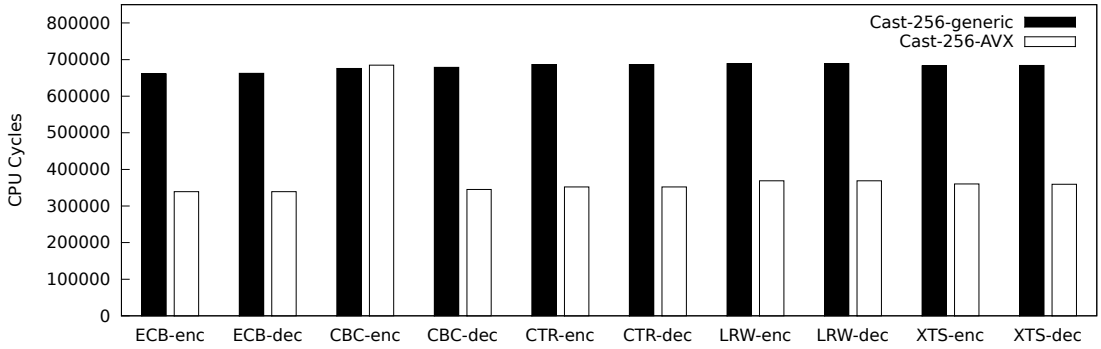

Figure 4.8: Tcrypt results for different modes with Cast-256 (256 bit key, 8192 input bytes)

In figure 4.9 the performance of the CBC decryption mode is shown for different input sizes. As reference the plain assembler and the AESNI implementation of AES is shown, too. The left diagram shows the absolute value of CPU cycles, that are needed for a specific input data size by a specific implementation, and the right diagram shows the CPU cycles, that are needed per byte. Furthermore the right diagram has a logarithmic scale, which makes it easier to look at the small input sizes. You can see, that Cast-256 is a slow cipher, even if compared to Twofish and Serpent. The AVX implementation of Cast-256 is a little bit slower than the assembler implementation of AES, but much faster than the generic implementation. In both diagrams you see, that the AESNI implementation beats both implementations of Cast-256 by far. All implementations listed in figure 4.9 converge against some value in the right diagram and therefore the speedup remains constant for large enough input sizes.

There is one thing, which is a little bit wired. If you look at the right diagram, you see that the generic

implementation of Cast-256 is always slower than the AVX implementation, even for very small input sizes. Actually this should not be the case, because for input sizes smaller than 128 bytes the generic implementation is called by the AVX implementation and this should take longer or at least not be faster than calling the generic implementation directly. In figure 4.8 we have clearly seen the overhead that is caused by calling the generic implementation indirectly for the CBC encryption mode with more input data. It would be easy to say that this is just a measuring fault, but the same behaviour has been reproduced several times. Of course the results for so few bytes of input data are not really accurate, but nevertheless there should be an explanation of this behaviour. One explanation might be that the AVX implementation is faster, because we are testing the implementations in asynchronous mode. For the AVX implementation the asynchronous mode is directly implemented in the gluecode of the AVX implementation and therefore it might be faster than the generic implementation of the asynchronous interface in the crypto API, which in the end calls the generic implementation of Cast-256. Concluding this considerations, we can say that the AVX implementation performs a lot better than the generic implementation for large input sizes and at least not worse than the generic implementation for small input sizes.

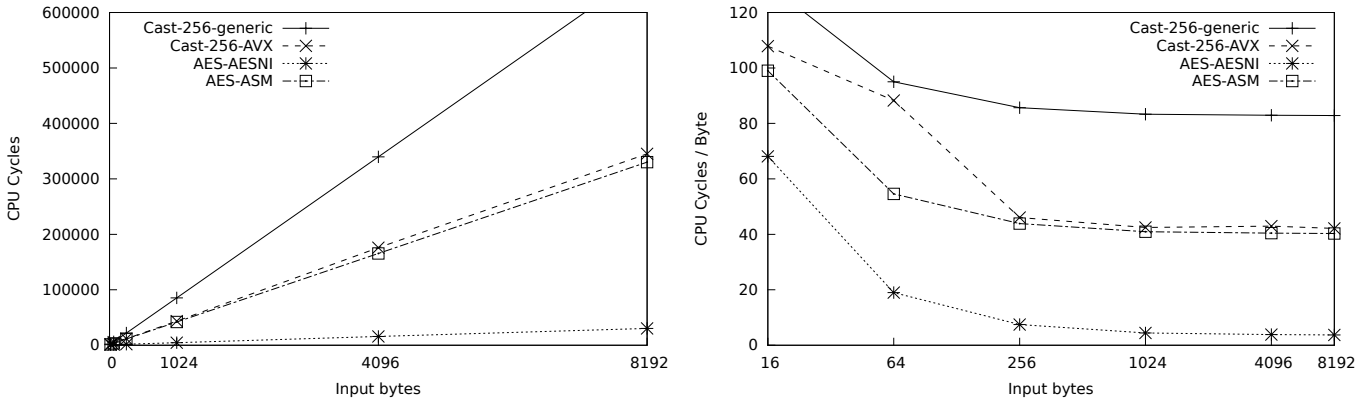

Figure 4.9: Tcrypt results of CBC decryption for Cast-256 and AES (256 bit key)

The last measurements, that were made for the Cast-256 implementation, are the disk reading speeds in CBC mode. The same setup as for the other implementations has been used, i.e. dm-crypt with a large enough ramdisk. The results are shown in table 4.13 together with the reference values for AES. You see once more that Cast-256 is a slow cipher, because even our optimized implementation is a little bit slower than the assembler implementation of AES. The speedup we have achieved is clearly visible in the data rate, we got when using this implementation for disk encryption. If someone ever would want to encrypt his harddisk with Cast-256, he certainly would recognize this speedup in applications that perform a lot of I/O operations on the disk.

Kernel Module	Disk Speed (MB/s)
`aes-x86_64`	318.68
`aesni-intel`	1055.75
`cast6-generic`	160.70
`cast6-avx-x86_64`	282.15

Table 4.13: Cast-256 disk reading speed results (cbc-essiv:sha256)

4.5 Security

Besides correctness and performance, the security of a cipher is the most important issue and it has to be checked carefully, whether a given implementation is secure. There are many different attacks, which could be carried out on our implementations. As we have implemented known and respected standards, that have

no weaknesses as such, we will not look at the ciphers from a mathematical point of view, but rather discuss, what could possible go wrong in the specific implementations. Furthermore we will focus on problems that could arise, when the ciphers are used for disk encryption within the Linux kernel. Of course it is possible to use the AVX and AVX2 implementations presented in chapter 3 in many different application contexts and in every context there might arise different problems, but we are not able to cover all possible scenarios in this section and disk encryption is the most common use case for our implementations, at least for the ones that run in kernel space.

First of all we have to think about what insecurity actually means. A cipher should certainly be considered as insecure, if conclusions from the ciphertext about the plaintext can be drawn. It would be even worse, if it is somehow possible to reconstruct the encryption key, that has been used to generate the ciphertext. In this second case it is immediately possible to retrieve the plaintext from the ciphertext, because an attacker has just to use the decryption routine of the symmetric cipher. We assume that the ciphers, we implemented, have no weaknesses in the sense that changes in the ciphertext somehow correlate to changes in the plaintext. This assumption is plausible, because in section 4.3 we have shown, that our implementations work correct according to the specifications and the ciphers are designed in a way, that eliminates such correlations. In chapter 3 some of these design principles have been named. These might be, for example, the diffusion properties of the Serpent S-boxes or the MDS matrix, which is used by Twofish.

The design principles, however, just ensure that changes in the ciphertext are not correlated to changes in the plaintext for exactly one block of data. As our ciphers are all processing many blocks at once, this case would not apply. If somebody for example changes the first byte of the plaintext, only the first block of the ciphertext would change, if all blocks are encrypted with our parallel cipher. This does not make it possible to decrypt the ciphertext, but as the overall structure changes in some predictable way, an attacker might draw conclusions from the changes in the ciphertext compared to the changes in the plaintext. Such attacks are qualified as *known plaintext* attacks, because the attacker changes the plaintext and looks at changes in the ciphertext. In this case equivalent plaintext blocks would result in equivalent ciphertext blocks, which would make it easier for an attacker to guess the meaning of a large enough ciphertext. To avoid this kind of attacks, our parallel ciphers have to be used in combination with a secure mode of operation. In section 2.1.1 we have presented three out of the five modes, that are available for our ciphers in kernel space, but only two of them, CBC and CTR, are able to ensure the decorrelation and prevent the known plaintext attack.

So let us assume the cipher has been designed properly and is used in conjunction with a secure mode of operation. It has to be ensured, that the secret encryption key, which is stored in memory all the time, during the disk is unlocked, cannot be accessed by an unauthorized person. The first thing, which comes in mind, is that the Linux kernel has to prevent such unauthorized accesses. Of course the security of the disk encryption is compromised, if someone gains root access by a local attack. In this case, however, not only the disk encryption is compromised, but instead the attacker can just read the data directly. So local privilege escalation is an issue, but not an attack, we have to take care of in our implementations, because the kernel itself has to prevent it.

Let us further assume the key cannot be accessed by an unauthorized user on the local system, what else could go wrong? As the key is stored in memory permanently during the disk is used, it might be possible to read out the key physically. One possibility is to read out the memory directly, by removing it from the running device and putting it into a different device, which runs some dumping software. For this scenario the memory has to be cooled down, to keep the data during the process, and therefore this is often referred to as *cold-boot attack*. Another possibility is to use the peripheral hardware to gain memory access, e.g. via Firewire. All our implementations are vulnerable to both scenarios, as are all other implementations in the Linux kernel, at least the ones that run in software. There exist approaches that circumvent this problem by storing all secret data, i.e. encryption key and round keys, in registers instead of the memory. For example there exists a cold-boot resistant implementation of AES [40]. It would be possible to write cold-boot resistant implementations of the ciphers, presented in this thesis, too, however this would result in a significant performance decrease and not every cipher makes a cold-boot resistant implementation easily possible. Twofish uses large key-dependent S-boxes, which certainly could not be stored in registers. In this case a different approach would be required, such as to encrypt the key-dependent S-boxes, that are still stored in memory, with some different secret key. If the memory would be dumped, the S-boxes could not be decrypted. For the encryption of the S-boxes, however, Twofish cannot be used by itself, but rather

another, very simple cipher would be required, that is able to store the data needed for the encryption of the S-boxes in the CPU registers. The key needed for the encryption of the S-boxes would need to be chosen completely at random. For ciphers, that do not need so much key-dependent data during the encryption routine, a cold-boot resistant implementation would of course be easier. Serpent, for instance, uses fixed S-boxes and the user supplied key is expanded to the subkeys by an affine recurrence. In this case the key scheduling part could be done on-the-fly during the encryption routine for each round and every part of the secret data could be stored in registers, like it has been shown for AES [40]. Providing cold-boot resistant implementations of our implementations would be possible in general, but depending on the cipher it would be more or less difficult to achieve this. It should also be noted, that if an attacker is able to dump the memory, he could read out sensitive data anyway. A cold-boot resistant implementation just prevents data, that currently is not present in memory.

Another possible attack scenario on symmetric ciphers, which we want to look at, are so called *timing attacks*. These attacks fall into the category of *side-channel attacks*, because not the outputs, the cipher produces on predictable inputs, are taken into account, but instead the side effects during encryption are examined. Unfortunately most of our implementations do not run in constant time, but instead different input data and different encryption keys lead to a different execution time. Daniel Bernstein showed in 2005 [7], how it is possible to recover the complete AES key of an AES implementation, just by looking at the timings, to which different and known inputs led. In his setup he attacked a network server, which used the AES implementation, and measured the response times. This work showed, that a timing attack is not just a theoretical issue, but has practical relevance. We have to think about why the timings of our implementations might be different on different inputs and how we can avoid this. In Cast-128, for example, we can guess whether a short or a long key has been used, because the number of rounds is different and therefore the encryption will take more or less time. Apart from this trivial example the situation is difficult, because actually all our implementations are written with branchless code. It is questionable whether the different instructions, that we use, take more or less time, but at least for the logical operations, which make up the greatest part, it can be assumed that they all take approximately equal time. However all our implementations, except Serpent, use S-boxes and therefore have to access different regions in memory. When doing this a lot of cache issues arise and consequently the time of these lookups differ, because it takes longer to access the memory than just the cache. The work [7] of Bernstein exploited these cache timing issues with the results mentioned above. All our implementations, except Serpent, are vulnerable to this attack, even if we would implement them in a cold-boot resistant way. To avoid cache-timing attacks completely, we need to avoid table lookups. With Serpent we have luck, and it is possible to implement AES without table lookups, when using the AESNI instruction set. If we want to avoid the table lookups in our other implementations, we would have to substitute them by some instructions that run in constant time. The S-boxes are usually generated by some algebraic operations and we could try to implement some sort of on-the-fly S-box generation, but it is not ensured that for every cipher the resulting implementation really runs in constant time and even if this would be the case, the performance decrease would be far beyond the speedup, that we got with our implementations. So if there is not hardware support, like for AES, or the cipher itself is not specified without table lookups, like Serpent, it is probably not possible to implement a specific cipher resistant against cache-timing attacks in a way that still allows the usage of this implementation in practice.

Concluding this section it can be said that our implementations are as secure as the already existing implementations in the Linux kernel. All the attacks mentioned above do not break the cipher directly, but instead exploit some other piece of software or are based on informations got from side effects, and are applicable to the generic implementations as well. Furthermore the scenarios mentioned in this sections are all serious security threats, but in the main use case of disk encryption, that is to protect a switched off and unattended computer, they all remain secure.

5 Conclusion and Future Work

In this chapter we want to draw conclusions about the work, which has been done during this thesis. In chapter 4 you saw the detailed evaluation of our implementations and especially the performance analysis. Based on these results we want to have a look on limitations that remain with our implementations in section 5.1. In section 5.2 we will cover topics that are relevant for future work within the context of this thesis. Of course there are always ciphers left, which could be accelerated, but it could also be considered to port our implementations to different architectures. Finally we will summarize the work and the results in section 5.3.

5.1 Limitations

All our implementations process more than one block at once. On the one side this is the cause of the speedup, we have achieved, but on the other side this forces us to reimplement the modes of operation. This is one limitation we have to accept, because we cannot parallelize the processing of just one block, as many operations, e.g. to process the block in subsequent rounds, have to be done sequentially. We can work around this limitation by writing more gluecode, but especially when integrating the implementation into the Linux kernel this is a drawback, since we have to break with the abstraction of the crypto API. This makes maintenance more difficult and increases the code size of the kernel. From a users point of view, however, this is no real limitation, because our ciphers work with all input sizes as intended.

With AVX there have been 256 bit wide registers introduced, but only 128 bits can be used efficiently with integer instructions, which we need all over our implementations. This is a serious limitation, because with this restriction the available registers are the same that were already available with SSE2 and the AVX implementations can just benefit, compared to the SSE2 implementations, by exploiting new instructions and the non-destructive three operand syntax, which has been introduced with AVX. Only with AVX2 this limitation can be eliminated, because the 256 bit wide registers will become usable with integer instructions, too. As we provide an AVX2 implementation for each of the five algorithms, we will get around this limitation in the future without additional work, aside from integrating the AVX2 implementations into the kernel.

The major limitation, which has occurred during this thesis, is that AVX is lacking instructions for parallel table lookups. Unfortunately every implementation, aside from the Serpent implementation, uses table lookups regularly in every round of encryption or decryption. Every time a table lookup is needed, the data has to be extracted to general purpose registers, the table lookup has to be done, and the data has to be reinserted into SIMD registers. Of course this takes a lot of time, but worse, the lookups in the general purpose registers have to be done sequentially. This means, that we cannot gain speed, because of the sequentiality of the lookups, but instead loose speed by moving data around. It depends on the design of the specific cipher, whether these steps are worth to be made or whether they would slow down the whole implementation too much. In all of the five algorithms there were still operations, which could be replaced by SIMD equivalents, and therefore we got an overall performance increase. However, if you look at the implementation of Blowfish, the speedup was quite low, because there were not much operations that could be parallelized compared to the table lookups, that need to be done sequentially. This issue is a limitation, which applies to basically all symmetric ciphers, and therefore a solution for this problem is preferable. Fortunately AVX2 provides instructions, that solve exactly this problem, and are able to do the table lookups in parallel and within the SIMD registers. This way even both problems get solved, because we do not loose speed by moving data around and gain speed by parallelizing operations. All our AVX2 implementations already exploit this new technique and therefore are able to overcome this major limitation in the future.

5.2 Future Work

A task, which should be done in the future, is to make meaningful performance evaluations of the five AVX2 implementations, as they could not been tested on real hardware and therefore no timing informations are available by now. If it shows up that the AVX2 implementations are faster than the AVX implementations, and they probably will, they could be integrated into the Linux kernel, like it has been done with the AVX implementations.

There are still ciphers left, for which a fast implementation could be provided. It is pointless trying to speed up AES with AVX, because of the AESNI implementation, but ciphers with similar properties, like Camellia [5] would be interesting, too. Every cipher, which is build with the structure of a Feistel network, is interesting, because the implementation of that cipher would be similar to the implementation of Twofish or Blowfish, depending on the block size, and could therefore probably be implemented with moderate effort. Besides symmetric ciphers, hash algorithms would be of interest, too. As already stated, it has been considered trying to accelerate the SHA-3 candidates and this would still be a challenging task. It would not be possible to process whole blocks in parallel, but there are chances that the processing of a single block could be accelerated with AVX or AVX2. It might be also worth to have a look at older hash algorithms like SHA-1, SHA-2 or even MD5, because they are widely used and will probably be available for years.

Another topic is to port our implementations to some different architecture and use a different instruction set. AMD has introduced XOP [3], which is compatible to Intel's AVX, but has some little advantages over AVX. In most of our implementations we needed packed rotations and implemented them by replacing the rotation by a packed logical left shift, a packed logical right shift and an or operation. It took three SIMD instructions for every rotation, although it would be possible to use just one instruction, if the data would be present in general purpose registers. Of course general purpose registers are not an option, because this would result in sequential processing of data. The XOP instruction set supports, as an addition to the instructions, that are supported by AVX, packed rotations with just one single instruction and would therefore be qualified for our implementations. This would require just a small change, but probably give some more boost to our implementations. The downside, however, is that the resulting implementations would not run on Intel CPUs anymore, but it might be possible to provide two implementations, which are selected depending on the current CPU automatically.

It would be really interesting and useful to port our implementations to the ARM platform. Most smartphones as well as small netbooks and other portable devices are equipped with an ARM processor. For netbooks, disk encryption is an important issue and it might get important for smartphones, too, which means that there is a demand for fast ciphers on this platform. There is an extension, called NEON [37],

which is supported by modern processors of the ARM architecture, for example the ARM Cortex-A series, that provides SIMD instructions, which are similar to those of AVX. Even on these processors, designed for mobile and embedded devices, the NEON extension provides support for 128 bit wide registers. Integer and floating point operations are supported simultaneously and the registers can be viewed as 32 64 bit wide registers or 16 128 bit wide registers. The latter can be considered equal to the 16 XMM registers, AVX provides, if we limit ourselves to the usage of integer instructions. Not only the registers have a similar meaning, but also the instruction set is related to AVX. Since ARM has a RISC instruction set, the three operand syntax is nothing special to SIMD instructions, but is also used for instructions, that operate on generic registers. There are instructions that do packed operations as well as movement and shuffle operations.

It should also be noted that on the ARM architecture there exists no AESNI instruction set, like on modern processors with x86_64 architecture. This makes it even more interesting to provide fast implementations of symmetric block ciphers based on SIMD instructions. It may be considered to accelerate AES itself, using the NEON instruction set, but even if this turns out to be a bad idea, it would be good to have fast implementations of other symmetric ciphers, like Twofish or Serpent. They would not be a lot slower than the fastest implementation of AES, because this implementation has to be done in software as well.

Concluding this section it can be said that there is still work left in the field of accelerating crypto primitives with SIMD instructions and that probably the most interesting and useful work would be to port the implementations of this thesis and related implementations to different architectures, most notably the ARM architecture.

5.3 Conclusion

There is demand for fast implementations of symmetric block ciphers in practical IT-Security, may it be for disk encryption or for popular Internet services like SMTP, HTTP and SSH. In many cases the use of encryption techniques is mandatory and a decrease in performance has to be accepted. The only way to improve this situation is to speed up cryptographic algorithms.

In this thesis fast implementations of the five symmetric block ciphers Serpent, Twofish, Blowfish, Cast-128 and Cast-256 have been presented. With the choice of these five algorithms we have accelerated today's most important symmetric block ciphers, aside from AES. The implementations make use of the *Advanced Vector Extensions* (AVX), a new SIMD instruction set, introduced by Intel, and are all based on the concept of parallel processing sequenced blocks. Every of the five AVX implementations is faster than the previously fastest implementation in the Linux kernel has been. These results have been verified in userspace, kernelspace and while measuring the data rate with disk encryption tests.

Because there were still limitations with the AVX implementations, we provided for every of the five algorithms an AVX2 implementation, which overcomes these limitations. We could not test the AVX2 implementations on real hardware, but verified the correctness with an emulator and made some guesses about the performance, which could be expected in the future. The AVX2 implementations are to be expected much faster than the AVX implementations and thus are expected to become important in the future.

For every AVX implementation we provided a patch, which integrates the implementation into the Linux kernel. This makes it possible to use our accelerated ciphers for disk encryption on a standard Linux system by everyone, who wants to test and verify the results of this thesis. As Linux is an open source operating system, we were able to contribute four of our five implementations to the official kernel and the implementation of Serpent and Twofish will already be available with version 3.6.

Bibliography

[1] C. Adams. The CAST-128 Encryption Algorithm, May 1997. http://tools.ietf.org/html/rfc2144.

[2] C. Adams and J. Gilchrist. The CAST-256 Encryption Algorithm, June 1999. http://tools.ietf.org/html/rfc2612.

[3] Advanced Micro Devices Inc. *AMD64 Architecture Programmer's Manual Volume 6*, 3.04 edition, November 2009.

[4] Ross Anderson, Eli Biham, and Lars Knudsen. Serpent: A Proposal for the Advanced Encryption Standard, March 2000. http://www.cl.cam.ac.uk/~rja14/Papers/serpent.pdf.

[5] Kazumaro Aoki, Tetsuya Ichikawa, and Masayuki Kanda et. al. Specification of Camellia – a 128-bit Block Cipher, September 2001. http://info.isl.ntt.co.jp/crypt/eng/camellia/dl/01espec.pdf.

[6] Jean-Philippe Aumasson, Luca Henzena, Willi Meier, and Raphael C.-W. Phan. SHA-3 proposal BLAKE, December 2010. http://131002.net/blake/blake.pdf.

[7] Daniel J. Bernstein. Cache-timing attacks on AES, April 2005.

[8] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The Keccak reference, January 2011. http://keccak.noekeon.org/Keccak-reference-3.0.pdf.

[9] Carolynn Burwick, Don Coppersmith, Edward D'Avignon, Rosario Gennaro, and Shai Halevi et al. MARS - a candidate cipher for AES, September 1999. http://www.research.ibm.com/security/mars.pdf.

[10] Intel Corporation. Intel Software Development Emulator, December 2009. http://software.intel.com/en-us/articles/intel-software-development-emulator/.

[11] Intel Corporation. Haswell New Instruction Descriptions, June 2011. http://software.intel.com/en-us/blogs/2011/06/13/haswell-new-instruction-descriptions-now-available/.

[12] Debian. Details of package gcc-4.7 in wheezy, April 2012. http://packages.debian.org/wheezy/gcc-4.7.

[13] Jake Edge. A netlink-based user-space crypto API, October 2010. http://lwn.net/Articles/410763/.

[14] Herbert Xu et. al. Scatterlist Cryptographic API, April 2012. http://www.kernel.org/doc/Documentation/crypto/api-intro.txt.

[15] Niels Ferguson, Stefan Lucks, and Bruce Schneier et. al. The Skein Hash Function Family, November 2008. http://www.skein-hash.info/sites/default/files/skein1.1.pdf.

[16] Praveen Gauravaram, Lars R. Knudsen, and Krystian Matusiewicz et. al. Grøstl – a SHA-3 candidate, March 2011. http://www.groestl.info/Groestl.pdf.

[17] Johannes Götzfried. crypto: cast5 - add x86_64/avx assembler implementation, July 2012. http://git.kernel.org/?p=linux/kernel/git/herbert/cryptodev-2.6.git;a=commit;h=f16479507ba5e85b43ce803f82d5c182ec8d04da.

[18] Johannes Götzfried. crypto: cast5 - prepare generic module for optimized implementations, July 2012. http://git.kernel.org/?p=linux/kernel/git/herbert/cryptodev-2.6.git;a=commit;h=8eefb4d16d6fee698f1d982245f71e9195d6d6d7.

[19] Johannes Götzfried. crypto: cast6 - add x86_64/avx assembler implementation, July 2012. http://git.kernel.org/?p=linux/kernel/git/herbert/cryptodev-2.6.git;a=commit;h=781bfe46558c23dc366788d627d181d12bc809e7.

[20] Johannes Götzfried. crypto: cast6 - prepare generic module for optimized implementations, July 2012. http://git.kernel.org/?p=linux/kernel/git/herbert/cryptodev-2.6.git;a=commit;h=142a1b8912dff1c2557a25c8b95732eb8ceffbf8.

[21] Johannes Götzfried. crypto: serpent - add x86_64/avx assembler implementation, June 2012. http://git.kernel.org/linus/7efe4076725aeb01722445b56613681aa492c8d6.

[22] Johannes Götzfried. crypto: testmgr - add larger cast5 testvectors, July 2012. http://git.kernel.org/?p=linux/kernel/git/herbert/cryptodev-2.6.git;a=commit;h=5d11474e015ec0ed5e7a44665216d0f6c45734a1.

[23] Johannes Götzfried. crypto: testmgr - add larger cast6 testvectors, July 2012. http://git.kernel.org/?p=linux/kernel/git/herbert/cryptodev-2.6.git;a=commit;h=a196d78298de3adab2ca0f0a0e2d4dc3de3a7329.

[24] Johannes Götzfried. crypto: testmgr - expand twofish test vectors, June 2012. http://git.kernel.org/linus/4da7de4d8be7d18559c56bca446b1161a3b63acc.

[25] Johannes Götzfried. crypto: twofish - add x86_64/avx assembler implementation, June 2012. http://git.kernel.org/linus/107778b592576c0c8e8d2ca7a2aa5415a4908223.

[26] Advanced Micro Devices Inc. Striking a Balance, May 2009. http://blogs.amd.com/developer/2009/05/06/striking-a-balance/.

[27] Free Software Foundation Inc. GCC 4.6 Release Series, March 2011. http://gcc.gnu.org/gcc-4.6/changes.html.

[28] Free Software Foundation Inc. GCC 4.7 Release Series, March 2012. http://gcc.gnu.org/gcc-4.7/changes.html.

[29] Linux Kernel Organization Inc. Majordomo lists at VGER.KERNEL.ORG, April 2012. http://vger.kernel.org/vger-lists.html.

[30] Linux Kernel Organization Inc. The Linux Kernel Archives, April 2012. http://kernel.org/.

[31] Intel Corporation. *Intel Advanced Encryption Standard (AES) New Instructions Set*, 323641-001 edition, May 2010.

[32] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 248966-025 edition, June 2011.

[33] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 325462-041us edition, December 2011.

[34] Intel Corporation. *Intel Advanced Vector Extensions Programming Reference*, 319433-011 edition, June 2011.

[35] Jussi Kivilinna. crypto-avx2, April 2012. https://gitorious.org/crypto-avx2.

[36] Chris Lomont. Introduction to Intel Advanced Vector Extensions, June 2011. http://software.intel.com/file/37205.

[37] ARM Ltd. NEON SIMD Instruction Set, June 2012. http://www.arm.com/products/processors/technologies/neon.php.

[38] Krystian Matusiewicz, Martin Schlaffer, and Søren S. Thomsen. Grøstl Implementation Guide, March 2012. http://www.groestl.info/groestl-implementation-guide.pdf.

[39] Nikos Mavrogiannopoulos, Phil Sutter, Michael Weiser, and Michal Ludvig. Cryptodev-linux module, April 2012. http://home.gna.org/cryptodev-linux/.

[40] Tilo Müller, Andreas Dewald, and Felix C. Freiling. AESSE: A Cold-boot Resistant Implementation of AES, April 2010. http://eurosys2010-dev.sigops-france.fr/workshops/EuroSec2010/p42-muller.pdf.

[41] National Institute of Standards and Technology. *DES Modes of Operation*, FIPS PUB 81 edition, December 1980.

[42] National Institute of Standards and Technology. *Data Encryption Standard (DES)*, FIPS PUB 46-3 edition, October 1999.

[43] National Institute of Standards and Technology. *Advanced Encryption Standard (AES)*, FIPS PUB 197 edition, November 2001.

[44] National Institute of Standards and Technology. *Recommendation for Block Cipher Modes of Operation*, special publication 800-38a edition, December 2001.

[45] National Institute of Standards and Technology. *Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices*, special publication 800-38e edition, January 2010.

[46] National Institute of Standards and Technology. *Recommendation for Block Cipher Modes of Operation: Three Variants of Ciphertext Stealing for CBC Mode*, addendum to special publication 800-38a edition, October 2010.

[47] Samuel Neves and Jean-Philippe Aumasson. Implementing BLAKE with AVX, AVX2, and XOP, March 2012.

[48] National Institute of Standards and Technology. Announcing Request for Candidate Algorithm Nominations for the Advanced Encryption Standard (AES), September 1997. http://csrc.nist.gov/archive/aes/pre-round1/aes_9709.htm.

[49] National Institute of Standards and Technology. Commerce Department Announces Winner of Global Information Security Competition, October 2000. http://www.nist.gov/public_affairs/releases/g00-176.cfm.

[50] National Institute of Standards and Technology. Cryptographic Hash Algorithm Competition, April 2012. http://csrc.nist.gov/groups/ST/hash/sha-3/index.html.

[51] Ronald L. Rivest, M.J.B. Robshaw, R. Sidney, and Y.L. Yin. The RC6 Block Cipher, August 1998. ftp://ftp.rsasecurity.com/pub/rsalabs/rc6/rc6v11.pdf.

[52] Bruce Schneier. Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish). *Cambridge Security Workshop Proceedings*, pages 191–204, 1993. http://www.schneier.com/paper-blowfish-fse.html.

[53] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. Twofish: A 128-Bit Block Cipher, June 1998. http://www.schneier.com/paper-twofish-paper.pdf.

[54] Claude E. Shannon. Communication Theory of Secrecy Systems. *Bell System Technical Journal*, 28-4:656–715, 1949. http://netlab.cs.ucla.edu/wiki/files/shannon1949.pdf.

[55] Suresh Siddha. x86: add linux kernel support for YMM state, June 2009. http://git.kernel.org/linus/a30469e7921a6dd2067e9e836d7787cfa0105627.

[56] Linus Torvalds. I'm doing a (free) operating system, August 1991. http://groups.google.com/group/comp.os.minix/msg/b813d52cbc5a044b.

[57] Linus Torvalds. Linux Kernel, COPYING, September 2005. http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=blob;f=COPYING.

[58] Linus Torvalds. Linux 3.0 release, July 2011. https://lwn.net/Articles/452531/.

[59] Hongjun Wu. The Hash Function JH, January 2011. http://www3.ntu.edu.sg/home/wuhj/research/jh/jh_round3.pdf.

[60] Herbert Xu. crypto: af_alg - User-space interface for Crypto API, April 2012. http://git.kernel.org/linus/03c8efc1ffeb6b82a22c1af8dd908af349563314.