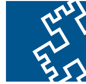


Block Ciphers: Fast Implementations on x86-64 Architecture

University of Oulu
Department of Information Processing Science
Master's Thesis
Jussi Kivilinna
May 20, 2013

Abstract

Encryption is being used more than ever before. It is used to prevent eavesdropping on our communications over cell phone calls and Internet, securing network connections, making e-commerce and e-banking possible and generally hiding information from unwanted eyes. The performance of encryption functions is therefore important as slow working implementation increases costs. At server side faster implementation can reduce the required capacity and on client side it can lower the power usage. Block ciphers are a class of encryption functions that are typically used to encrypt large bulk data, and thus make them a subject of many studies when endeavoring greater performance. The x86-64 architecture is the most dominant processor architecture in server and desktop computers; it has numerous different instruction set extensions, which make the architecture a target of constant new research on fast software implementations. The examined block ciphers – Blowfish, AES, Camellia, Serpent and Twofish – are widely used in various applications and their different designs make them interesting objects of investigation.

Several optimization techniques to speed up implementations have been reported in previous research; such as the use of table look-ups, bit-slicing, byte-slicing and the utilization of "out-of-order" scheduling capabilities. We examine these different techniques and utilize them to construct new implementations of the selected block ciphers. Focus with these new implementations is in modes of operation which allow multiple blocks to be processed in parallel; such as the counter mode. The performance measurements of new implementations were carried out by using the System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives (SUPERCOP) framework on four different processors: AMD K8, AMD K10, Intel Core2 and Intel Sandy-Bridge.

The parallel processing permitted by particular modes of operation can improve performance of a block cipher even with traditional table look-up optimization. Bit-slicing, byte-slicing and word-slicing can be used to parallelize block cipher processing in vector registers. These 'sliced' techniques can improve the throughput of block cipher implementations significantly compared to table look-up based approaches. Our byte-sliced AES-NI/AVX implementation of Camellia reaches the speed of 5.32 cycles per byte on Intel Sandy-Bridge processor, being 2.65 times faster than our two-way table look-up implementation and 3.96 times faster than the implementation found in the OpenSSL library.

Keywords
block cipher, software implementation, bit-slicing, byte-slicing, x86-64, AVX, AES-NI, Blowfish, AES, Camellia, Serpent, Twofish

Foreword

This thesis was written for my Master degree in Information Processing Science at the University of Oulu. This work originally started as a personal interest and a hobby on fast implementations of block ciphers for x86-64 architecture for Linux Kernel. Very often as I worked on different implementations, I would come with new ideas for faster solutions – obviously only to find out that those solutions had already been reported either in academics or open source software communities. This led me to do more extensive research on different reported optimization techniques and then applying techniques on implementing various block ciphers. As result this research study of a hobbyist evolved to the research subject for this thesis.

I want to thank my supervisor Docent Juha Kortelainen for his guidance and advice on the writing process. I also want to thank my love Vuokko, my family and friends for their support during my studies and writing of this thesis.

Jussi Kivilinna

Oulu, May 20, 2013

Notation

$\oplus$	Bitwise exclusive-or operation (XOR)
$\vee$	Bitwise or operation (OR)
$\wedge$	Bitwise and operation (AND)
$\boxplus$	Integer addition
$\ll$	Bitwise left shift
$\gg$	Bitwise right shift
$\lll$	Bitwise rotation to left
$\ggg$	Bitwise rotation to right

Contents

Abstract . . . 3
Foreword . . . 5
Notation . . . 7
Contents . . . 9
1 Introduction . . . 11
1.1 Research question . . . 12
1.2 Limitations . . . 13
2 Background . . . 14
2.1 Principles of encryption . . . 14
2.2 Block ciphers . . . 16
2.2.1 Substitution-permutation networks . . . 17
2.2.2 Feistel ciphers . . . 18
2.3 Block cipher modes . . . 20
2.3.1 Electronic Codebook . . . 20
2.3.2 Cipher Block Chaining . . . 20
2.3.3 Counter Mode . . . 21
2.4 Different block ciphers . . . 22
2.4.1 Blowfish . . . 23
2.4.2 AES . . . 23
2.4.3 Camellia . . . 24
2.4.4 Serpent . . . 25
2.4.5 Twofish . . . 26
2.5 The x86-64 architecture . . . 26
3 Optimization Techniques in the Literature . . . 28
3.1 Table look-ups in general . . . 29
3.2 The use of look-up tables on the x86-64 architecture . . . 30
3.3 Parallel execution of instructions on “out-of-order” CPUs . . . 31
3.4 “Slicing” techniques . . . 32
4 Implementation . . . 34
4.1 Generic approach to block cipher implementation construction . . . 35
4.2 Blowfish . . . 36
4.3 AES . . . 39
4.4 Camellia . . . 41
4.5 Serpent . . . 50
4.6 Twofish . . . 52
5 Evaluation . . . 54
6 Results . . . 58
6.1 Blowfish . . . 59
6.2 AES . . . 60
6.3 Camellia . . . 62
6.4 Serpent . . . 63
6.5 Twofish . . . 64
7 Conclusions . . . 65
References . . . 68
Appendices . . . 73

1 Introduction

Encryption is a part of our everyday life although mostly invisible. It is used to prevent eavesdropping on our communications over cell phone calls and Internet, securing network connections, making e-commerce and e-banking possible and generally hiding information from unwanted eyes. Through the history of civilization, new encryption functions have been constructed as old ones have been broken (Mollin, 2005).

One class of encryption functions are block ciphers. Block ciphers are normally used for bulk encryption tasks, such as encrypting communication channels and for disk and file encryption (Stallings, 2010; Wright, Dave, & Zadok, 2003). A block cipher consists of two functions, encryption function and its inverse, decryption function. Block ciphers process data as fixed-size blocks of bytes (Stallings, 2010). When using a block cipher, the both sending and receiving ends of the encrypted channel need to use the same key. Using the same key to encrypt the same plaintext block results the same ciphertext block (Katz & Lindell, 2008). The most commonly used block cipher today is Rinjdael, which was selected as the winner of National Institute of Standards and Technology (NIST) sponsored contest for new encryption standard in 2000 (Ferguson & Schneier, 2003). After winning of the Advanced Encryption Standard contest, the Rinjdael cipher is now commonly known by the name AES (Daemen & Rijmen, 2002).

Since block ciphers are able to encrypt only exactly fixed-size blocks, they are used with block cipher modes of operation to either encrypt larger sets of data (Ferguson & Schneier, 2003). A commonly used encryption mode for large data sets is the so called cipher block chaining (CBC), which combines plaintext with previous ciphertext block using XOR-operation (Mollin, 2005). This chaining prevents the same plaintext blocks from producing the same ciphertext blocks for sub-blocks of a large data set thus preventing leaking the information of the original plaintext. A downside of the CBC-mode is that encryption is not parallelizable (Katz & Lindell, 2008). That is, the encrypted block depends on the encryption result of the previous block and thus large plaintext cannot be processed, for example, in parallel processors. However, the decryption of CBC-mode does allow the use of parallel blocks and the process can be accelerated by utilizing parallel processing possibilities of both hardware and software (Akdemir et al., 2010). Counter mode (CTR) is used to make the block cipher function as a stream cipher and it is being applied increasingly often (Mollin, 2005; Ferguson & Schneier, 2003). In CTR-mode, the initialization value (IV) is increased by one for each produced block and the produced key-stream is XORed with plaintext to generate the ciphertext. One of the benefits of CTR-mode is that it allows splitting the encryption of IVs to independent steps that can be run in parallel. It is also possible to precompute the key-stream data if the IV is known ahead of encryption and decryption (Stallings, 2010).

As block ciphers are used for bulk encryption for large sets of data, performance is one critical property for them (Robshaw, 1995). One requirement for AES contestants was a higher performance in hardware and software implementations than the previous encryption standard (Daemen & Rijmen, 2002). Better implementations, optimized for specific processor architectures, can bring performance improvements of order of magnitude thus giving higher throughput and, in case throughput bottleneck being

elsewhere, reduce CPU usage (Potlapally, Ravi, Raghunathan, & Jha, 2003). Less CPU usage also results to smaller power usage. Most commonly used ciphers have hardware implementations available and today even on consumer processors (Akdemir et al., 2010). However, software implementations remain important for less commonly used algorithms and for new ciphers in newer encryption standards that will become more used in the future but yet do not have gained enough popularity for wide hardware implementations and acceleration.

The processors of the x86-64 architecture are the engine behind the majority of server and desktop systems (Bryant & O'Hallaron, 2005; Padgett & Kerner, 2007). They are also used to small extent in embedded applications. A good availability and a quite broad availability of different instruction sets make them an interesting platform for new cipher implementations. Vector register instruction sets and out-of-order execution can be used to speed up parallel computation and the first is usually used to speed up video and audio processing (Tran, Cho, & Cho, 2009). Vector instruction sets are also used for fast software implementations of some block ciphers (Gladman, 2000; Hamburg, 2009) and as new instruction sets are added to the x86-64 architecture, new possibilities for even faster implementations are opened (Neves & Aumasson, 2012; Götzfried, 2012*a*).

The block cipher algorithms examined in this thesis will be the following: AES, Blowfish, Camellia, Twofish and Serpent. AES – originally called Rinjdael – is the algorithm most widely used of these five. AES even has instruction level hardware acceleration available in the x86-64 architecture, in the form of the AES-NI instruction set (Gueron, 2009). Rinjdael was the winner of AES standardization process stated by NIST and it is the recommended block cipher for use by Cryptography Research and Evaluation Committees (CRYPTREC) and New European Schemes for Signatures, Integrity and Encryption project (NESSIE) (The CRYPTREC Advisory Committee, 2003; NESSIE consortium, 2003). Camellia is a block cipher that is also recommended by both CRYPTREC and NESSIE. Twofish and Serpent are AES finalists and are used today to some extent (Ferguson & Schneier, 2003). Blowfish, one of the first patent-free and royal-free algorithms, was published in 1994 and has ever since gained large popularity (Schneier, 1994*b*; Mollin, 2005).

1.1 Research question

Our research problem is stated as follows: How to use advanced features of the x86-64 architecture (such as out-of-order execution) and available vector register instruction set extensions (SSE2, AVX, AES-NI) to construct faster implementations of commonly used block ciphers (Blowfish, AES, Twofish, Serpent, Camellia)?

Out-of-order execution allows the utilization of instruction level parallelism to speed up the code execution. It allows the processor to execute those instructions that do not have resource or register dependencies between each other to be executed simultaneously (Stallings, 2009). In other words, the processor can look ahead of instruction queue and execute operations in advance. Block cipher algorithms tend to have a large number of inter-algorithm data dependencies. These algorithms contain computations that cannot be performed in parallel since the encryption is typically highly sequential. As a result, the implementations of block cipher algorithms do not typically exhibit great amount of instruction-level parallelism. Therefore, block cipher algorithms typically cannot benefit from the out-of-order parallelism with conventional implementation approach (Matsui, 2006). However, we can bypass these restrictions by taking two or more blocks of data to process at the same time. Since two blocks do not have dependencies, we can attain

higher throughput to encryption (Matsui, 2006). Because of limited processor register resources, such approach cannot practically be implemented in a higher level programming language. More fine-grained instruction ordering, data/register handling and processor architecture specificity than is possible to generate through compilers is required (Fog, 2012*b*). Therefore implementations will be constructed in assembly language.

Vector register instructions can be used to speed up batch processing. The data is loaded into vector register slots and the same operation is performed on each slot with one instruction (Stallings, 2009). Vector registers are not usually used for block cipher implementation. This is because the common implementation practice is to use precalculated lookup tables for performing the key cryptographic primitives of algorithms. The problem with the vector register instruction set is that they seldom have vectorized table lookups available (Intel Corp., 2013*b*). However, some ciphers can be implemented faster with vector registers, using advanced techniques such as bit-slicing (Biham, 1997; Matsui, 2006). Anyways, the possibility to use such techniques lies within the inner structure and the design of the cipher algorithm in question. On the x86-64 architecture there is also the AES-NI instruction set that implements primitives of the AES cipher (Gueron, 2009). Some of these primitives or close relatives of them are used in other ciphers, such as in Camellia (Satoh & Morioka, 2003). Could the AES-NI instructions be used to construct a faster implementation of Camellia?

In our research we use the constructive research method to build fast assembly language implementations of the selected block ciphers. The implementation performance is evaluated using the SUPERCOP (System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives) toolkit developed by VAMPIRE laboratory (Virtual Application and Implementation Research Lab) (Bernstein & Lange, 2013). The predecessor of the SUPERCOP framework, namely eSTREAM, has been used in the previous research to evaluate implementation performance. It is being recommended to be used to get constant and comparable results (Bernstein & Schwabe, 2008; Käsper & Schwabe, 2009). As different processor models have different performance characteristics (Fog, 2012*a*), results will be gathered from several processors, that are available from different vendors. The following processor architectures were available at the time of measurements for this thesis: Intel Core2, Intel Sandy-Bridge, AMD K8 and AMD K10.

1.2 Limitations

To be able to do parallel encryption with block ciphers, one need to use an encryption mode in which there are no dependencies between encrypted blocks. For example in the CBC-mode, the data block that is to be encrypted is combined with the previously encrypted block (Stallings, 2010). This creates data dependency and prevents the parallel processing of those blocks. However, the decryption in the CBC-mode does not require the previous or the next block to be decrypted first. Therefore, the decryption in the CBC-mode can be sped up by exploiting available parallelism (Akdemir et al., 2010). The use of parallelizable encryption modes is increasing as fast hardware implementation also benefits from parallelism (Mollin, 2005). One parallelizable mode is the counter mode (CTR) that changes a block cipher to work like a stream cipher (Stallings, 2010). The CTR-mode is also the mode that is used in the SUPERCOP framework for the AES block cipher and therefore this mode will be also used in our thesis in performance evaluation.

The implementation will be written in x86-64 assembly language and therefore it cannot be used as such on other processor architectures. Optimization concepts such as utilizing out-

of-order execution and vector registers, however, can be used to construct implementations on other architectures, if only such features are available. Normally those implementations that use new instruction set extensions cannot be applied to those older processors that do not support the extended instruction sets. Certain older processors realize new instruction sets in a limited way so those implementations can be considerably slower than on newer processors (Fog, 2012*a*).

2 Background

In this section, a short introduction into encryption functions and block ciphers are given. We also discuss briefly about the x86-64 architecture but postpone the actual optimization techniques, methods and details with the x86-64 architecture to the next section.

First we look at the definitions of the terms used. *Plaintext* is the original message in a form that is understandable by all parties, including the sender of message and the receiver. Plaintext is also understandable by anyone eavesdropping on the communication between the two (Anderson, 2008). *Ciphertext* on the other hand is the encrypted message that should not be understood by other parties except the sender and the receiver of the message. Plaintext is converted to ciphertext by the process of *encryption* and ciphertext is converted back to plaintext by *decryption* (Stallings, 2010). The system for performing the encryption and the decryption of messages is called a *cipher* or a *cryptographic system*. The science and art of constructing cryptographic systems is called *cryptography*. The attacker tries to find out the original plaintext message (or the key) without knowing all the details of the system. This area of scientific research is called *cryptanalysis* and the research area combining cryptography and cryptanalysis is called *cryptology* (Stallings, 2010). Several different cryptographic primitives exist, such as *stream ciphers* and *block ciphers*. Now *key* in cryptography is an entity that is used to customize the used cipher so that only the parties that have the correct key can expose the original plaintext message from the encrypted one (Mollin, 2005). A cryptographic system that uses the same key in both the sending and receiving sides of the communication is called a *shared-key*, *secret-key* or *symmetric-key* cipher. A system that uses different keys for encryption and decryption is called a *public-key* or an *asymmetric-key* cipher (Anderson, 2008).

2.1 Principles of encryption

In symmetric-key cryptography the same key is used for both encryption and decryption. The encryption function for symmetric-key cipher is a bijective function transforming the input plaintext message to ciphertext. The plaintext message belongs to finite message space M and the ciphertext message to finite ciphertext message space C (Mollin, 2005). A keyed encryption function can be presented as

$$c := E_e(m);\ m \in M, c \in C, e \in K_e$$

where e is the encryption key, m is the input plaintext message and c the output ciphertext message. The key e is selected from the available keyspace K_e and determines how the function E_k maps plaintext messages to ciphertext messages (Mollin, 2005). A good cipher should perform a different transform with each different key from the keyspace (Ferguson & Schneier, 2003).

Likewise the decryption function is bijective function for transforming the input ciphertext to a plaintext message. The decryption function can be presented as

$$m := D_d(c);\ m \in M, c \in C, d \in K_d$$

where d in the decryption key, c is the input ciphertext message and m is the output plaintext message. The decryption function with the correct corresponding decryption key d is an inverse of the encryption function with the corresponding encryption key e,

$$m := D_d(c) = E_e^{-1}(c).$$

Combining these corresponding keys with the encryption and decryption functions determines the cryptographic system or cipher. In a symmetric-key cipher the encryption key can easily be transformed into the decryption key and vice versa. Typically the encryption key and the decryption keys are the same, $e = d$, thus clarifying the use of the term *symmetric-key* (Mollin, 2005).

For asymmetric-key ciphers, the decryption key is different from the encryption key and the encryption key cannot easily be transformed into the decryption key without some extra knowledge (Katz & Lindell, 2008). Asymmetric-key ciphers allow the encryption key to be made public so anyone having access to the public key may produce a ciphertext message that only the holder of the “private” decryption key can transform to readable plaintext (Mollin, 2005). The symmetric-key encryption, contrariwise, requires the key to be shared by some method between the sender and the receiver, without making the key public (Stallings, 2010). In this light the asymmetric-key encryption might seem more desirable for communication as the encryption keys can be made public without risking critical information being leaked. However, asymmetric-key ciphers require much computational processing and as a result are at least 2 to 3 orders of magnitude slower than symmetric-key ciphers (Katz & Lindell, 2008). Therefore, the symmetric-key ciphers are used for the bulk encryption of large data. Asymmetric-key ciphers and their related key exchange schemes are used to transmit the shared symmetric-key between the sending and receiving sides (Stallings, 2010).

Modern ciphers are designed so that the security of the encryption scheme depends only on the secrecy of the encryption key. The cipher algorithm must be designed in such a way that the algorithm can be allowed to fall into the hands of the enemy without risking the security. This is one of the cipher design principles that was presented by Kerckhoffs (1883) and it is now known as Kerckhoffs’ principle (Mollin, 2005). The reason why such a principle is valuable is that algorithms are hard to change after the deployment to use in software and hardware. By making security depend on the key, one can use the same algorithm for a much longer time and make the use of encryption more practical (Ferguson & Schneier, 2003). By allowing the cipher algorithm to be public, the algorithm also gains attention from the cryptanalysis research. In this way the algorithm gets public scrutiny and the possible weaknesses of the cipher can be revealed before the cipher is deployed into use (Ferguson & Schneier, 2003; Mollin, 2005).

Different classes of symmetric-key encryption systems exist, such as block ciphers and stream ciphers (Stallings, 2010). Block ciphers encrypt fixed-size plaintext messages to ciphertext messages of the same size (Mollin, 2005). Stream ciphers on the other hand encrypt messages one bit or byte at a time (Stallings, 2010). Typically stream ciphers generate a pseudo-random *key stream* based on the used encryption key, which is then XORed with the plaintext message to produce the ciphertext message (Katz & Lindell, 2008). We shall detail the block ciphers on the next section. Later in the section 2.3 we look at a way to construct a stream cipher from a block cipher.

2.2 Block ciphers

Block ciphers can be considered to be the basic building block of the symmetric-key encryption schemes, rather than being used only to encrypt messages (Katz & Lindell, 2008). As the performance of encryption is critical especially on the server side of communications, good block ciphers are designed to be fast in software and hardware (Ferguson & Schneier, 2003; Robshaw, 1995). The input for a block cipher encryption is a fixed-size plaintext message, which is transformed to a same size ciphertext message. This fixed message length is called the *block size*. The most common block sizes used in the current block ciphers are 64 bits and 128 bits (Stallings, 2010). A block cipher system contains the encryption function, the decryption function and the encryption key. The length of the key can vary and the accepted key lengths depend on the cipher algorithm (Katz & Lindell, 2008). For each different key the block cipher encryption function will perform different permutation on the same input plaintext block. With the encryption key being secret to the outsiders, the permutation appears as random. Therefore block ciphers can be described as a *pseudo-random permutation*. The reason why block ciphers are pseudo-random instead of truly random is that with knowledge of the key, the ciphertext is no longer random (Katz & Lindell, 2008).

Although the construction of the encryption function is different for each cipher algorithm, there are founding principles on which the design and the construction of the block ciphers are based (Stallings, 2010). The fundamental cryptographic techniques of *confusion* and *diffusion* were introduced by Shannon (1949). The goal of confusion in the cipher design is to make the statistical analysis of the ciphertext difficult for cryptanalysts. This is achieved by obscuring the ciphertext so that statistical redundancies or patterns disappear. Strong confusion in a cipher can be accomplished by using a complex non-linear substitution algorithm (Mollin, 2005). The goal of diffusion is to spread the information of the plaintext block to whole width of the ciphertext block. In a binary cipher, this means that the change in one plaintext bit results to a change in the majority of the ciphertext bits (Stallings, 2010). This spreading of the plaintext information over the whole width of the ciphertext makes cryptanalysis much harder. Confusion attempts to prevent successful search for redundancies in the plaintext through observing statistical patterns from the ciphertext (Mollin, 2005).

Strong block ciphers are typically constructed from weaker parts, *rounds* (Ferguson & Schneier, 2003). The structure of such strong block cipher is such that the weak block cipher – round – is repeated several times with different *sub-keys* or *round keys*. These multiple round keys are derived from the encryption key of the block cipher, which is sometimes called the *master key*. The process of constructing the sub-keys from the master key is called *key schedule*. The strength of block cipher against cryptanalysis grows with the number of rounds as the amount of diffusion and confusion increases (Katz & Lindell, 2008).

The two most common classes of block ciphers are substitution-permutation networks and Feistel ciphers (Katz & Lindell, 2008; Ferguson & Schneier, 2003). We will detail these two different block cipher constructs in the following two subsections. For a note, we call the intermediate input data moving through between different rounds as *the block state*. The input plaintext or ciphertext block to a block cipher is the first block state, and it is replaced by the round with new block state. This new block state may be completely replaced or contain part of the previous block state unmodified. The block state at the end of the encryption (or the decryption) process is then output as the ciphertext (or the plaintext) block.

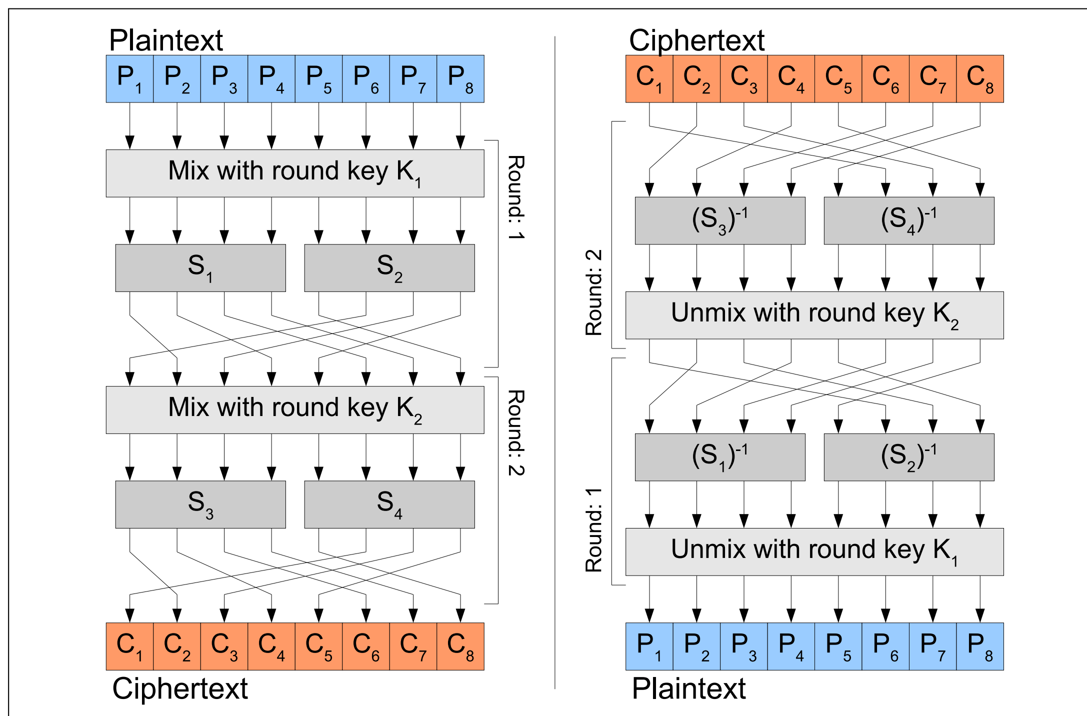


Figure 1. A two round substitution-permutation network cipher. The encryption function is shown on the left and the decryption on the right.

2.2.1 Substitution-permutation networks

A practical adaptation of the confusion and the diffusion can be obtained with substitution-permutation networks. The round functions in substitution-permutation networks do not depend on the key, but are predefined (Katz & Lindell, 2008). These round functions performing the fixed substitution are called as *S-boxes*. The S-boxes are functions that take a small portion from the input of the round. In a substitution phase of the round, multiple of these S-box functions are applied to the round input and the block state is substituted by new values. The permutation phase spreads the bits of the substituted values across the whole block by simply reordering bits throughout the full width of block. As the S-boxes and the permutation phase are fixed and thus do not depend on the key, one more phase is needed to make the substitution-permutation network secrecy to depend on the key. For each round, a sub-key is produced by the key schedule from the master key. These sub-keys are mixed with the block state in each round, making the whole cipher perform different pseudo-random permutation for each different master key (Katz & Lindell, 2008).

Figure 1 shows an example of a two round substitution-permutation network cipher with eight bit block size. This example illustrates the design principles of the substitution-permutation networks. As seen in our example, the decryption S-box functions in the decryption are the inverse functions of the encryption S-boxes. Therefore, the S-boxes used in substitution-permutation networks must be invertible (Katz & Lindell, 2008). Also the permutation phase of decryption is the inverse of the permutation function in the encryption. The key can be mixed with the block state with exclusive-or operation. In decryption, the same exclusive-or operation can then be used to cancel the effect of mixing made in encryption. So the both mixing and unmixing phases are realized with the same XOR operator (Katz & Lindell, 2008). The other option is to use another operator pair for the mixing and unmixing such as arithmetic addition and subtraction. Obviously the order of the operations in the decryption is reversed compared to that of the encryption. Also the

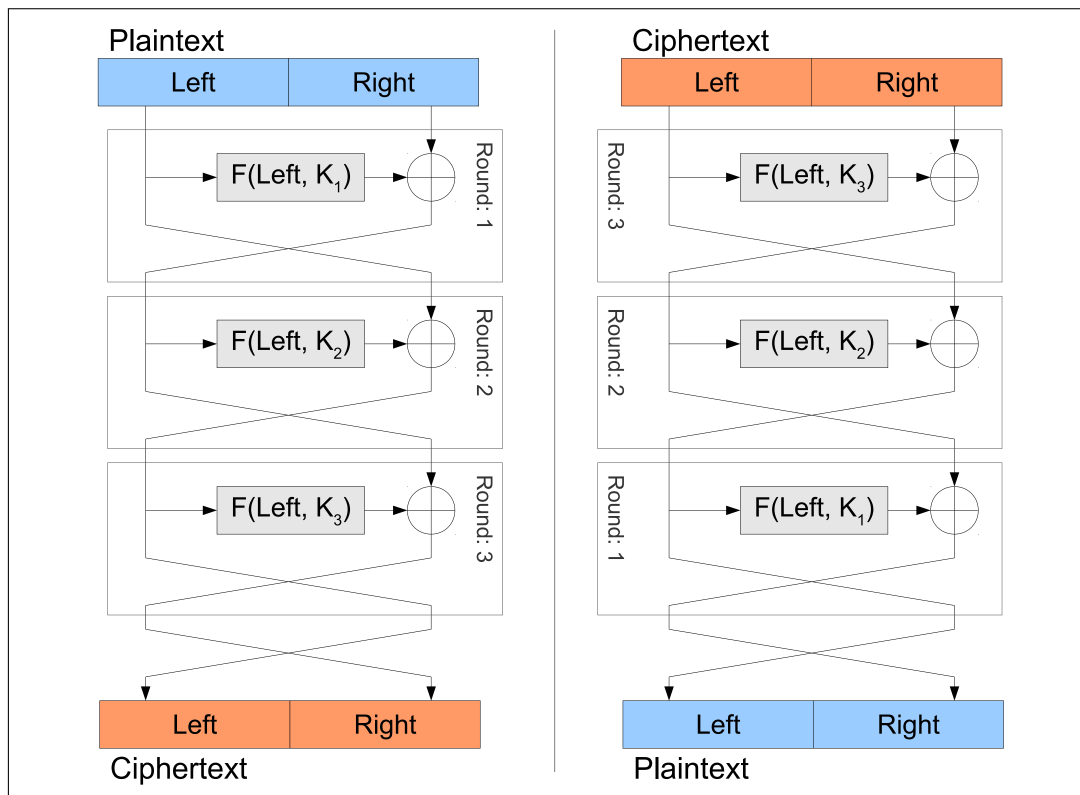


Figure 2. A three round Feistel network with both encryption and decryption illustrated. The round keys K_i for each round is used as the second input for round function F. The symbol $\oplus$ means bitwise exclusive or – XOR – operation. The encryption function is shown on the left and the decryption on the right.

decryption operations are the inverses of the encryption. This is what makes the whole decryption function inverse of the encryption function.

The avalanche effect is an important design principle for the substitution-permutation networks and the block ciphers in general (Katz & Lindell, 2008). The output of block cipher must be largely changed when input is changed even by the smallest amount. With a strong block cipher, a change of one bit in input plaintext block should result a change of the half of the output bits on average. In substitution-permutation network this can be realized by two design principles for S-boxes and permutation phases. The S-boxes should change at least two bits in the output if a single input bit is changed. The permutation phase should spread the block bits so that the output bits of each S-box function is distributed to input of different S-boxes for the next round (Katz & Lindell, 2008).

2.2.2 Feistel ciphers

Instead of the substitution-permutation network structure, many current block cipher designs use a more complex Feistel network structure (Stallings, 2010). This structure was invented in 1950s and 1960s by Feistel and his team (Anderson, 2008). What makes this construct important is that it allows building an invertible block cipher by using non-invertible components (Katz & Lindell, 2008).

Feistel cipher has a special ladder structure as seen in Figure 2. For a Feistel cipher the input block is split into two halves, called the left and the right side. For each round i, a round function $F(\cdot\,;K_i)$ is invoked with the left side as the input. The output of round function is mixed with the right side, using exclusive-or operation or another mix-unmix

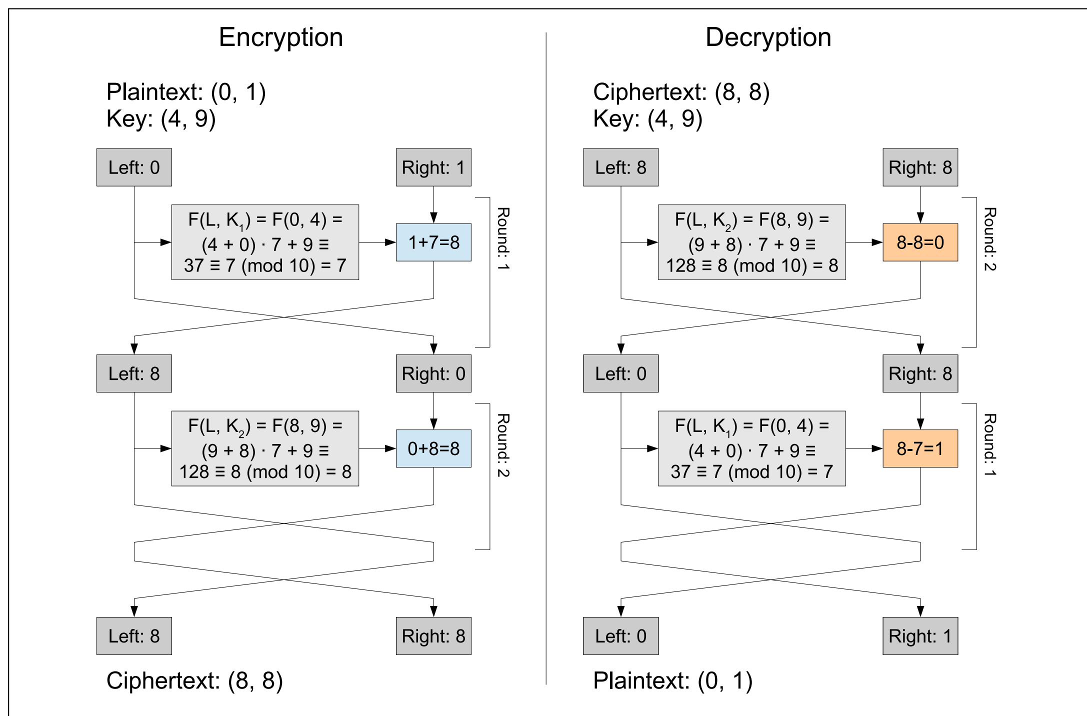


Figure 3. An example of the operation of a two round Feistel cipher.

operator pair (Anderson, 2008). At the end of the round, the left side and the new right side values are swapped, so that the new combined right side becomes the input for next round function $F(\cdot; K_{i+1})$. The round function is typically different for each round and the difference between rounds is commonly achieved by supplying the round function a different round key. The round function is built from the same building blocks as were described for the substitution-permutation networks – S-boxes, permutation and the key schedule (Katz & Lindell, 2008).

An important property of the Feistel cipher structure is that the round function F does not need to be an invertible function. As a result, the round function can be constructed from S-boxes that are non-invertible giving the designer of cipher more freedom. Also there is no requirement for the round function to be based on S-boxes, since the Feistel cipher structure can handle any kind of round function design (Katz & Lindell, 2008). The reason for this property is that the high-level structure of Feistel network preserves the left side that was used for the round function input in the previous round (Stallings, 2010). Because of this, we can apply the round functions in a reversed order, using the last preserved left side from encryption as the first left side for decryption. Thus the result of the first round function in decryption is the same as the result of encryption round function in encryption. By combining this round function result with the proper unmix operator, typically exclusive-or, will restore the next left side for decryption (Stallings, 2010). When the round functions of a Feistel cipher are only modified by the round key, the decryption can be implemented efficiently by using the same encryption implementation with just the round keys being fed to algorithm in reversed order (Robshaw, 1995).

Figure 3 shows the functioning of a simple two round Feistel cipher. The plaintext and ciphertext are in the form of two digits, with values from the range $0\ldots9$. The master key is also defined as a pair of two digits. The key schedule is such that the first digit of the master key is used as the first round key K_1 and the second digit as the second round key K_2. The round function in our simple example is

$$F(L, K_i) = (K_i + L) \times 7 + 9 \ (mod\ 10)$$

where L is the left side block state for the round and K_i is the round key. For combining the result of round function, we use addition and subtraction in modulo 10 as the mixing-unmixing pair. The subtraction in decryption cancels the addition done in encryption.

2.3 Block cipher modes

As block ciphers process only blocks that have a predefined fixed size, different encryption schemes called *modes of operation* are needed to encrypt plaintext messages larger than block size (Stallings, 2010; Ferguson & Schneier, 2003). In this section, we look at three different common *block cipher modes* – electronic codebook, cipher block chaining and counter mode.

2.3.1 Electronic Codebook

Electronic codebook (ECB) is the simplest block cipher mode (Stallings, 2010). In the ECB mode, the large input data is split to multiple plaintext chunks with the width of the block size of the used block cipher. Each of these plaintext chunks are then encrypted separately using the block cipher with a selected master key (Ferguson & Schneier, 2003). Thus the encryption for ECB mode can be written as

$$c_i := E_k(p_i);\ i = 1, 2, \ldots, n$$

where n is the total number of blocks, $p_1, p_2, \ldots, p_n$ are the plaintext blocks, and k the master key.

The decryption is performed simply by using the decryption function of the block cipher on the ciphertext blocks that were generated by the encryption. The ECB mode has the property that equal plaintext blocks are mapped into equal ciphertext blocks (Stallings, 2010). In such a case ECB mode can leak information from the plaintext, cannot considered to be secure, and thus should not be used (Ferguson & Schneier, 2003; Katz & Lindell, 2008).

2.3.2 Cipher Block Chaining

The cipher block chaining mode overcomes the security issue of ECB mode, by using the ciphertext from previous encrypted block as additional input to the current encryption (Stallings, 2010). This way the CBC mode produces different ciphertext blocks even if plaintext blocks, that are the same, are encrypted repeatedly (Katz & Lindell, 2008). The CBC mode encryption is defined as

$$c_i := E_k(p_i \oplus c_{i-1});\ i = 1, 2, \ldots, n$$

where n is the total number of blocks, $p_1, p_2, \ldots, p_n$ are the plaintext blocks, k the master key, and $c_1, c_2, \ldots, c_n$ are the ciphertext blocks. The feedback from previous processed block randomizes the input to the block cipher, which makes the cryptanalysis harder (Ferguson & Schneier, 2003).

For the first processed plaintext block p_1, a *initialization value* (IV) is used as the ciphertext block c_0. Randomized initialization values should be used because deterministic initialization values (fixed IVs and counter IVs) make CBC mode insecure in similar way as the ECB mode (Ferguson & Schneier, 2003; Katz & Lindell, 2008).

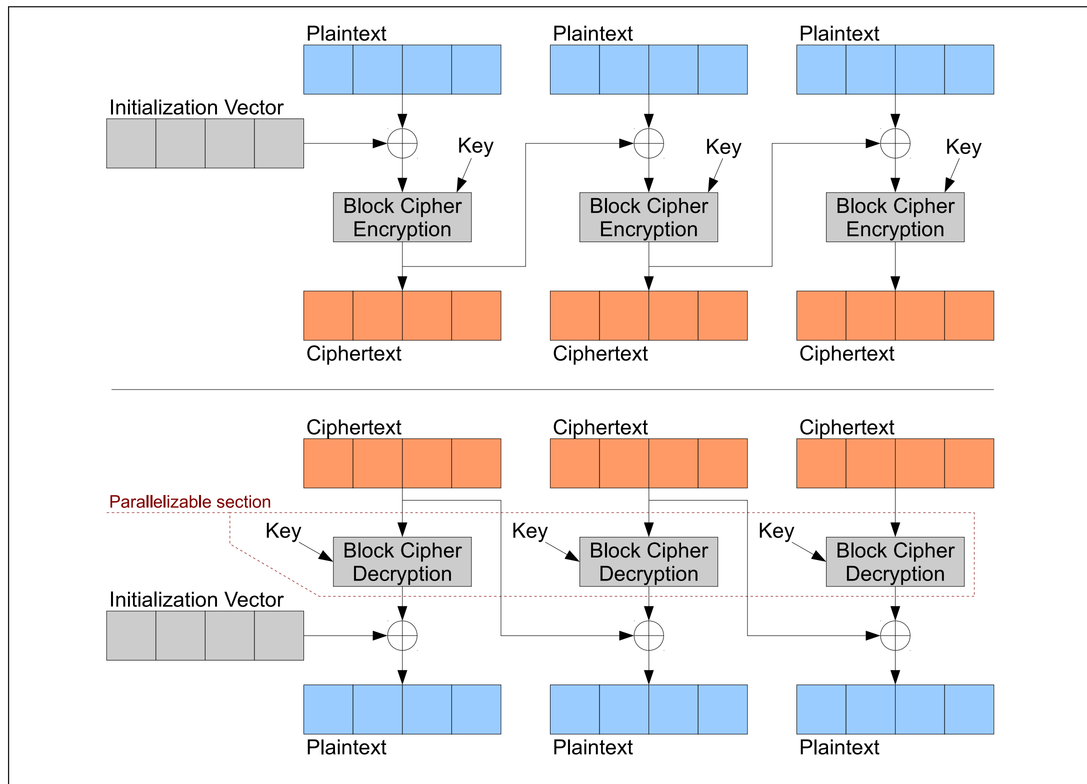


Figure 4. Cipher block chaining mode encryption and decryption with the parallelizable section in decryption high-lighted. The encryption is shown above and the decryption below.

Figure 4 illustrates the encryption and decryption in the CBC mode. As the encryption uses the previous ciphertext as feedback, the encryption cannot be parallelized at block cipher level. However in the decryption all ciphertext is already available, so CBC mode decryption can implemented in parallelized manner, processing multiple blocks in parallel (Akdemir et al., 2010).

2.3.3 Counter Mode

Figure 5 shows the operation of *the counter mode* in which the initialization vector is set to some random value and then used as counter (Katz & Lindell, 2008). For each block, the counter value is increased by one and this counter value is encrypted. Sometimes a part of the IV is initialized with a random number that is used only once. Such 'used only once' value is also called as *nonce*. With 128-bit block size, one could set the upper 64-bits of the IV with a nonce value and a 64-bit counter is placed at the lower 64-bits. The CTR mode encryption with nonce can formulated as follows (Ferguson & Schneier, 2003):

$$s_i := E_k(nonce + (i-1));\ i = 1, 2, \ldots, n$$

$$c_i := p_i \oplus s_i$$

where n is the total number of blocks, k the master key, $p_1, p_2, \ldots, p_i$ are the plaintext blocks, $c_1, c_2, \ldots, c_n$ are the ciphertext blocks and $s_1, s_2, \ldots, s_i$ the so called *key stream*.

From the operation of the counter mode, we can observe that the encryption of the counter value is separated from the handling of the plaintext message and that the plaintext does not affect the encryption of the counter value. The encryption phase generates a key stream based on the randomized initialization vector and the master key. This key stream is then mixed with the plaintext message using exclusive-or operation to produce the ciphertext

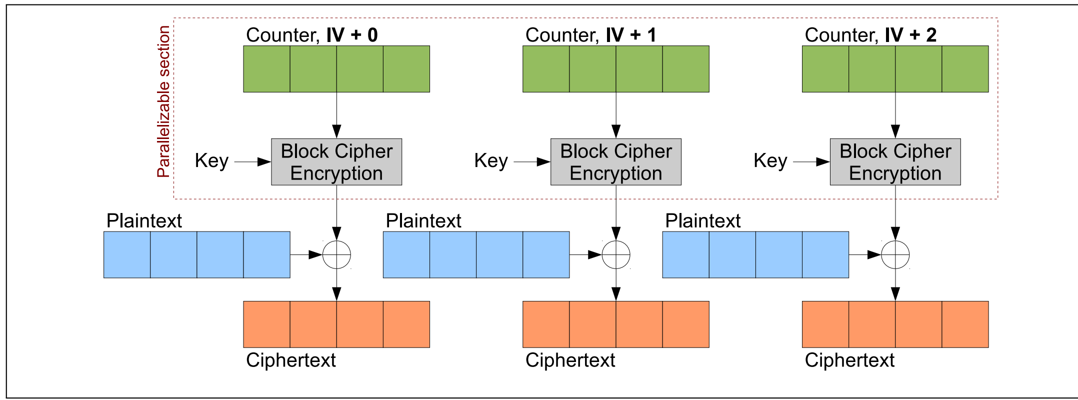


Figure 5. Counter mode with the parallelizable section in high-lighted.

message. This is what makes the counter mode a stream cipher and allows the use of any block cipher in the stream cipher construction (Katz & Lindell, 2008; Mollin, 2005).

The decryption in counter mode is exactly the same as the encryption. The same initialization vector and master key are used to generate the same key stream which is then applied to the ciphertext message with exclusive-or operation (Mollin, 2005). The exclusive-or operation has the property that if the same value is XORed twice, these operations cancel each out. Thus for decryption we get the following result:

$$s_i := E_k(nonce + (i - 1));\ i = 1, 2, \ldots, n$$

$$p_i := c_i \oplus s_i = (p_i \oplus s_i) \oplus s_i = p_i \oplus (s_i \oplus s_i) = p_i \oplus 0 = p_i$$

The counter mode has several interesting benefits that have made it popular in past years (Mollin, 2005; Stallings, 2010). From the performance point of view, counter mode has the clear advantage that the encryption of the counter values do not depend on the previous ciphertext or the key stream blocks. This allows the utilization of parallel encryption to exploit parallel execution capabilities of the processor. Since the decryption is the same as encryption, the implementation of the counter mode is simpler than that of the ECB or CBC modes. When utilizing the counter mode, there is no need to implement the decryption of the underlying block cipher (Stallings, 2010). The counter mode has also being proven to be secure, if a random initialization vector is used and the underlying block cipher is secure (Katz & Lindell, 2008).

2.4 Different block ciphers

In this subsection, we give short descriptions of the five block ciphers that are implemented in this thesis. Each of these ciphers has a structure of its own; this variation makes a more detailed examination interesting from fast implementation point of view. Each of these block ciphers is used in practice at least to some extent. AES and Camellia have been endorsed by different standardization organizations (Ferguson & Schneier, 2003; The CRYPTREC Advisory Committee, 2003; NESSIE consortium, 2003). AES is widely used in different embedded applications such as encrypted Wi-Fi connections (Mathews & Hunt, 2007). Blowfish, Twofish and Serpent are also used in several different cryptographic products (Schneier, 2013*a*, 2013*b*; Fruhwirth, 2011; TrueCrypt Foundation, 2013).

2.4.1 Blowfish

Blowfish was designed by Schneier (1994*b*) and it was one of the first ciphers to be published as patent-free and royal-free (Schneier, 1994*b*; Mollin, 2005). As a result, Blowfish has gained popularity and it is utilized by various products (Mollin, 2005).

Blowfish is a Feistel cipher with block size of 64 bits or, in the other words, 8 bytes. The supported master key lengths for Blowfish range from 32 to 448 bits. The key schedule for Blowfish is performed before encryption and it generates four 8×32 -bit arrays that are used as the four S-boxes by the round function. This makes the S-boxes directly dependent of the master key. Additional eighteen 32-bit round keys are also generated by the key schedule (Schneier, 1994*b*). The round structure of Blowfish is shown in Figure 10. The decryption can be executed with the encryption function just by reversing the order of the round keys. Thus the implementation of Blowfish is in this sense easy because implementing the encryption function also yields the implementation of the decryption function (Mollin, 2005).

2.4.2 AES

In 2000, the Rijndael cipher was announced as the winner of the Advance Encryption Standard contest (Katz & Lindell, 2008). The contest was held by the National Institute of Standards and Technology for the search of new standard block cipher for the U.S. government (Ferguson & Schneier, 2003). Fifteen proposals were submitted by the cryptographic community, out of which the five finalists were selected for the final round (Ferguson & Schneier, 2003). As the result of winning the contest, the Rijndeal cipher is now known as AES. However, the Rijndael cipher is not exactly the same as the standardized version of AES. The Rijndael cipher supports three different block sizes: 128, 192 and 256 bits (Daemen & Rijmen, 2000). However, the standard AES cipher defines only 128 bits block size (Katz & Lindell, 2008). The AES cipher supports three different key lengths that were required by the NIST: 128, 192 and 256 bits (Daemen & Rijmen, 2002). Next we will give short overview of the structure of the AES cipher of 128 bits block size.

The structure of AES encryption function can be seen as a substitution-permutation network (Katz & Lindell, 2008). The SP-network structure is illustrated in Figure 1. The 128-bit round keys are produced by key schedule from the master key. The round keys are mixed in the round function with the block state using exclusive-or. This key-mixing phase of round function is called `AddRoundKey`. The round function performs the substitution phase by passing the sixteen bytes of the block state through an 8×8-bit S-box function. This parallel S-box phase is called `SubBytes`. The S-box of AES performs multiplicative inverse in particular finite field $GF(2^8)$ and affine transform in the form of bit-matrix multiplication. As a result, the S-box function is a bijection as required for a substitution-permutation network construct. The inverse of S-box for decryption is constructed from the inverse of the affine transform and the same multiplicative inverse in $GF(2^8)$ (Stallings, 2010). The substitution phase of decryption that performs the sixteen parallel inverse S-box operations is called `InvSubBytes` (Stallings, 2010).

The permutation phase of AES is split to two different phases, `ShiftRows` and `MixColumns` (Katz & Lindell, 2008). Figure 6 illustrates these phases. For these phases, the block state can be seen as 4×4-byte matrix. The `ShiftRows` rotates the values in the state matrix by different amounts along the rows. The first row is not rotated (or can be said to be

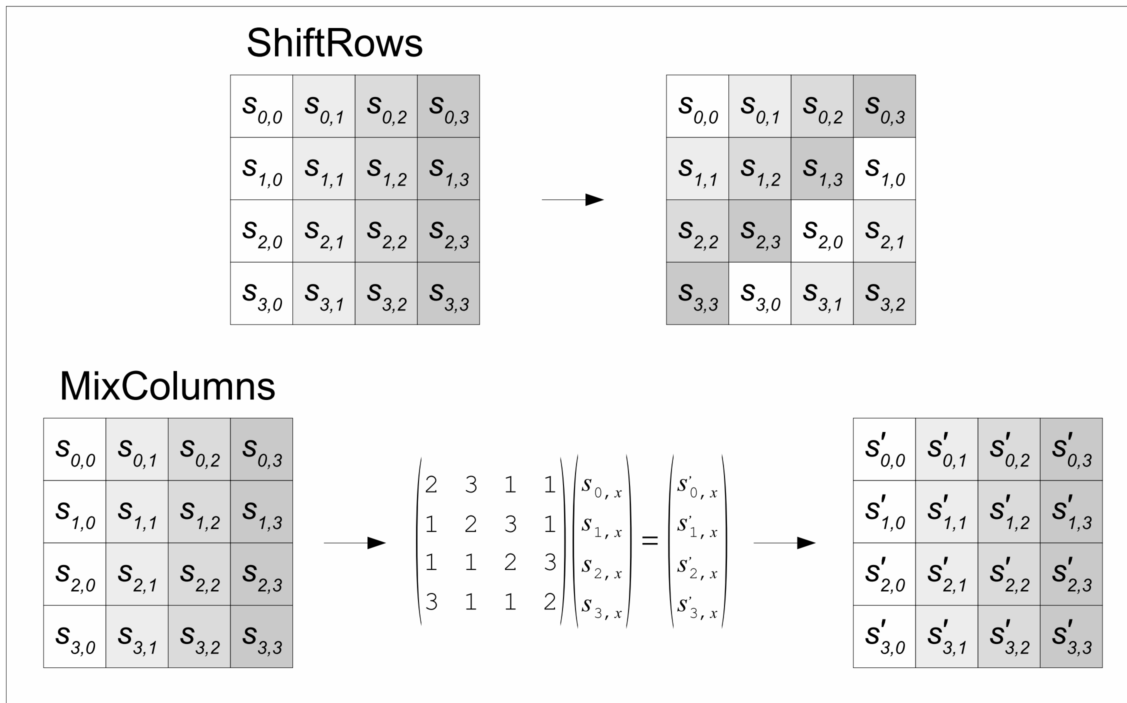


Figure 6. The permutation phases of AES encryption, ShiftRows and MixColumns.

rotated by zero places), the second row by one place, the third by two and the last row by three places. The `MixColumns` phase performs matrix multiplication in the particular finite field $GF(2^8)$ with each column individually as input and output. Both of these operations can be reversed and these inverses used in the decryption are called `InvShiftRows` and `InvMixColumns` (Stallings, 2010).

The AES encryption process starts with initial transformation, which is an additional `AddRoundKey` operation (Stallings, 2010). The round function constructed from the different phases presented previously and are executed in following order: `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey`. The number of rounds depends on the length of the used master key. This round function is repeated nine times with 128-bit master key, eleven times with 192-bit key and thirteen times with 256-bit master key. After these round functions, an additional last round function is performed. The last round function does not have the `MixColumns` phase. The decryption can be performed by reversing the order of operations and replacing the operations with their inverses (Stallings, 2010).

2.4.3 Camellia

Camellia is a block cipher developed by cryptography researchers at Nippon Telegraph and Telephone Corporation and Mitsubishi Electric Corporation (Aoki et al., 2001*a*). Camellia as well as AES has been selected by Cryptography Research and Evaluation Committees (CRYPTOREC) and New European Schemes for Signatures, Integrity and Encryption project (NESSIE) as recommended block cipher for use (The CRYPTREC Advisory Committee, 2003; NESSIE consortium, 2003). Camellia is also included in ISO/IEC standard (International Organization for Standardization, 2010).

Camellia is a Feistel cipher with 128 bits block size and it supports the same key-sizes that are supported by AES – 128, 192 and 256 bits (Aoki et al., 2001*a*). The structure of the encryption function follows the typical ladder construction of Feistel ciphers. The input block state is split to two 64-bit halves and in each round the left side is passed as

parameter to the round function, F-function. The result of the round function is mixed with the right side using exclusive-or operation and the two sides are swapped. The Camellia round function is illustrated in Figure 17. As Camellia is a Feistel cipher, the encryption function can be used as the decryption function simply by changing the order of the round keys (Aoki et al., 2001*a*).

The round function performs round key addition with exclusive-or which is followed by S-box phase, S-function. Camellia uses four different 8×8-bit S-boxes that perform multiplicative inverse in particular finite field $GF((2^4)^2)$ and affine transforms in $GF(2)$ (Aoki et al., 2001*a*; Satoh & Morioka, 2003). The four S-boxes differ in the way the input and output is rotated. After S-function phase, the output bytes from S-function are mixed with each other in P-function using exclusive-or operation. As such, we can notice that the round function F is constructed as substitution-permutation network, where S-function performs the substitution and P-function performs the permutation (Aoki et al., 2001*a*).

The regular Feistel structure in the encryption function is interrupted by FL- and FL^{-1}-functions (Aoki et al., 2001*b*). This non-Feistel round is performed every sixth round and its goal is to make future unknown attacks on the cipher harder (Aoki et al., 2001*a*). In this round, the left side of block state is passed to FL-function and the right side to FL^{-1} function. The use of function and its inverse as a pair allows the same non-Feistel round to be used for decryption without change (Aoki et al., 2001*b*). Figure 14 shows the structure of our implementation of non-Feistel round.

2.4.4 Serpent

The Serpent cipher was one of the five finalists in the AES contest (Ferguson & Schneier, 2003). As all the ciphers in the AES contest, Serpent has 128-bit block size and supports key sizes of 128, 192 and 256 bits. The Serpent encryption function uses the substitution-permutation network structure. Serpent is said to be designed with a stronger emphasis on security, whereas AES puts emphasis on efficiency. As a result, Serpent is typically slower on software that AES (Ferguson & Schneier, 2003).

The round function has the typical phases of a SP-network, round key mixing, substitution phase utilizing S-boxes and a linear transformation as permutation phase (Anderson, Biham, & Knudsen, 1998). Round keys delivered from the master key are mixed with exclusive-or operation. The substitution phase applies S-boxes in parallel to the block state as seen in our example of SP-network in Figure 1. The permutation phase of a Serpent round function mixes the bits of block state so that each input bit of S-boxes in the next round will be from the different S-boxes of current round (Anderson et al., 1998). The decryption for Serpent is built from the inverses of the encryption phases that are applied in reversed order.

In the substitution phase of Serpent round function, the block state is split into 32 four bit parts and these parts are passed to 4×4-bit S-box function (Anderson et al., 1998). Eight different S-box functions are used in different round functions but within a round function the same S-box function is used for all 4-bit parts. In other words, the same S-box function is applied 32 times in parallel on the block state (Anderson et al., 1998). This allows implementing the Serpent cipher in a *bit-sliced* manner (Ferguson & Schneier, 2003). The bit-slicing technique is discussed in more detail in section 3.4.

Furthermore, as Serpent is designed the bit-sliced optimization in mind, the input and the output to the encryption function do not have to be transformed to a bit-sliced form

(Anderson et al., 1998). As it happens, the non-bitsliced description of Serpent cipher defines the initial and final permutations that are identical to bit-slicing of the input and the output. Applying bit-slicing at input and output is the inverse of these permutations and as such bit-slicing can be archived by removing the additional permutations from beginning and the end of encryption function. Since the same S-box is applied 32 times in parallel in each round, the substitution phase can be implemented in bit-sliced manner by treating the block state as 32 separate 4-bit blocks. Also the permutation phase is designed so that it can be implemented efficiently for bit-sliced implementation (Anderson et al., 1998).

2.4.5 Twofish

Twofish is a Feistel cipher and was one of the five finalists in the AES contest (Ferguson & Schneier, 2003). Like AES, the block size of Twofish is 128 bits and it supports three key lengths 128, 192 and 256 bits (Schneier et al., 1998). The input plaintext to encryption functions is split to two halves as with typical Feistel ciphers. Moreover, the halves are split to two 32-bit values on which most of the operations are performed (Ferguson & Schneier, 2003).

The Twofish cipher customizes its S-boxes with the master key and therefore practical software implementations compute the look-up tables in advance of encryption or decryption (Schneier et al., 1998). The use of look-up tables as an optimization technique is discussed in more detail in the section 3.1. In the round function F, the g-function is the part where the S-box functions are executed. Since the g-function only uses a 32-bit input and the input to F-function is the 64-bit block state, the g-function is executed twice as illustrated in Figure 19. Outputs of the both g-functions are then mixed in the PHT-function and the round key is mixed with the state using 32-bit integer addition (Schneier et al., 1998).

The result of F-function is output as two 32-bit values that are mixed with the right side block state using exclusive-or operation. The 32-bit values of the right side state are additionally rotated one-bit left and right (Schneier et al., 1998). These rotations were introduced to make the cryptanalysis more difficult, but they have disadvantages. The software implementations are slowed by about 5% by the introduction of the one-bit rotations. These rotations also break the typical Feistel structure so that the decryption function cannot be constructed from the encryption function simply reversing the order of the round keys (Ferguson & Schneier, 2003).

2.5 The x86-64 architecture

The *x86-64 architecture* was introduced by Advanced Micro Devices (AMD) as a straightforward extension of previous 32-bit Intel IA32 architecture. The IA32 architecture is also known by names “i386”, “x86” and “x86-32”. We use the latter to distinct the 32-bit architecture from the 64-bit x86-64 architecture. The x86-64 architecture is also known by many other names such as “AMD64”, “Intel EM64T”, “Intel64”, “x64” and “x86-64”. We will use the name “x86-64” as it is vendor neutral and it is the original name of the architecture (Padgett & Kerner, 2007). Processors of these architectures have been highly successful and can be found in the majority of world’s computers (Bryant & O’Hallaron, 2005). The instruction set of these architectures follows Complex Instruction Set Computer (CISC) design. The benefit of CISC is that the programs can be written with a relatively small number of instructions (Irvine, 2010). The disadvantage is that

Figure 7. The arraignment of the different sized registers on the x86-64 architecture. Eight new 64-bit integer registers %r8. . . %r15 have been introduced and the old 32-bit registers have been extended to 64-bit versions. The byte registers highlighted with green background can only be accessed by using old instruction encoding of x86-32. The byte registers highlighted with red background are only available with REX encoded x86-64 instructions.

more complex instructions are hard to decode and execute. The Reduced Instruction Set Computer (RISC) design has a small number of simple instructions which allows the hardware to more quickly decode and execute instructions. This approach has been successful as high-speed engineering and graphics workstations have used RISC in past years (Irvine, 2010). Today the processors of the x86-64 architecture include the front-end that decodes complex CISC instructions to multiple RISC-like simple micro-operations and a fast RISC-like back-end for executing these micro-operations (Irvine, 2010; Fog, 2012*a*).

The x86-32 architecture defines eight 32-bit *general purpose registers*, `%eax`, `%ebx`, `%ecx`, `%edx`, `%ebp`, `%esp`, `%esi` and `%edi` (Irvine, 2010). As the x86-32 architecture is an extension of the original 16-bit Intel architecture, the lower 32-bit portion of these registers can be accessed through 16-bit registers as seen in Figure 7. Furthermore, in the x86-32 architecture the 8-bit halves of four 16-bit registers `%ax`, `%bx`, `%cx`, `%dx` can be accessed through byte registers. In the x86-32 architecture, the lower 8-bits of registers `%ebp`, `%esp`, `%esi` and `%edi` are not available as distinct byte registers (Irvine, 2010).

The main difference in the x86-64 architecture to the x86-32 architecture is that the registers are extended to 64-bits and that additional eight new registers `%r8`–`%r15` have been introduced (Bryant & O'Hallaron, 2005). These 64-bit registers and their 32-bit, 16-bit and 8-bit counterparts are presented in Figure 7. Most of the old x86-32 instructions and their machine code presentations remain the same in the x86-64 architecture. Register extension (REX) prefixes is used in the 64-bit mode to introduce new features to the old x86-32 machine codes. For example, a REX prefix can be used to change register width from 32-bits to 64-bits when accessing old x86-32 registers. REX prefix is also used when instruction needs to access the new `%r8`–`%r15` registers and it is used to access the byte registers of `%ebp`, `%esp`, `%esi` and `%edi`. When REX prefix is used the instruction using

byte registers does not access the second byte registers `%ah`, `%bh`, `%ch`, `%dh`, but instead uses `%bpl`, `%spl`, `%sil`, `%dil` register (Bryant & O'Hallaron, 2005).

In addition to the general purpose registers and instruction set operating on them, the x86 architecture has several different vector register instruction set extensions. The first such extension was 'MMX' instruction set introduced in 1997 by Intel (Bryant & O'Hallaron, 2005) and supports eight 64-bit vector registers (Intel Corp., 2013*b*). These instructions allow the data processing called *single instruction, multiple data* (SIMD). The vector registers are loaded with parallel data elements and the same operations are applied to the parallel elements in registers. Parallel processing with SIMD is very well suited for image and video processing (Bryant & O'Hallaron, 2005). Later vector instruction set include Streaming SIMD Extensions (SSE), SSE2 and SSE3, that added 128-bit vector registers for floating point and integer SIMD processing. The processors of the x86-64 architecture all support SSE2 instruction set (Fog, 2012*b*). On the x86-64 architecture, there are sixteen 128-bit vector registers also known as 'XMM' registers (Bryant & O'Hallaron, 2005). Further instruction set extensions include Supplemental Streaming SIMD Extensions 3 (SSSE3), which adds new instruction such as **`pshufb`** for shuffling byte elements within vector register (Intel Corp., 2013*b*). As we see later, this instruction is very useful for fast block cipher implementations. The AES New Instructions (AES-NI) instruction set adds hardware accelerated primitives of AES block cipher and allow AES round functions to be processed with single instruction with the block state stored in one XMM register (Gueron, 2009).

The Advanced Vector Extensions (AVX) instruction set allows the use of three-operand instructions to reduce number of register moves (Götzfried, 2012*a*). The instructions in the x86-64 architecture take mostly two operands. For example, addition of two registers uses one of the input registers for output. So the calculation takes the form $a := a + b$, where the old value of a register is lost. The three-operand format used in the AVX instruction set can avoid this since output register is explicitly stated, thus the calculation is expressed as $c := a + b$.

3 Optimization Techniques in the Literature

In this part, we discuss the optimization techniques of block ciphers used for earlier software implementations and particularly x86-64 implementations that have been presented in previous research. A great deal of research on this area, beginning from the specification of block ciphers, has been carried out. Typically, a cipher specification contains an overview of various cipher specific or general optimization techniques available to typical processors in the time the cipher was published (Aoki et al., 2001*b*; Schneier et al., 1998; Daemen & Rijmen, 2002).

Further research results on improving implementations in specific processor architectures are then published. Bernstein and Schwabe (2008) studied the performance of AES on various computer architectures, including x86-64, using a common table look-up optimization method. Matsui (2006) has studied the performance of AES and Camellia ciphers on the x86-64 architecture using the table look-up approach and the so called *bit-slicing* technique. The bit-slicing technique is interesting as the implementation mangles input and output data into a format that allows the use of hardware implementation techniques on software (Rudra et al., 2001). Bit-slicing block ciphers have received lot of

an interest and research (Matsui, 2006; Rebeiro, Selvakumar, & Devi, 2006; Matsui & Nakajima, 2007; Käsper & Schwabe, 2009). Other "slicing" techniques have also being utilized when endeavoring for the highest performance (Lloyd, 2009*b*; Matsui & Fukuda, 2005; Gladman, 2000; Aoki, Matusiewicz, Roland, Sasaki, & Schläffer, 2012). We will describe the table look-up and slicing techniques in more detail in the subsections below and also give a view to "out-of-order" execution capabilities that are typical to the majority of the x86-64 processors.

The research on optimized implementations is not just because of academic interest, but the findings have been utilized in various software projects. For example, Crypto++ library makes use of the results by Bernstein and Schwabe (2008) for an x86-64 optimized implementation of AES (Dai, 2013). The OpenSSL library (The OpenSSL Project, 2013) has incorporated the bit-sliced based AES implementation by Käsper and Schwabe (2009) and the vector permutation AES implementation by Hamburg (2009).

3.1 Table look-ups in general

Table look-ups are used as a generic optimization technique to implement parts of the cipher algorithms. Especially the S-boxes are typically realized using table look-ups (Schneier et al., 1998; Aoki et al., 2001*b*). The idea behind the table look-ups is to precalculate the S-box function for all the input values and store these precalculated function values to an array in the computer memory (Schneier et al., 1998; Aoki et al., 2001*b*). For example, with an S-box taking 8-bit input, we calculate the S-box function with the all $2^8 = 256$ input values and store the output values to the memory. The software implementation of the S-box function is then performed by using the 8-bit input as an index value for accessing the precalculated value from the memory array. The width of the elements in the array is the same as the width of S-box function output. So if the S-box function outputs 16-bits, the element size in the array will be 16-bits or two bytes. Such an 8×16-bit look-up table would then be an array with 256 two-byte elements, taking the total space of 512 bytes.

Modern block ciphers are usually designed so that one can include more building blocks of the cipher algorithm to the table look-ups for faster software implementation (Schneier et al., 1998; Daemen & Rijmen, 2000; Schneier, 1994*b*; Aoki et al., 2001*b*). Often the permutation phase in round function can be merged with the S-box phase to the same table look-up operation (Schneier et al., 1998; Aoki et al., 2001*b*; Daemen & Rijmen, 2002). Implementations can have multiple tables that perform the specific S-box function and the permutation for different bytes of the block state (Aoki et al., 2001*b*). Typically the large look-up tables take an 8-bit input index and output a 32-bit or 64-bit value (Aoki et al., 2001*b*; Schneier et al., 1998; Daemen & Rijmen, 2002).

The actual implementation details vary between different micro-architectures and finding the fastest possible way to perform table look-ups (Bernstein & Schwabe, 2008). For fast table look-ups on large 8×32 and 8×64-bit tables, the key points for the implementation performance is how the byte indexes are extracted from the larger block state and how these indexes are scaled to the array element size as fast as possible (Bernstein & Schwabe, 2008). A byte index for table look-up needs to be multiplied by the table element width before the index can be used for memory addressing. In the next subsection, we look how to perform these operations fast on the x86-64 architecture.

3.2 The use of look-up tables on the x86-64 architecture

A common practical optimization technique of block ciphers for the x86-64 processors is to use large precalculated look-up tables. Usually 4×32-bit look-up tables are used to optimize the S-box function and permutations that otherwise are slow to implement in software (Schneier et al., 1998; Daemen & Rijmen, 2000; Schneier, 1994*b*; Aoki et al., 2001*b*). For example, the Blowfish cipher is designed to use four 4×32-bit S-boxes that are generated from the master key. These S-boxes are implemented as four 4×32-bit look-up tables that require 4 KiB of memory (Schneier, 1994*b*). The AES round function contains the `SubBytes`, `ShiftRow` and `MixColumns` phases. The last round function is slightly different as it only contains the `SubBytes` and `ShiftRow` phases. The round function can be replaced by four static 4×32-bit look-up tables and the last round with another four 4×32-bit tables, requiring total memory of 8 KiB (Erdelsky, 2002).

In typical x86-64 implementations, the block state is stored in multiple 64-bit or 32-bit registers (Matsui, 2006). For example, the block state in AES implementations is usually stored in four 32-bit registers (Bernstein & Schwabe, 2008; Erdelsky, 2002). So for table look-ups taking 8-bit inputs, this one byte input has to be extracted from a larger state register. Implementing this requires shift left operations with byte masking. For example, the first byte from the variable `reg64` can be extracted with operation '`reg64 & 0xFF`' and the second byte with '`(reg64 >> 8) & 0xFF`' (Bernstein & Schwabe, 2008). On the x86-64 architecture these operations can be performed by special instruction **`movzb`** – move byte with zero extend (Matsui, 2006). The instruction reads the byte register and writes this byte value to a larger target register (Intel Corp., 2013*b*). Reading the first byte of register `%r8` to register `%r9` can be written in assembly as '**`movzbq`** `%r8b, %r9`'.

On the x86-64 architecture byte registers are aliased to parts of larger registers (Intel Corp., 2013*a*). For example, `%al` is an 8-bit register that is mapped into the first eight bits of 64-bit register `%rax`. Moreover, all small registers are aliased to larger architectural registers. The 32-bit register `%eax` is mapped into the first 32 bits of `%rax` register. Similarly the 16-bit registers are mapped into the larger-64 bit and 32-bit registers (Irvine, 2010). Figure 7 illustrates this relationship between the different size registers.

Figure 7 highlights four byte registers in green. These byte registers are small set of original registers inherited from the old x86-32 architecture that are mapped to the second 8-bits of the larger 64-bit register. These large registers are `%rax`, `%rbx`, `%rcx` and `%rdx` with corresponding second byte registers being `%ah`, `%bh`, `%ch` and `%dh`. A further restriction in the use of these second byte registers is that they can only be used with instructions that have encoding in original x86-32 architecture (Intel Corp., 2013*a*). In other words the second byte registers cannot be used in conjunction with the new architectural registers `%r8-%r15` and the old registers that have extended to 64-bits. For example, to load the second byte of the register `%rax` to register `%rsi`, one might use instruction '**`movzbq`** `%ah, %rsi`'. However, this is an invalid instruction as the target register is 64-bit wide and old x86-32 instruction encoding does not support such operation. We can however use instruction '**`movzbl`** `%ah, %esi`' which can be encoded as an x86-32 instruction (Matsui, 2006). The x86-64 architecture zeros the upper 32-bits when the lower 32-bits of the larger 64-bit register – in the previous example `%esi` – is the target register. So it is possible to use the second byte registers in the x86-64 architecture by carefully selecting the source and target registers from the available register pool.

After reading the value from the look-up table, the read value is used in some way. Usually it is mixed with other registers using addition or exclusive-or operation. Instead of using

extra instructions for these operations, on the x86-64 architecture, we can combine the memory read instruction with the arithmetic operations (Intel Corp., 2013*a*). In other words instead of using an input register operand for arithmetic instruction, a memory operand can be used instead. In this way the table look-up is used as a direct input for the arithmetic operation. For an example, consider a case where the register `%rbp` contains a memory pointer to the value that we want to add to another register. We could load the value to temporary register `%r15d` with instruction '`movl (%rbp), %r15d`' and perform addition to the target register `%edi` with instruction '`addl %r15d, %edi`'. If we do not need the value from memory for anything else, instead of using intermediate register `%r15d`, we can perform the same operation with a single instruction '`addl (%rbp), %edi`' (Bernstein & Schwabe, 2008).

The memory address calculation logic on the x86-64 architecture is powerful and versatile and simplifies the use of byte values for table indexes. The address calculation logic can take two registers – the base register and the index register – as input with an additional large constant as the memory offset and a small constant as the index register multiplier. Memory addresses can be written in x86-64 assembly as '`offset_const(base_reg, index_reg, multiplier_const)`'. If any of the first three parameters is omitted, zero is used instead. If the multiplier constant is omitted, the value 1 is used instead (Intel Corp., 2013*a*).

As 4×32-bit look-up tables have 32-bit elements, on a simpler architecture one would need to multiply the index register by four (which can be performed with shift left by two) and then add the starting location address of table to the index register (Bernstein & Schwabe, 2008). This would be written as `(byte_index * 4) + table_address`. When look-up tables are static, the table address is a constant value to fixed memory location. On the x86-64 architecture, this form of memory location can be accessed with '`table_address(,byte_index_reg,4)`'. In case look-up tables are constructed based on a block cipher master key, the table address is in form '`table_offset_in_context + context_register`' and the address calculation can be written as '`(byte_index * 4 + context_register + table_offset_in_context)`'.

Combining the facts above, we can write the operation of performing the two table look-ups from the two lower bytes from register `%rax` and XORing those values together with using four instructions:

```
movzbl %al,                          tmp_reg
movl table_sbox1(,tmp_reg,4),        dest_reg
movzbl %ah,                          tmp_reg
xorl table_sbox2(,tmp_reg,4),        dest_reg
```

3.3 Parallel execution of instructions on "out-of-order" CPUs

Most of the current x86-64 processors are capable of dispatching multiple instructions in parallel, and, therefore, being able to execute more than one instruction per CPU cycle (Stallings, 2009). These processors have multiple execution ports to where instructions can be scheduled (Fog, 2012*a*). Furthermore, many x86-64 processors have *out-of-order* scheduling, allowing the processor to pick non-dependent instructions for execution in advance from the instruction stream. Contrariwise, the *in-order* CPUs require parallelism to be explicitly exploited by exact instruction ordering (Fog, 2012*b*; Bernstein & Schwabe, 2008). So, for example, when an in-order processor is busy reading memory to one register, it can schedule the next instruction after the memory read instruction for execution if the next instruction is not dependent on the memory read. With out-of-order processors,

scheduling can skip next instructions if they depend on memory read and find instructions that are not dependent and schedule them for parallel execution (Fog, 2012*a*).

The actual implementation details of instruction scheduling and execution vary between different processor architectures. The current out-of-order x86-64 processors decode and split complex x86-64 instructions into one or multiple internal operations also called micro-operations – µops (Matsui, 2006; Fog, 2012*b*). For example, an instruction to do an arithmetic operation to a target register from memory '`addq 32(,%rbp,8), %rsi`' might be split to three µops: address calculation µop, memory read µop and 64-bit addition µop. However, some processors might have combined address calculation and memory read execution available thus the `addq` instruction being split to two µops.

Because of out-of-order scheduling, reducing cycles in fast software implementation is not just a matter of reducing number of instructions. Some implementation might have a higher instruction count, but would also be able to exploit the execution parallelism available in the processor to greater extent. Such an implementation can be able to compensate for the additional instruction count by attaining higher throughput (Bernstein & Schwabe, 2008).

3.4 "Slicing" techniques

Bit-slicing is a technique for implementing encryption algorithms without using look-up tables (Bernstein & Schwabe, 2008). This technique was first introduced by Biham (1997), who used bit-slicing to implement DES block cipher. With bit-slicing, n-bit registers are seen as SIMD vector registers with n one-bit elements. Performing one bit-wise instruction on these bit-sliced registers now equals to n one-bit parallel operations (Bhaskar, Dubey, Kumar, & Rudra, 2003). These parallel one-bit operations can be seen the equivalent of gates in the hardware implementations of encryption functions (Käsper & Schwabe, 2009) and therefore bit-sliced implementations can utilize results originally presented for hardware implementations (Bernstein & Schwabe, 2008). Some block ciphers have been designed so that some primitives of the block cipher are implemented in a bit-sliced manner in software (Käsper & Schwabe, 2009). One such block cipher is Serpent (Anderson et al., 1998). With Serpent, one can implement the S-box phases of the round function so that the 128-bit block state is bit-sliced to four 32-bit registers. The bit-wise logical instruction presentation of 4×4-bit S-boxes can then be applied to four registers, effectively processing of 32 S-box operations in parallel (Osvik, 2000).

As Serpent has been designed to utilize the bit-slicing technique, there is no requirement for Serpent to process multiple blocks in parallel to benefit from the bit-slicing (Anderson et al., 1998). However, when implementing a bit-sliced block cipher that has not being designed in such a way, number of parallel blocks needed is typically equal to bit-width of used registers (Bernstein & Schwabe, 2008). In other words, when using 64-bit registers for bit-sliced implementation, one need to process 64 blocks in parallel (Biham, 1997). For bit-sliced implementation, the presentation of input blocks need to be changed so that the first bits of the input blocks are in the first register, the second bits are in the second register and so forth. The n parallel input blocks, all n-bit wide, can be seen as $n \times n$-bit matrix, which is then transposed to get a bit-sliced presentation (Rebeiro et al., 2006). The same operation is needed at the end of the bit-sliced implementation to get the output into the correct form. This preprocessing and postprocessing of the input and output adds additional overhead to bit-sliced implementations (Bernstein & Schwabe, 2008). Figure 8 illustrates the input bit-slicing with four 4-bit registers.

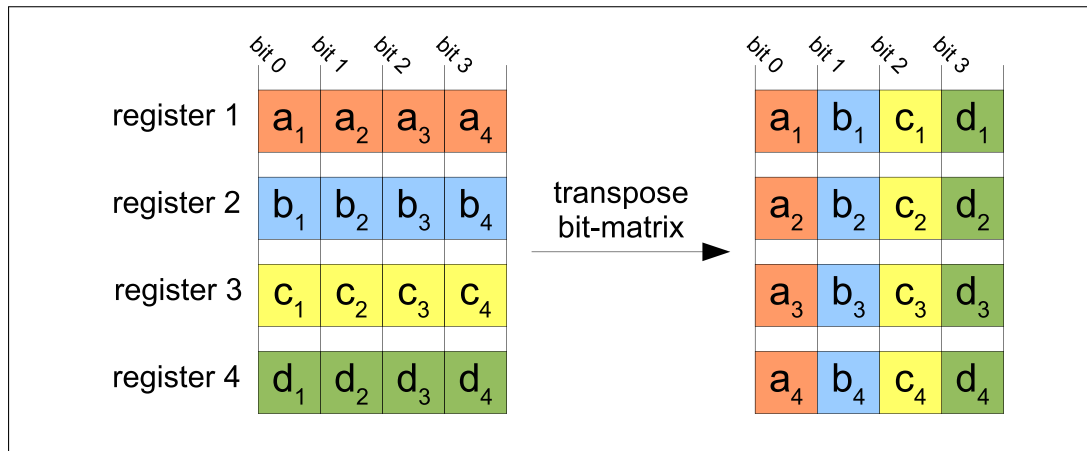


Figure 8. Bit-slicing input of four 4-bit registers.

On processors with a vector instruction set that allow shuffling the inner register elements, it is sometimes possible to take just parallel input blocks of a number equal to the bit-width of those inner elements (Käsper & Schwabe, 2009). For an example, consider XMM registers that can store sixteen byte-elements; since the SSSE3 instruction set has instructions for the shuffling of byte elements in the 128-bit vector registers, it is possible to implement bit-sliced AES using only eight parallel blocks. Whereas 128×128-bit input would require 2048 byte of data, 8×128-bit input only requires 128 bytes, making such implementation more suitable for the encryption of short messages (Käsper & Schwabe, 2009).

The slicing technique can be applied with elements wider than just one-bit. Wider slicing can yield greater performance when a fast operation for larger element is available. The *byte-slicing* approach is similar to bit-slicing, with the difference that input data is transposed in 8-bit quantities instead of one-bit (Rudra et al., 2001). Intel AES new instructions (AES-NI) (Gueron, 2009) provide instructions to perform the AES operations with the block state stored in XMM registers. The AES-NI instructions can be reused for implementing other algorithms, which make the use of the AES primitives. For example, the AES-NI instruction set have been used for implementing parts of the Grøstl algorithm (Aoki et al., 2012) and, in this thesis, we present a byte-sliced implementation of Camellia cipher that utilize AES-NI.

The most interesting primitive of AES is the `SubBytes` phase that performs the S-box function on all bytes of the block state. The AES-NI instruction set defines **`aeslastenc`** instruction that performs the last round of AES. The last round of AES consists of `AddRoundKey`, `SubBytes` and `ShiftRow` operations. The **`aeslastenc`** instruction takes two inputs, the last round key and the block state. We can feed a zeroed round key as the input to the **`aeslastenc`** instruction to disable the `AddRoundKey` operation. We can also apply the inverse of the `ShiftRow` operation to the block state before or after the **`aeslastenc`** instruction to cancel the effect of the `ShiftRow` operation. This way we can extract a plain `SubBytes` operation from the **`aeslastenc`** instruction (Gueron, 2009; Aoki et al., 2012).

The width of the “slice” can be extended beyond one byte. In the literature there is no generic term for larger slices, but Rudra et al. (2001) give the term “data-slicing” to describe all slice sizes. Here we discuss the *word-slicing* approach that is used to slice input block state in SIMD registers to larger 32-bit or 64-bit slices. This technique fits well for implementing encryption functions that do not require the use of look-up tables for fast software implementation. For example, Gladman (2000) has implemented two block

parallel 32-bit word-sliced Serpent using 64-bit registers in MMX instruction set. Matsui and Fukuda (2005) extended Gladman's method to four block parallel implementation using 128-bit registers of the SSE2 instruction set.

Recent research by Götzfried (2012*a*) combined table look-up approach with word-slicing. In this approach, word-slices are moved from vector registers to general purpose registers for table look-up phase and the results of table look-ups are moved back to the word elements of vector registers. This data transfer between different register types is needed because SIMD instruction sets typically lack the ability to do vectorized table look-ups from random access memory directly to vector registers (Neves & Aumasson, 2012). The other parts of the encryption aside of the table look-ups are accelerated by doing parallel operation on each word-slice in vector register with one instruction.

The new AVX2 instruction set (Intel Corp., 2011), that will first be available in Intel Haswell processors to be released in 2013 (Buxton, 2011), will contain such vectorized table look-up instructions (Götzfried, 2012*a*; Neves & Aumasson, 2012). As Götzfried (2012*a*) has demonstrated, the construction of word-sliced implementations is made more feasible and greatly simplified by the introduction of the new gather instructions. The vector gather instructions will allow to perform parallelized vectorized table look-ups from SIMD registers. With **`vpgather*`** instructions the more traditional table look-up based implementations can extend to exploit the parallelism available in SIMD processors (Götzfried, 2012*a*; Neves & Aumasson, 2012).

4 Implementation

In this section, we explore the design process of the implementation of the block ciphers AES, Blowfish, Camellia, Serpent and Twofish. First we give the description of generic implementation issues and then the details of the implementations for each block cipher.

Our implementations were written using x86-64 assembly language. The choice of assembly language is obvious as the goal was to produce the fastest possible implementations for specific processor architecture (Fog, 2012*b*). The assembly language is essentially a machine code, written in human readable form. In assembly language, each statement produces one machine understandable instruction (Irvine, 2010). The use of assembly allows the implementer to use special features of the underlying architecture and to perform fine-grain adjustment of the execution order of instructions (Moona, 2007). This allows the implementer to reach higher performance than that would be achievable by using a higher level programming language such as C (Fog, 2012*b*). The source-code in C-language does not represent machine code and therefore requires an extra phase, compiling, to generate machine runnable executable. The problem in reaching absolutely highest performance implementation with a higher level language is the level of abstraction presented. Higher level languages do not expose the fine-grain details of the underlying processor architecture and the compiler translating source code to machine code cannot reach the same architecture specific optimization as with the use of assembly (Irvine, 2010; Fog, 2012*b*).

We selected the GNU assembly with GCC macro preprocessing for the reasons of our familiarity with it and to gain the benefit of using C-language preprocessors macros. The native assembly syntax in GNU assembly is called 'AT&T', which is different from the Intel syntax (Fog, 2012*b*). The GNU Compiler Collection (GCC) is a commonly used

```
/* parameters
 *   ctx: pointer to cipher context
 *    st: 64-bit block state variable
 *     t: temporary 32-bit variable
 */
#define F64_lr(ctx, st, t) ({ \
  st = ror64(st, 16);           \
  t  = ctx->s[0][(st >> 8) & 0xff]; \
  t += ctx->s[1][st & 0xff];        \
  _x = rol64(st, 16);           \
  t ^= ctx->s[2][(st >> 8) & 0xff]; \
  t += ctx->s[3][st & 0xff];        \
  t; /* return value from macro */  \
})
```

Figure 9. The C-language implementation of Blowfish F-function.

C-compiler in Linux operating systems and it is licensed under the General Public License (GPL). GCC also supports various other major and minor operating systems (GNU Project, 2013). Using the GNU assembler lets our implementations be easily integrable to software projects containing the support for GCC. The GNU Compiler Collection has support for assembly language with C-language preprocessing, allowing the use of macros to implement the high level program structures (Moona, 2007). By using macros we can define the block cipher structures such as a round function as a macro and then use multiple macro invocations to duplicate the consecutive round functions without repeating the same function in the assembly language.

4.1 Generic approach to block cipher implementation construction

Our approach to construct the assembly implementations started first at generating implementation using C-language with some aspects of the processor architecture emulated. For example, the x86-64 architecture has 64-bit registers available, but many block ciphers contain a more natural alignment to 32-bit words. Thus new C-language implementations, storing the block cipher state to 64-bit words, were constructed to test the concept of storing block state to fewer registers. Figure 9 shows the Blowfish F-function macro in C-language using a single 64-bit variable `st` for storing the block state.

Verifying the correctness of block cipher implementations is relatively simple, thanks to the properties of pseudo-random functions. Even the smallest execution errors, such as flip of one single bit, propagate through avalanche effect to a completely different – invalid – output. Implementations were tested against predefined test vectors and against reference implementation using large bulk encryption and decryption.

The next phase was to construct a working assembly implementation, with focus on the correctness. The upper level structure of block cipher such as the round function macros were copied from the used reference implementation. Then each assembly macro was implemented and integrated to the reference block cipher implementation using function calls. This hybrid realization with some element replaced with an assembly part allowed us to become convinced that the assembly part was correctly carried out. In case the assembly part was not correctly implemented, hybrid implementation did not pass the test vector and large correctness tests. Thus the assembly part was investigated further and the process iterated until the assembly part was correctly executed.

The next phase was the optimization of implementation for the processors used for the measurements in this study. Modifications to the execution details and the ordering of the assembly instructions were introduced. A custom framework using **`cpuid`** and **`rdtsc`**

instructions to measure the number of execution cycles as illustrated by Matsui and Fukuda (2005) was used to get quick performance readings after each modification. Although these measurements were not rigorous enough to be presented as results, measurements provided clear simple number comparable between each optimization cycle. These quick construction time measurements were also performed on multiple different processors. This was done to avoid optimizing implementations too specifically for one processor model in expense of performance on the other processors. So, for example, a gain of 2% in the speed on one processor model was not regarded acceptable if the performance on the other processors degraded by 5%. The optimization phase was iterated until a reasonable balance in implementation performance between the different processors was reached.

4.2 Blowfish

In this subsection, we first present two different table-lookup based implementations of Blowfish. Then we look at an implementation utilizing table look-ups in combination of word-slicing with AVX instruction set by Götzfried (2012*a*) and provide an improved version of this word-sliced realization. For references to the source code of the new Blowfish implementations presented here, see appendix A.

Typically Feistel-network cipher implementations split the two halves of block state to different registers, in the case of Blowfish to two 32-bit registers (Schneier, 1994*b*, 1994*a*). Our basic implementation takes a slightly different approach in the storing intermediate 64-bit block state. These two registers are then said to contain the left and right states of block. The F-function takes the left register as the input and the output of the F-function is XORed to the right side register. At the end of the round the left and right registers are swapped (Schneier, 1994*b*).

In our first realization, we store the block state to a single 64-bit register. The left block state is contained in the lower 32-bit part of 64-bit register and it is used as the input for the F-function. By using the **`movzb`** instructions, we can extract the first and the second bytes of the block state for table look-ups directly. Then the block state is rotated bitwise right by 16-bits using 64-bit rotation instruction **`rorq`** and the next two bytes are fetched with two **`movzb`** instructions. After the second set of **`movzb`** instructions, the block state is rotated 16-bits right the second time to have the both 32-bit sides of the original block state swap sides in the 64-bit register. Now the lower 32-bits of the block state register contain the previous right side of the block state and the 32-bit result of the F-function can be added to the block state. Figure 10 shows the structure of a typical Blowfish round implementation and our round structure. Additionally the adding of two 32-bit round keys has been merged to single 64-bit operation, executed every two rounds.

Our assembly implementation of the F-function has 11 instructions (see Figure 11) and one round has 11.5 instructions, as the key addition adds one instruction per two rounds. However this implementation is not faster than typical 32-bit implementation, because the number of instructions is not significantly lower. One way to further optimize the performance would be to find a way to do operations more parallel, allowing the out-of-order execution to better exploit the execution ports available in the processor.

Blowfish has such a structure that instruction level parallelism is not attained using conventional optimization strategies as all instructions and operations are tightly dependent on the previous operations. Matsui (2006) noted about the same issue when implementing Camellia cipher using x86-64 assembly. The solution to expose more parallelism was to process two distinct blocks in parallel. This method requires that there are enough registers

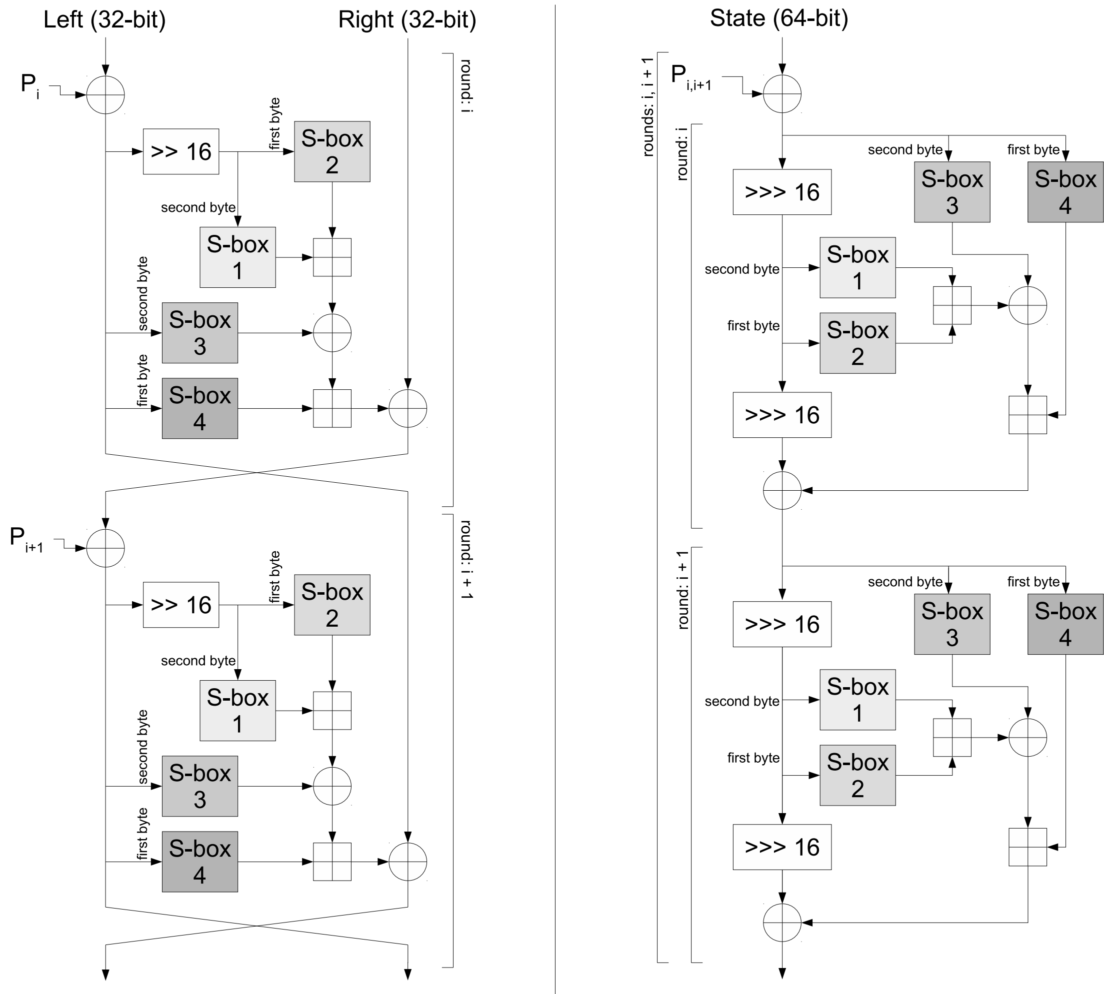


Figure 10. The structure of typical Blowfish implementation is shown on the left, with the structure of our implementation on the right. Order of table lookups is reverse on the x86-64 architecture than that of Blowfish specification. This is because Blowfish is defined for big-endian systems, whereas x86-64 is little-endian (Intel Corp., 2013*a*; Schneier, 1994*b*). The symbol $\oplus$ means bitwise exclusive or – XOR – operation, $\boxplus$ means 32-bit integer addition. The $\ggg$ operation means bitwise right rotation while $\gg$ is for bitwise right shift.

```
/* parameters
 *  x: 64-bit block state register
 */
#define F(x) \
  movzbl x ## bh,       RT1d;   /* read second byte of left */ \
  movzbl x ## bl,       RT3d;   /* read first byte of left */ \
  rorq $16,             x;      /* rotate x right by two bytes */ \
  movzbl x ## bh,       RT0d;   /* read fourth byte of left */ \
  movzbl x ## bl,       RT2d;   /* read third byte of left */ \
  rorq $16,             x;      /* rotate x right by two bytes */ \
  /* now sides of block state in x have been swapped */ \
  movl s0(CTX,RT0,4),   RT0d;   /* tables look-ups and */ \
  addl s1(CTX,RT2,4),   RT0d;   /* F-function calculations. */ \
  xorl s2(CTX,RT1,4),   RT0d; \
  addl s3(CTX,RT3,4),   RT0d; \
  xorq RT0,             x;      /* Adding F-function result */
                                /* to right side */
```

Figure 11. Assembly implementation of Blowfish F-function.

```
/* parameters
 *   n: round number, multiple of 2
 */
#define two_rounds_enc4(n) \
  /* Add round keys P(n) and P(n + 1) to RA, RB, RC and RD. */
  add_roundkey_enc4(n); \
  \
  /* round: n */ \
  F(RA); \
  F(RB); \
  F(RC); \
  F(RD); \
  \
  /* round: n + 1 */ \
  F(RA); \
  F(RB); \
  F(RC); \
  F(RD);
```

Figure 12. The round macro of 4-way Blowfish assembly implementation.

available to store multiple blocks and that a non-feedback block cipher mode is used to attain input blocks that do not depend on previous block cipher operations. (Matsui, 2006)

Since our implementation stores the block state in single 64-bit register, we can store block states in the four 64-bit registers that have the both low byte registers available for use with the **movzb** instruction. So to better exploit more parallelism we constructed implementation that takes four input blocks. The code for each separate block instruction stream was interleaved at the round level, such that the F-function of block A was followed by the F-function of B, C and D (see Figure 12). The out-of-order execution, as explained earlier, schedule the non-depended operations from the different instructions streams to the parallel execution ports. This allows our four block parallel – 4-way – implementation to gain much greater performance. As seen in the results section 6, on all processors used for measurements, our 4-way implementation was over two times faster than the reference implementation and our one block non-parallel – one-way – implementation.

Götzfried (2012*a*) implemented Blowfish in a 16-way parallel manner, using word-slicing combined with table look-ups. In this implementation (Götzfried, 2012*b*), 16 input blocks are word-sliced in eight 128-bit XMM registers using the AVX instruction set. Words are extracted from the XMM registers to regular integer registers for the table look-ups. The table indexes are then extracted from these integer registers using move byte instruction **movb** and more bytes in the integer register are exposed by shifting the value to left in multiples of 8-bits. The calculations related to table look-ups are combined with the memory read as is possible on the x86-64 architecture. The round key addition benefits from the vectorized block state, because it is able to do round addition on four parallel block states in one XMM register with one instruction.

We improved this implementation by fixing few problems that were found to make the performance of this implementation sub-optimal. First, the use of **movb** to extract bytes has bad consequences for performance because this instruction, unlike the **movzb** instruction, does not completely overwrite the target register. The byte registers on x86-64 are aliased to lower bytes of larger architectural 64-bit registers. So the **movb** instruction moves the byte value from the source register to the lower byte slot of the target 64-bit register, keeping upper the 7 bytes of the target register unchanged. Now if this is unintentional like with Götzfried's implementation, we have false dependencies between **movb** instructions operating on the same target register and the 'out-of-order' scheduler in the processor cannot parallelize the operations performed on this register (Fog, 2012*b*).

In our improved implementation, the **movb** instructions have been replaced with the **movzbl** instructions that zero-extend the source byte to the target register. In other words, the source byte is copied to the low 8-bits of architectural register and the upper bits of register are set to zero. As the whole target register is written, false dependencies are avoided and code is better suited for the 'out-of-order' reordering by the processor (Fog, 2012*a*).

We also removed byte-swapping operation in Götzfried's implementation which was used for the reordering of bytes for table look-ups. This reordering can be achieved at the table look-up phase without additional instructions. This can be done by loading bytes to temporary registers and using these registers in different order for table look-ups.

Further fine-grained ordering and interleaving was utilized especially at the phase where words are extracted from and inserted to XMM registers. We moved the extraction of words from XMM register of next round block state before insertion of current round block state to XMM register. This way processor can proceed to do next table look-ups faster without possibly blocking on previous block state.

4.3 AES

Fast AES implementations have been target of many studies. Bernstein and Schwabe (2008) and Matsui (2006) study fast execution of the AES cipher on the x86-64 architecture using conventional table-lookup approach. Matsui (2006) also presents and studies a bit-sliced implementation of AES and the bit-slicing approach to AES implementation is perfected in later research by Matsui and Nakajima (2007) and Käsper and Schwabe (2009). The latest Intel and AMD processors of today also contain hardware acceleration of AES available in the form of instruction set extension 'AES-NI' (Gueron, 2009), which gives order of magnitude greater throughput than software implementations with the fraction of program size (Akdemir et al., 2010).

One might ask if there is any room for improvement. Compared to Feistel-network block ciphers implemented in this research, AES entail greater parallelism in its round structure making it easy to exploit parallel execution available in out-of-order CPUs (Daemen & Rijmen, 2000) without resorting to parallel blocks approach. For our research, we experiment with a two block parallel table-lookup implementation of AES to see if this approach would gain better throughput on older processors or not. We also improve the bit-sliced realization by Käsper and Schwabe (2009) by converting their implementation from the SSSE3 instruction set to the AVX instruction set. For references to the source code of the new AES implementations presented here, see appendix A.

The table-lookup based AES implementations on the x86-64 architecture typically store the block state into four 32-bit registers and construct new block state to another four register set through using old state as input for 8×32-bit table-lookups (Erdelsky, 2002). In our implementation the block state is stored in two 64-bit registers from which bytes are extracted for 8×32-bit table-lookups. In the round function, intermediate block state is gathered to four 32-bit registers and at the end of the round these four 32-bit registers are merged to two 64-bit registers, now containing the new block state (see Figure 13). For a one block non-parallel implementation, this approach adds extra instructions for merging the 32-bit registers to 64-bit registers and introduces more register dependencies to the instruction stream, yielding lower throughput.

Sacrificing throughput by using 64-bit registers to store the block state reduces register pressure and allows us to construct a two block parallel implementation. To be able to

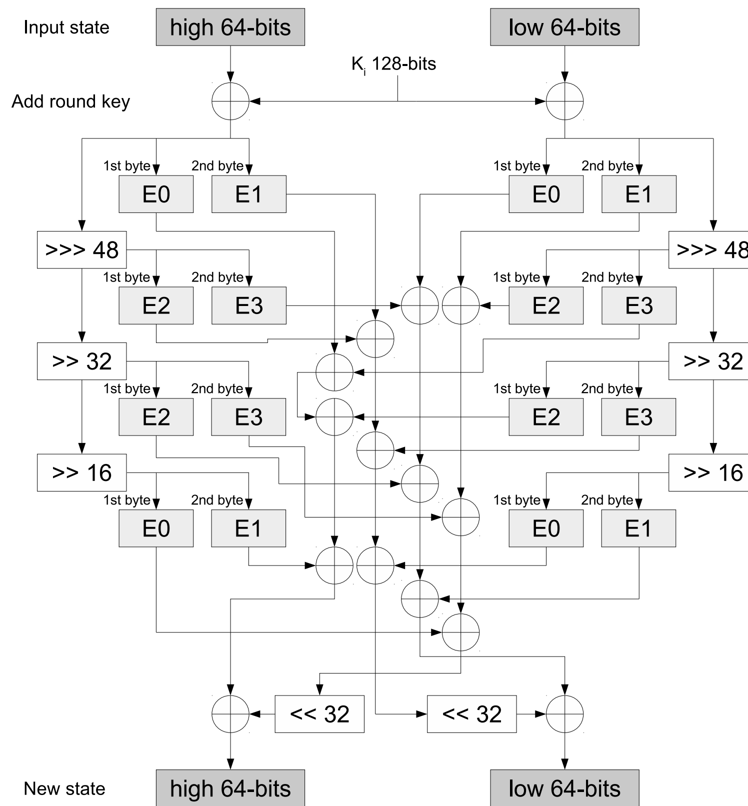


Figure 13. The structure of AES round function for one block in two block parallel implementation. The symbol $\oplus$ means bitwise exclusive or – XOR – operation and $\ggg$ operation means bitwise right rotation.

extract the second byte values using **`movzbl`** instructions, the first block state is stored in registers `%rax` and `%rbx` whereas the second block state is stored in `%rcx` and `%rdx`. In addition two sets of four 32-bit registers are needed to hold the intermediate block state, totaling the number of needed registers to store the block states to twelve. In addition one register is needed to store the pointer to the round keys and another for the stack pointer, leaving two registers free for use as temporary registers.

As an additional counter mode specific optimization, we utilize counter-mode caching (Bernstein & Schwabe, 2008). As later explained in section 5, the measurement framework used in our study uses the counter-mode for testing the performance of block cipher implementations (Bernstein & Lange, 2013). In counter-mode, the input block to block cipher is increased by one for the next block. This means that the upper 15 bytes of input block stay same for 256 blocks (Bernstein & Schwabe, 2008). As input blocks do not change much, we can cache the most of the processing for the first and the second rounds. So we use the precomputed cache values and a byte counter as input to our encryption function. For every 256 blocks when the byte counter overflows, the counter cache is recomputed with new initialization vector.

Our implementation uses 8 KiB precomputed look-up tables. Four 8×32-bit tables total size of 4 KiB are used to implement the round function containing `SubBytes`, `ShiftRow` and `MixColumns` phases. Another four 8×32-bit tables are used to implement the last round with `SubBytes` and `ShiftRow` phases. Bernstein and Schwabe (2008) describe a way to combine these look-up tables so that they require only 3 KiB, by using unaligned memory reads. With our two-way implementation using of 3 KiB tables was found to be slightly slower than with 8 KiB tables and therefore the larger tables were chosen for our implementation.

Our bit-sliced implementation is based on work by Käsper and Schwabe (2009). In their realization eight parallel blocks are bit-sliced so that the bit-slicing is done at byte element level. Previous bit-slice implementation by Matsui (2006) did the bit-slicing on full 128-bit block level and required 128 parallel blocks. Using only eight parallel blocks is practical because the most computationally heavy phase of AES round, `SubBytes` is done at byte level in parallel to the all 16 bytes of the block state. The `ShiftRow` and `MixColumns` that mix the values in bytes, use the **pshufb** instruction for moving the bytes within XMM registers. Käsper and Schwabe (2009) report their execution reaching the speed of 6.92 cycles per byte for long messages in counter mode with Intel i7 Nehalem processor.

In our realization, we used the implementation introduced by Käsper and Schwabe (2009) and in a straightforward way converted it from the SSSE3 instruction set to the AVX instruction set. The AVX instruction set utilizes three-operands for instructions instead of the two-operand form used in the SSSE3 instruction set (Intel Corp., 2013*a*). The two-operand instruction requires the use of extra register move instruction when both source registers need to be preserved for further use. With three-operand AVX instructions we can avoid these extra register copies, thus speed up the execution slightly. Our AVX implementation of `SubBytes` avoids move instructions and requires 132 instructions, which is 31 instructions or 19.0% less than in the SSSE3 implementation. In total our AES round implementation requires 194 instructions, being 28 instruction or 12.6% less than in the SSSE3 implementation (Käsper & Schwabe, 2009). Our `ShiftRows` and `MixColumns` use extra memory copies to load the constant vectors into temporary registers. This is to avoid combined memory loads with **pshufb** instruction, which are translated to separate micro-operations by the processor thus requiring more operations.

4.4 Camellia

For the Camellia block cipher we present implementations using two distinct implementation strategies. Our first implementation is a realization following the research by Matsui (2006). The work by Matsui (2006) is a two block parallel – two-way – look-up table based approach and has the best performance reported for a look-up table based Camellia implementations on the x86-64 architecture, 10.9 cycles per byte on *Athlon 64* processor. Matsui (2006) has not released the full implementation, so our attempt is to reproduce the results of their research. In our table look-up execution we spent time on perfecting the performance by careful ordering of the instructions and the use of intermediate variables to break dependency chains in the round function. Our second implementation strategy takes the byte-sliced approach and utilizes the AES-NI instruction set for an accelerated Camellia S-box execution. For references to the source code of the new Camellia implementations presented here, see appendix A.

Our table look-up implementation uses eight 8×64-bit tables for realizing the F-function of Camellia and these look-up tables require 16 KiB of space. The input 128-bit plaintext block is split to two 64-bit registers which represent the left and right sides of block state in the Feistel structure of the cipher. The implementation of the round function as presented by Matsui (2006) is quite straightforward (see Figure 14). Each byte of the left side 64-bit register is extracted with the use of the first and second byte **movzb** move instructions. The extracted bytes are then used as 8-bit index values for the table look-ups. The left side 64-bit register is rotated to the right by 16 bits after each **movzb** pair to get the access to further bytes. After extracting all eight bytes, the left side 64-bit register has been rotated to the right by total 64-bits and therefore is restored to the original state. The 64-bit results of table look-ups then XORed to the right side 64-bit register. Since the

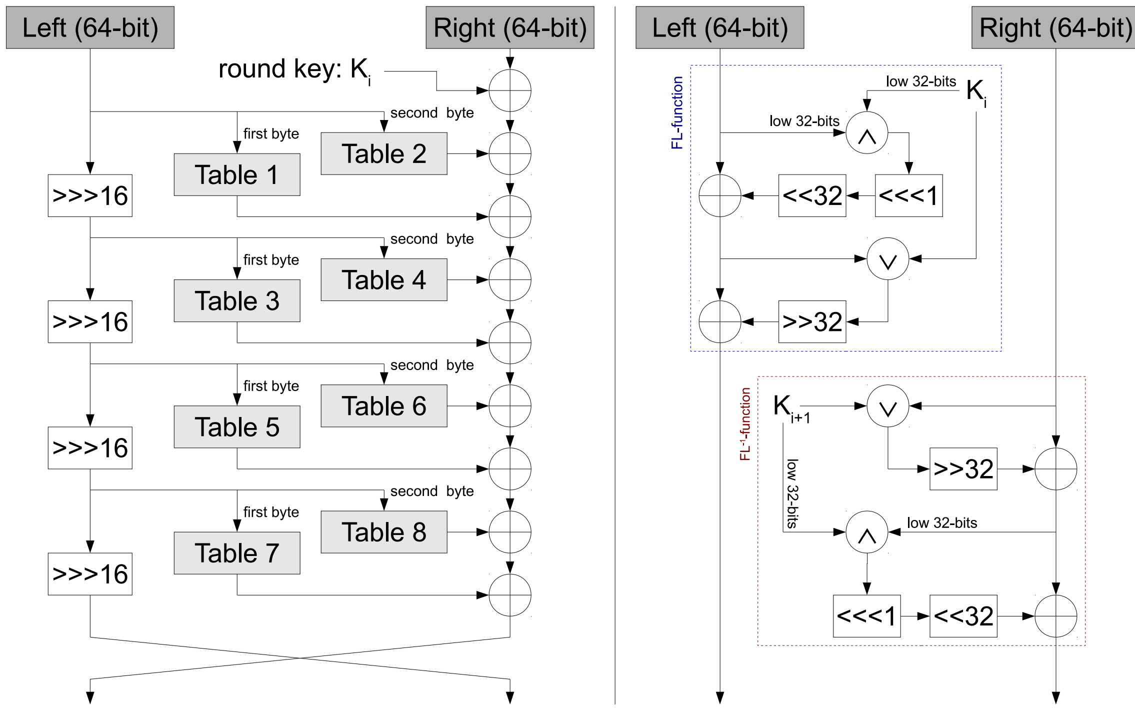


Figure 14. On the left, the structure of Camellia round function for one block in two block parallel implementation. On the right, the structure of the non-Feistel round. The symbol $\oplus$ means bitwise XOR, $\vee$ bitwise OR and $\wedge$ bitwise AND operation. The $\ggg$ operation means bitwise right rotation while $\gg$ is for bitwise right shift. Operations $\ll$ and $\lll$ mean left shift and rotation.

exclusive-or operation is commutative and it is the only operation used, we can avoid using intermediate registers and XOR the look-up table values directly to the right side register.

The round key addition is moved to other side of F-function as an optimization. Regular Camellia F-function is defined as $F(x) = P(S(x \oplus k))$, where x is the left side block state and k the round key. The S-function applies Camellia s-boxes on the input block state. The P-function mixes the bytes of the block state with each other using the exclusive-or operation. It is possible to process the round keys so that addition can be done after S and P-functions (Aoki et al., 2001*b*). The F-function is then defined as $F(x) = P(S(x)) \oplus k'$, where k' is the processed round key.

In the non-Feistel round of Camellia, the FL-function is applied to the other 64-bit half of block state and the inverse of the FL-function to the other half. The FL-function (and its inverse) in our implementation is identical to the one presented by Matsui (2006). The non-Feistel round is visualized in Figure 14.

Figure 15 shows the round function macro used in our two block parallel implementation. The processing on the two blocks is minimally interleaved and the out-of-order scheduling in the processor is let to find the instruction level parallelism in the code. For greater throughput, our realization uses two target registers for the first block to break register dependencies. The '`RT2`' register is loaded with the round key at the start of round in macro '`roundsm2_ab_to_cd`' and it is the other target register for first block table look-ups, while the other is the left side state register '`cd##0`'. The second block table look-ups are added directly to the left side state register '`cd##1`'.

Our second implementation strategy involves Intel's AVX (Intel Corp., 2011) and AES-NI (Gueron, 2009) instruction sets. The technique utilizes the byte-slice approach and reuses AES `SubBytes`-function extracted from the AES-NI instruction set for executing the Camellia s-boxes s_1, s_2, s_3 and s_4.

```
#define xor2ror16(T0, T1, tmp1, tmp2, ab, dst) \
        movzbl ab##bl,                  tmp2##d; \
        movzbl ab##bh,                  tmp1##d; \
        rorq $16,                       ab; \
        xorq T0(, tmp2, 8),             dst; \
        xorq T1(, tmp1, 8),             dst;

#define roundsm2(ab, cd) \
        xor2ror16(sp00444404, sp03303033, RT0, RT1, ab##0, cd##0); \
        xor2ror16(sp22000222, sp10011110, RT0, RT1, ab##0, RT2); \
        xor2ror16(sp11101110, sp44044404, RT0, RT1, ab##0, cd##0); \
        xor2ror16(sp30333033, sp02220222, RT0, RT1, ab##0, RT2); \
        \
        xor2ror16(sp00444404, sp03303033, RT0, RT1, ab##1, cd##1); \
        xorq RT2,                                          cd##0; \
        xor2ror16(sp22000222, sp10011110, RT0, RT1, ab##1, cd##1); \
        xor2ror16(sp11101110, sp44044404, RT0, RT1, ab##1, cd##1); \
        xor2ror16(sp30333033, sp02220222, RT0, RT1, ab##1, cd##1);

#define roundsm2_ab_to_cd(subkey) \
        movq (key_table + ((subkey) * 2) * 4)(CTX),      RT2; \
        xorq RT2,                                        RCD1; \
        roundsm2(RAB, RCD);
```

Figure 15. Round macros of our two-way Camellia assembly implementation.

The Camellia s-boxes and the AES `SubBytes` use very similar basic components for their construction. The AES `SubBytes` uses multiplicative inversion on $GF(2^8)$ field and the Camellia s_1-function multiplicative inversion on $GF((2^4)^2)$ composite field (Satoh & Morioka, 2003). In their research Satoh and Morioka (2003) described a unified hardware architecture for Camellia and AES, which uses inversion in $GF((2^4)^2)$ for carrying out the AES `SubBytes`. Galois fields of equal size are isomorphic and one can define isomorphism functions in the form of $GF(2)$ matrix multiplication for mapping values between $GF((2^4)^2)$ and $GF(2^8)$ (Satoh & Morioka, 2003). In our byte-sliced implementation, we use this result in the opposite way. We use isomorphism functions to expose the multiplicative inverse on $GF(2^8)$ field from `SubBytes` as the multiplicative inverse on $GF((2^4)^2)$ composite field for the Camellia s-boxes.

First we look at how to isolate the 'multiplicative inverse on $GF(2^8)$' from `SubBytes`. The `SubBytes` (1) function is constructed from the combination of inverse on $GF(2^8)$ followed by an affine transform in $GF(2)$ (Satoh & Morioka, 2003; Daemen & Rijmen, 2002). Equation (3) shows the AES affine transformation.

$$\texttt{SubBytes}(x) = A(inv(x)) \tag{1}$$

$$inv(x) = x^{-1};\ x \in GF(2^8) \tag{2}$$

$$A: \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} := \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} \oplus \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \tag{3}$$

The multiplicative inverse on $GF(2^8)$ operation (4) can be exposed by applying the inverse of AES affine transform (5) after `SubBytes`.

$$inv(x) = A^{-1}(A(inv(x))) = A^{-1}(\texttt{SubBytes}(x)) \tag{4}$$

$$A^{-1}: \begin{pmatrix} y_0\\ y_1\\ y_2\\ y_3\\ y_4\\ y_5\\ y_6\\ y_7 \end{pmatrix} := \begin{pmatrix} 0&0&1&0&0&1&0&1\\ 1&0&0&1&0&0&1&0\\ 0&1&0&0&1&0&0&1\\ 1&0&1&0&0&1&0&0\\ 0&1&0&1&0&0&1&0\\ 0&0&1&0&1&0&0&1\\ 1&0&0&1&0&1&0&0\\ 0&1&0&0&1&0&1&0 \end{pmatrix} \begin{pmatrix} x_0\\ x_1\\ x_2\\ x_3\\ x_4\\ x_5\\ x_6\\ x_7 \end{pmatrix} \oplus \begin{pmatrix} 1\\ 0\\ 1\\ 0\\ 0\\ 0\\ 0\\ 0 \end{pmatrix} \tag{5}$$

Fast execution of matrix multiplication in $GF(2)$ using the vector register instruction set extensions on the x86-64 architecture can be achieved with the **pshufb** instruction. The **pshufb** and **vpshufb** instructions can be used to implement the 4-bit to 8-bit look-ups although instruction is mainly intended for shuffling bytes within a vector register (Fog, 2012*a*; Hamburg, 2009).

The affine transform matrix in equation (5) can be split in two halves each taking four of eight input bits, $x_0 - x_7$. The results from the split matrices in equations (6, 7) are then merged with bitwise addition completing the inverse affine transformation (8). For our implementation, the constant bit-vector used for addition was combined with the x_0, x_1, x_2, x_3 matrix in equation (6).

$$A^{-1}_{low}: \begin{pmatrix} y_0\\ y_1\\ y_2\\ y_3\\ y_4\\ y_5\\ y_6\\ y_7 \end{pmatrix} := \begin{pmatrix} 0&0&1&0\\ 1&0&0&1\\ 0&1&0&0\\ 1&0&1&0\\ 0&1&0&1\\ 0&0&1&0\\ 1&0&0&1\\ 0&1&0&0 \end{pmatrix} \begin{pmatrix} x_0\\ x_1\\ x_2\\ x_3 \end{pmatrix} \oplus \begin{pmatrix} 1\\ 0\\ 1\\ 0\\ 0\\ 0\\ 0\\ 0 \end{pmatrix} \tag{6}$$

$$A^{-1}_{high}: \begin{pmatrix} y_0\\ y_1\\ y_2\\ y_3\\ y_4\\ y_5\\ y_6\\ y_7 \end{pmatrix} := \begin{pmatrix} 0&1&0&1\\ 0&0&1&0\\ 1&0&0&1\\ 0&1&0&0\\ 0&0&1&0\\ 1&0&0&1\\ 0&1&0&0\\ 1&0&1&0 \end{pmatrix} \begin{pmatrix} x_4\\ x_5\\ x_6\\ x_7 \end{pmatrix} \tag{7}$$

$$A^{-1}(x) = A^{-1}_{low}(x_0, x_1, x_2, x_3) \oplus A^{-1}_{high}(x_4, x_5, x_6, x_7) \tag{8}$$

Two 4×8-bit look-up tables for the **pshufb** instruction can be construction from equations (6, 7). The 128-bit look-up tables for the inverse AES affine transform are shown in Table 1. Figure 16 shows the implementation of the inverse affine transform using the AVX instruction set. The look-up is performed for all the 16 bytes of the input register in parallel. By performing this set of instructions after the **vaeslastenc** instruction, we have isolated the multiplicative inverse on $GF(2^8)$ for the 16 parallel bytes.

The Camellia s-box function s_1 is defined as in equation (9), where x is the input byte, $g(\cdot)$ presented by equation (10) is the function performing the multiplicative inverse in $GF((2^4)^2)$ field.

$$s_1(x) = h(g(f(\mathtt{0xc5} \oplus x))) \oplus \mathtt{0x6e} \tag{9}$$

```
/* load 4-bit mask */
vmovdqa 0x0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f, %xmm1;
/* load lower look-up table */
vmovdqa 0xa0ea347e89c31d57f2b8662cdb914f05, %xmm2;
/* load higher look-up table */
vmovdqa 0x5afe13b7c86c81257fdb3692ed49a400, %xmm3;

/* extract lower 4-bits of input (%xmm0) */
vpand   %xmm0, %xmm1,   %xmm7;
/* extract higher 4-bits of input */
vpandn  %xmm0, %xmm1,   %xmm0;
vpsrld  $4, %xmm0,      %xmm0;

/* look-up for lower part */
vpshufb %xmm7, %xmm2,   %xmm7;
/* look-up for higher part */
vpshufb %xmm0, %xmm3,   %xmm0;
/* combine results to output (%xmm0) */
vpxor   %xmm7, %xmm0,   %xmm0;
```

Figure 16. Implementation of AES inverse affine transform, equation (5), using AVX instruction set

$$g(x) = x^{-1};\ x \in GF((2^4)^2) \tag{10}$$

The $f(\cdot)$ and $h(\cdot)$ functions are linear transformations that can be presented in a bit-matrix format (Aoki et al., 2001*b*; Satoh & Morioka, 2003). Equation (11) shows the bit-matrix for $f(\cdot)$ combined with addition with the value `0xc5` and equation (12) presents matrix for $h(\cdot)$ and the addition with `0x6e`. As these bit-matrices are linear functions, it is possible to pass the addition values through the matrix part of the affine function (Satoh & Morioka, 2003). Thus our presentation of '$f(x \oplus \texttt{0xc5})$' takes the form '$f(x \oplus \texttt{0xc5}) = f(x) \oplus \texttt{0x75}$'.

$$F: \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} := \begin{pmatrix} 0&1&0&0&0&1&0&0 \\ 1&0&0&0&0&0&1&0 \\ 0&0&1&0&1&0&0&1 \\ 0&0&1&0&0&0&0&1 \\ 0&0&0&1&0&0&1&0 \\ 0&1&0&0&1&0&0&0 \\ 1&0&0&0&0&0&0&1 \\ 0&0&0&1&0&1&0&0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} \tag{11}$$

$$H: \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} := \begin{pmatrix} 0&1&0&0&1&1&0&0 \\ 0&1&0&0&0&1&0&0 \\ 0&0&0&1&0&0&1&0 \\ 0&1&0&0&0&0&0&1 \\ 0&0&1&0&0&0&1&0 \\ 1&0&0&0&0&0&0&1 \\ 1&0&0&0&1&0&0&0 \\ 0&0&1&0&0&1&0&0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \end{pmatrix} \tag{12}$$

Rest of the s-box functions s_2, s_3 and s_4 are defined as slight modifications of the s_1-function (Aoki et al., 2001*b*). The s_2-function (13) rotates the output byte one-bit to left and the s_3-function (14) to the right, whereas the s_4 (15) rotates the input byte to the left by one-bit.

$$s_2(x) = s_1(x) \lll 1 \tag{13}$$

$$s_3(x) = s_1(x) \ggg 1 \tag{14}$$

$$s_4(x) = s_1(x \lll 1) \tag{15}$$

In their paper, Satoh and Morioka (2003) presented isomorphism functions δ and δ^{-1} for mapping the values from $GF(2^8)$ to $GF((2^4)^2)$ and from $GF((2^4)^2)$ to $GF(2^8)$. Equation (16) shows the δ-function for transforming the values from AES $GF(2^8)$ to Camellia $GF((2^4)^2)$ field. The δ^{-1}-function in equation (17) is used to transform the values from $GF((2^4)^2)$ to $GF(2^8)$.

$$\delta : \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} := \begin{pmatrix} 1&0&1&0&0&0&0&1 \\ 1&0&1&0&1&1&0&0 \\ 1&1&0&1&0&0&1&0 \\ 0&1&1&1&0&0&0&0 \\ 1&1&0&0&0&1&1&0 \\ 0&1&0&1&0&0&1&0 \\ 0&0&0&0&1&0&1&0 \\ 1&1&0&1&1&1&0&1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} \tag{16}$$

$$\delta^{-1} : \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} := \begin{pmatrix} 0&0&1&0&0&1&0&0 \\ 1&1&1&0&1&1&1&0 \\ 1&0&1&0&0&1&0&0 \\ 0&1&0&1&1&0&1&0 \\ 1&0&1&1&0&0&1&0 \\ 0&1&1&1&0&0&1&0 \\ 1&0&1&1&0&0&0&0 \\ 0&1&0&1&0&0&0&1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} \tag{17}$$

Now we can define the Camellia s_1-function to use the multiplicative inverse in $GF(2^8)$ using the isomorphism functions (16, 17). Furthermore, we can replace the multiplicative inverse in $GF(2^8)$ with combined $\texttt{SubBytes} \times A^{-1}$-function as in equation (4). As a result we can present the s_1-function as:

$$s_1(x) = h(g(f(\texttt{0xc5} \oplus x))) \oplus \texttt{0x6e} = H(g(F(x)) = H(\delta(inv(\delta^{-1}(F(x)))))$$

$$s_1(x) = H(\delta(A^{-1}(\texttt{SubBytes}(\delta^{-1}(F(x)))))) \tag{18}$$

However, it should be noted that the bit-endianness of Camellia and AES differ and therefore function in equation (19) is needed for swapping the bit-order when moving between the fields. We also found out that the basis change matrices presented by Satoh and Morioka (2003) were in the bit-endianness of Camellia.

$$S = S^{-1} : \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} := \begin{pmatrix} 0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 1&0&0&0&0&0&0&0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} \tag{19}$$

By refining the function in equation (18) with the bit-swapping function (19), we get an equation of the s_1-function that can be used as the basis for our implementation.

$$s_1(x) = S^{-1}(H(\delta(S(A^{-1}(\texttt{SubBytes}(S^{-1}(\delta^{-1}(F(S(x)))))))))) \tag{20}$$

From the equation (20) we define a pre-filter function θ_1 and a post-filter function ψ_1 as in equations (21, 22). As these functions contain only linear transformations in the form of bit-matrices and additions in $GF(2)$, we can combine these operations into a single bit-matrix operation and addition for each filter function.

$$\theta_1(x) = S^{-1}(\delta^{-1}(F(S(x)))) \tag{21}$$

$$\psi_1(x) = S^{-1}(H(\delta(S(A^{-1}(x))))) \tag{22}$$

$$\theta_1 : \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} := \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} \oplus \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \tag{23}$$

$$\psi_1 : \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} := \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} \tag{24}$$

Now we can define the Camellia s_1-function using θ_1, ψ_1 and `SubBytes` functions. Also, we represent the other three Camellia s-box functions in a new way.

$$s_1(x) = \psi_1(\texttt{SubBytes}(\theta_1(x))) \tag{25}$$

$$s_2(x) = s_1(x) \lll 1 \tag{26}$$

$$s_3(x) = s_1(x) \ggg 1 \tag{27}$$

$$s_4(x) = s_1(x \lll 1) \tag{28}$$

For the s_2 and s_3 functions we can define new post-filter bit-matrix multiplication functions ψ_2 and ψ_3 to incorporate the rotation operation. For same purpose, a new pre-filter function θ_4 is defined for the s-box s_4.

$$\psi_2(x) = \psi_1(x) \lll 1 : \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} := \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \end{pmatrix} \tag{29}$$

Table 1. Look-up tables in 128-bit little-endian format for AES inverse affine transform function A^{-1} and our Camellia pre-filter functions θ_1, θ_4 and post-filter functions ψ_1, ψ_2 and ψ_3.

Function	Input bits	Equation	Look-up
A^{-1}	low: x_0, x_1, x_2, x_3	6	`0xa0ea347e89c31d57f2b8662cdb914f05`
	high: x_4, x_5, x_6, x_7	7	`0x5afe13b7c86c81257fdb3692ed49a400`
θ_1	low: x_0, x_1, x_2, x_3	23	`0x88258d20e34ee64b862b832eed40e845`
	high: x_4, x_5, x_6, x_7		`0x2372d283a9f858092a7bdb8aa0f15100`
θ_4	low: x_0, x_1, x_2, x_3	31	`0x74711f1a7a7f111425204e4b2b2e4045`
	high: x_4, x_5, x_6, x_7		`0xdf2e55a4d6275cad7283f8097b8af100`
ψ_1	low: x_0, x_1, x_2, x_3	24	`0xd12122d2df2f2cdc31c1c2323fcfcc3c`
	high: x_4, x_5, x_6, x_7		`0x0cf58a73db225da4a8512ed77f86f900`
ψ_2	low: x_0, x_1, x_2, x_3	29	`0xa34244a5bf5e58b9628385647e9f9978`
	high: x_4, x_5, x_6, x_7		`0x18eb15e6b744ba4951a25caffe0df300`
ψ_3	low: x_0, x_1, x_2, x_3	30	`0xe8901169ef97166e98e061199fe7661e`
	high: x_4, x_5, x_6, x_7		`0x06fa45b9ed11ae5254a817ebbf43fc00`

$$\psi_3(x) = \psi_1(x) \ggg 1 : \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} := \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{30}$$

$$\theta_4(x) = \theta_1(x \lll 1) : \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} := \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} \oplus \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \tag{31}$$

With the help of the above functions, the Camellia s_2, s_3 and s_4 functions can be presented as:

$$s_2(x) = \psi_2(\texttt{SubBytes}(\theta_1(x))) \tag{32}$$

$$s_3(x) = \psi_3(\texttt{SubBytes}(\theta_1(x))) \tag{33}$$

$$s_4(x) = \psi_1(\texttt{SubBytes}(\theta_4(x))) \tag{34}$$

As was shown for the AES inverse affine transform function in equations (5 ,6, 7), the five pre-filter and post-filter functions can be split so they take 4-bit input and output 8-bit result. From these split functions we can construct 128-bit look-up tables, which we present in Table 1. The implementation of these pre-filter and post-filter functions is the same as was show in Figure 16 for the inverse of AES affine transform.

Our byte-sliced implementation uses the filter look-ups to compute the Camellia S-boxes using the AES-NI instruction set. The byte-sliced implementation uses 128-bit XMM

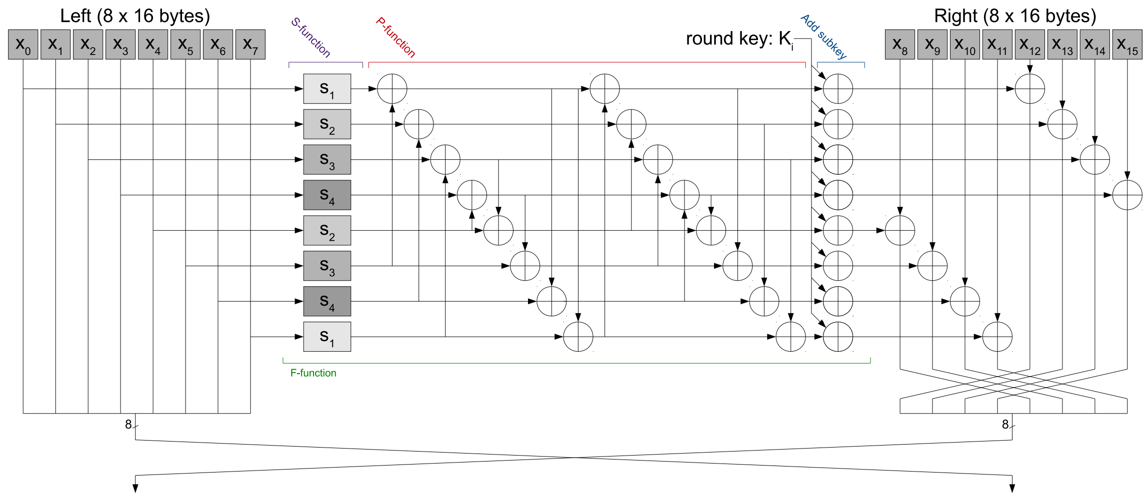


Figure 17. The structure of the Camellia round function in byte-sliced implementation. The symbol $\oplus$ means bitwise exclusive or – XOR – operation.

registers, and thus requires 16 parallel input blocks, 256 bytes, for optimum performance. The input byte-slicing transform used in our implementation does not attempt to do perfect byte-matrix transposition. Instead we perform semi-transposition so that the first bytes of input blocks are moved to the first XMM register, the second bytes to the second XMM register and so forth. However, within the XMM registers, the bytes are not stored in the original order. In other words, for example, the bytes of the first input block might be stored in the 14th elements of the XMM registers and the bytes of the second input block in the third elements of the XMM registers. The output semi-transposition and the writing results to the memory are ordered so that the blocks are output in the correct order.

Figure 17 shows the structure of our byte-sliced Feistel round function implementation. Notably the round key addition has been moved to the other side of the S-boxes, as an optimization technique (Aoki et al., 2001*b*). The left side registers '$x_0 \dots x_7$' are passed to the S-function, which is the S-box phase of the F-function. In this part appropriate pre-filtering is applied to each input bytes, then bytes are processed using the `SubBytes` function extracted from the AES-NI instruction set and finally bytes are passed through post-filtering. After the S-function, the transformed bytes are passed to the P-function phase which is straightforward implementation from the Camellia specification (Aoki et al., 2001*b*). The default F-function does the addition of the round key before the S-function. The Camellia specification gives an optimization technique, in which the round key addition can be moved to the other side of the F-function, by doing additional preprocessing of the round keys. In our case the F-function is in the form of $P(S(X)) \oplus k'$ instead of the regular $P(S(X))$ (Aoki et al., 2001*b*). The specification defines the F-function so that the upper four bytes are swapped with the lower four bytes at the end of the function. In our round function this swapping is done after adding the result of the F-function to the right side registers '$x_8 \dots x_{15}$'.

Byte-sliced implementation of the non-Feistel round of Camellia is in most parts straightforward. Although the FL and FL^{-1} functions contain only bitwise operations that mostly do not impose problem, implementing the 32-bit wide one-bit rotation requires some concern with the byte-sliced block state. The rotation implementation is shown in Figure 18. The input bytes are compared against zero using the 'signed byte compare greater than' **`vpcmpgtb`** instruction (Intel Corp., 2011). This operation is effectively the same as extracting the highest bit of byte element and filling all bits of the byte element in the target temporary register with the extracted bit. So if the highest bit was set, the byte element is set to all ones (the value `0xff`) and to zero if the bit was not set. Now we

```
/*
 * IN:
 *  v0..3: byte-sliced 32-bit integers
 *   zero: zero initialized register
 * OUT:
 *  v0..3: (IN <<< 1)
 */
#define rol32_1_16(v0, v1, v2, v3, t0, t1, t2, zero) \
        /* t0 = (v0 & 0x80) ? 0xff : 0x00 */ \
        vpcmpgtb v0, zero, t0; \
        /* v0 = v0 + v0  =>  v0 <<= 1 */ \
        vpaddb v0, v0, v0; \
        /* t0 = (t0 < 0) : -t0 ? t0  =>  t0 = (v0 >> 7) & 1 */
        vpabsb t0, t0; \
        \
        vpcmpgtb v1, zero, t1; \
        vpaddb v1, v1, v1; \
        vpabsb t1, t1; \
        \
        vpcmpgtb v2, zero, t2; \
        vpaddb v2, v2, v2; \
        vpabsb t2, t2; \
        \
        vpor t0, v1, v1; \
        \
        vpcmpgtb v3, zero, t0; \
        vpaddb v3, v3, v3; \
        vpabsb t0, t0; \
        \
        vpor t1, v2, v2; \
        vpor t2, v3, v3; \
        vpor t0, v0, v0;
```

Figure 18. The implementation of the one-bit rotate left on byte-sliced 32-bit word using the AVX instruction set.

can perform the 'get absolute value of byte' **vpabsb** instruction to the temporary register, transforming the byte elements with the value `0xff` to value `0x01`. So with the **vpcmpgtb** + **vpabsb** instruction pair we can extract the highest bits of the input byte elements in a byte-sliced manner. Now byte-wise addition is performed with the input register itself, effectively performing multiply by two or, in other words, one-bit shift to the left. The temporary register now contains the rotation carry-bits for each byte element and these are merged with the next input register. All input vector registers are processed in the same way. The rotation carry-bits of the fourth input vector are merged with the first input vector, completing the 32-bit wide one-bit rotation.

Since we need 16 input blocks, we have to use the stack memory as a temporary storage as the x86-64 architecture has only sixteen 128-bit XMM registers. Storing all of the block state for all 16 parallel input blocks to registers thus is not possible because we need registers for intermediate temporary values in the F-function. Our solution is to save the left side of block state before the round function, and save the right side state in the round function after the input for the F-function has been extracted. For the non-Feistel round, the both block state sides are input as stored in the stack and used in the calculations as memory operands.

4.5 Serpent

In this subsection, we look at an eight block parallel *word-sliced* realization of Serpent block cipher. Our implementation uses the SSE2 instruction set to load four parallel blocks to four 128-bit XMM registers that are word-sliced in 32-bit word quantities. Furthermore the 'out-of-order' parallelism is exploited by processing two sets of word-sliced input

blocks at the same time, giving us an 8-way implementation. For references to the source code of our Serpent implementation presented here, see appendix A.

Serpent is well suited for the slicing methods because of its implementation in software avoids using look-up tables (Matsui & Fukuda, 2005; Anderson et al., 1998). Instead of using the look-up tables for S-box execution, Serpent computes the S-boxes in a bit-sliced manner using bitwise logical operations to represent the S-box. Word-sliced realization of Serpent have been used by Gladman (2000), who used the 64-bit SIMD registers of the MMX instruction set to process two parallel blocks on the x86-32 architecture. Matsui and Fukuda (2005) further improved Gladman's MMX implementation, expanding the execution from the 64-bit MMX registers to the 128-bit XMM registers of the SSE2 instruction set that were available on the Intel Pentium-4 processors. On the x86-64 architecture, Lloyd (2009*b*) has demonstrated 4-way and 8-way parallel implementations of Serpent using the SSE2 instruction set. Lloyd reported that 8-way parallel implementation reached the speed of 160 MiB/s in CTR-mode on 2.4 Ghz Intel Core2 processor, which translates to 14.3 cycles per byte.

Our implementation is rooted on the paper by Osvik (2000), which gives optimized representations of the Serpent S-boxes, for two-way superscalar Intel Pentium processors. These S-box presentations are very useful for SSE2 implementation since they only require the use of one extra register in addition of the four block state registers (Osvik, 2000). The S-boxes are also presented in two-operand format that is used for the x86-64 and SSE2 instructions. The majority of SSE2 instructions use two-operand format, meaning that the instruction takes two input registers and writes result to the second input register. Contrariwise, three-operand format uses a third register as the output (Intel Corp., 2013*a*).

Osvik has published Serpent implementation which uses these S-box representations in the Linux Kernel using the C-language (Osvik, 2002). We used this implementation as the basis of our SSE2 implementation. In practice, each C-language macro in this realization was translated to assembly language. Extra work is required for word-slicing the input blocks and loading the 32-bit round keys to the 128-bit registers.

For word-slicing the four input blocks, we use **punpckldq**, **punpckhdq**, **punpcklqdq** and **punpckhqdq** unpack instructions (Lloyd, 2009*a*). These instructions read either every odd or even elements of the source registers and interleave these to the target register (Intel Corp., 2013*b*). The **punpckldq** and **punpckhdq** instruction pair work on 32-bit elements and the **punpcklqdq** and **punpckhqdq** instruction pair on 64-bit elements (Intel Corp., 2013*b*). Assembly macro below takes four input blocks loaded in four registers $x_0 \ldots x_3$ and three temporary registers $t_0 \ldots t_2$. Using the unpack instructions 4×4-matrix transposition is performed on the four input registers, word-slicing the input. The word-sliced block state is output in the same $x_0 \ldots x_3$ registers. The word-sliced state is returned into normal form by another invocation of this macro at the end of the cipher function.

```
#define transpose_4x4(x0,x1,x2,x3,t0,t1,t2) \
        movdqa x0,              t2; \
        punpckldq x1,           x0; \
        punpckhdq x1,           t2; \
        movdqa x2,              t1; \
        punpckhdq x3,           x2; \
        punpckldq x3,           t1; \
        movdqa x0,              x1; \
        punpcklqdq t1,          x0; \
        punpckhqdq t1,          x1; \
        movdqa t2,              x3; \
        punpcklqdq x2,          t2; \
        punpckhqdq x2,          x3; \
        movdqa t2,              x2;
```

To get the 32-bit round keys added to the all parallel blocks, the round key is loaded to the first 32-bit element of a temporary XMM register, then the first 32-bit element duplicated to all the four 32-bit elements of the same XMM register with the **pshufd** instruction. This temporary XMM register can then be added as the part of the round key to the word-sliced block state.

```
movd roundkey_n(CTX),   %xmm7;
pshufd $0, %xmm7,       %xmm7;
```

The instructions of the S-boxes were split in half for interleaving the two parallel SSE2 instruction stream. This was found to be the most optimal level of interleaving of instructions. The linear mixing phase of Serpent was interleaved in 10 to 11 instruction granularity.

Götzfried (2012*a*) presented an improved implementation (Götzfried, 2012*c*) of our SSE2 implementation, using the AVX instruction set to reduce the number of the register move instructions. The AVX instruction set use three-operand format as opposite of two-operand format of the SSE2 instructions. The three-operand format allows implementation to avoid extra copying register values with extra move instructions, reducing instruction count and thus gaining some more performance (Götzfried, 2012*a*). We take measurements of this implementation for comparison in our study.

4.6 Twofish

In this subsection, we detail our implementations of Twofish cipher. First we present three implementations using the table-lookup strategy for performing the g-function of the F-function. Then we look at an 8-way implementation utilizing table look-ups with combination of word-slicing with the AVX instruction set by Götzfried (2012*a*) and improve this word-sliced implementation. For references to the source code of the new Twofish implementations presented here, see appendix A.

As Twofish is a Feistel cipher, the 128-bit block state is split to four 32-bit registers in typical software implementations. Two of these registers consist of so called left side and other two the right side. In fact Twofish is designed to use 32-bit word registers and uses such operations as 32-bit integer addition and bitwise rotations on 32-bit words (Schneier et al., 1998).

Our first implementation uses the regular one-way non-parallel approach and it is constructed with minimal register usage in mind. Instead of storing the block state into four 32-bit registers, we use two 64-bit registers for storing the left and the right sides of the block state. Where needed, the state in 64-bit register is split to two 32-bit registers and then merged when the 32-bit split is no longer needed. Although the approach using 64-bit registers for storing the block state is slower than the approach using four 32-bit registers, this approach allows us to construct two block parallel and three block parallel implementations from our one-way implementation.

Our round function (see Figure 19) first performs the g-functions of the F-function with the left side block state as the input using 4 KiB look-up tables that are delivered from the master key. In this phase, the input bytes are extracted for table look-up indexes from the left side registers using the **movzbl** and **rorq** instructions in non-destructive manner. The results of table look-ups are merged with XOR operation to two 32-bit intermediate registers. Followed by the g-function is the Pseudo-Hadamard Transform (PHT) phase of the F-function, implemented using 32-bit addition for mixing the two intermediate

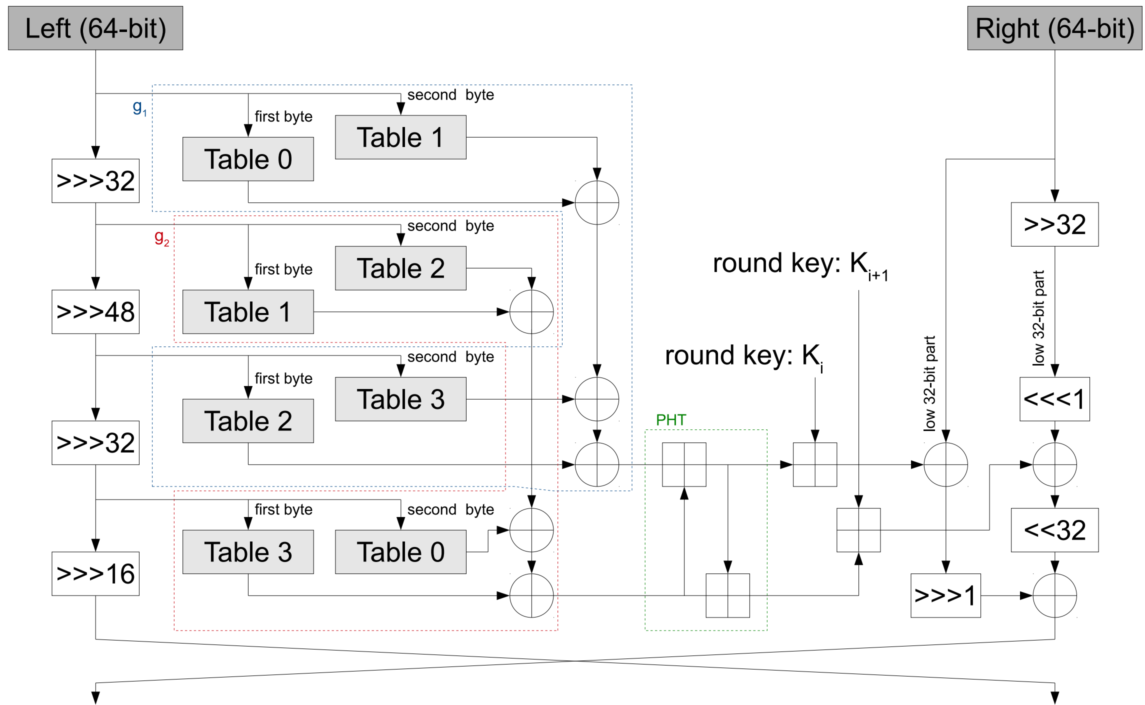


Figure 19. The structure of the Twofish round function in our implementation. The symbol ⊕ means bitwise XOR operation and ⊞ 32-bit addition operation. The ⋙ operation means bitwise right rotation while ≫ is for bitwise right shift. Operations ≪ and ⋘ mean left shift and rotation.

registers with each other and with the round key. Finally the result of the F-function is merged with the right side register with XOR operation accompanied by additional one-bit rotations of the left side registers. As the last part of the F-function uses excessively 32-bit integer operations, the 64-bit left side block state is split to two 32-bit halves and the halves are merged at the very end of the round function.

As the block state is stored in only two 64-bit registers, we can easily construct a two-way parallel implementation on top of this one-way implementation. In our two-way implementation the two block states are stored in `%rax`, `%rbx`, `%rcx` and `%rdx` from which the second byte can be accessed with the **`movzbl`** instruction. For a three-way parallel implementation we store the first half of the three block state in these registers and the second half on additional three registers `%r8`, `%r9` and `%r10`. Because these registers lack ability to access the second bytes directly, the halves of the block states are swapped between registers in the round function using the **`xchgq`** instruction. This instruction takes two registers as the input and exchanges their values (Intel Corp., 2013*b*).

Götzfried (2012*a*) implemented Twofish in an 8-way parallel manner, using word-slicing combined with table look-ups. In this implementation (Götzfried, 2012*d*), 8 input blocks are word-sliced in eight 128-bit XMM registers using the AVX instruction set. The g-function is implemented such that the word-slices are extracted from the XMM registers to the regular integer registers for table look-ups. Table indexes are then extracted from these integer registers using the move byte instruction **`movb`** and more bytes in the integer register are exposed by shifting the value to the left in multiples of 8-bits. The XOR operations for the table look-up values are combined with the memory reads. After the g-function phase, the XORed results are moved from the general purpose registers back to the XMM registers. The PHT phase and the round key addition are performed in a 32-bit word-sliced manner. Also the merging of the F-function result to the right side of the block state and the 32-bit rotations for the right side block state are performed word-sliced.

We improved Götzfried's implementation by fixing few problems that were found to make the performance of this implementation sub-optimal. First, as in his word-sliced Blowfish implementation, the use of **movb** instruction to extract bytes has bad consequences for performance because this instruction, unlike the **movzb** instruction, does not completely overwrite the target register. In our improved implementation, we have replaced the **movb** instructions with the **movzbl** instructions that zero-extend the source byte to the target register. This way we avoid the false register dependencies in byte extraction phase (Fog, 2012*a*).

We also fine-grained the ordering and the interleaving of the instructions at phase where word-slices are extracted from and inserted to the XMM registers. We moved the extraction of words from the XMM register of the next round block state before the insertion of the current round block state to the XMM register. This way the processor can proceed to do the next table look-ups faster without possibly blocking on the previous block state. The word-sliced end of the round function was also separated from the table look-up part and interleaved for better performance.

5 Evaluation

For evaluating the performance of our block cipher implementations, we used the 'System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives' framework (SUPERCOP) that has been developed by VAMPIRE – Virtual Applications and Implementation Research Lab. The SUPERCOP benchmarking framework contains facilities for conducting verifiable benchmark results for software implementations of various cryptographic primitives (Bernstein & Lange, 2013). For our implementation we use the stream cipher benchmarking facility of the SUPERCOP framework, which is based on a previous stream cipher benchmarking framework used in eSTREAM, the ECRYPT Stream Cipher Project. The eSTREAM project was an European Commission funded project from 2004 to 2008 for promoting designs of new stream ciphers (De Cannière, 2005). For evaluation purposes the project produced the eSTREAM benchmarking framework for collecting rigorous and verifiable software performance results of submitted stream ciphers on different target processors (Schwabe, 2013).

Our benchmarks use four computers with processors of different core versions. Details of these computers are listed in Table 2. The processors on older computers `nano` and `homura` have the same micro-architecture cores as that have been used in previous studies. Bernstein and Schwabe (2008) use Athlon 64 X2 processor that is based on the same AMD K8 micro-architecture as our computer `nano`, and Intel Core2 processor as is used in our computer `homura`. Also Käsper and Schwabe (2009) used Intel Core2 in their study for evaluating the performance of their bit-sliced AES implementation. Athlon 64 with AMD K8 architecture also used by Matsui (2006) in evaluating the performance of AES and Camellia implementations.

Previous research by Bernstein and Schwabe (2008) and Käsper and Schwabe (2009) have applied the eSTREAM benchmarking framework for evaluating the performance of their software implementations of AES. Since the framework is used for stream cipher performance evaluation, a block cipher mode that makes block cipher operate as a stream cipher have been used in these studies. Such mode applied in both these researches is the counter mode (CTR) that allows multiple block parallel implementations being

Table 2. Computers used for the performance evaluations.

	`nano`	`fate`
CPU model	AMD Turion 64 X2 TL-60	AMD Phenom II X6 1055T
CPU core	K8 (Tyler)	K10 (Thuban)
CPUID	60F82h	100FA0h
RAM	1.5 GiB	8 GiB
OS	Ubuntu 12.04, x86_64	Ubuntu 12.10, x86_64
	`homura`	`railgun`
CPU model	Intel Core2 T8100	Intel Core i5-2450M
CPU core	Core2 (Penryn-3M)	Sandy Bridge
CPUID	10676h	206A7h
RAM	2 GiB	6 GiB
OS	Ubuntu 12.04, x86_64	Ubuntu 12.10, x86_64

carried out (Ferguson & Schneier, 2003). The implementations in both these researches have been published in the format that is required for the eSTREAM framework. In fact the implementations by Bernstein and Schwabe (2008) form a part of the official eSTREAM benchmarking framework package (Schwabe, 2013; Bernstein, 2008). Since the eSTREAM framework has been assimilated by the SUPERCOP framework, these implementations are available to us do performance re-evaluation on the processors used in our study. For re-evaluating the bit-sliced SSSE3 implementation by Käsper and Schwabe (2009), we used a source package that was made available by the researchers at their webpage (Schwabe, 2013). Re-evaluation is important partially because the SUPERCOP framework uses a different API for integrating stream cipher implementations. Because of this, the old eSTREAM framework implementations in the SUPERCOP framework utilize an API wrapper for gluing the old eSTREAM framework API of these implementations to the newer SUPERCOP framework API. Furthermore, the output format in the SUPERCOP framework differs significantly from the eSTREAM framework. Whereas the eSTREAM framework outputs a simple numeric value for long message encryption (De Cannière, 2005), the SUPERCOP framework saves the results in large raw data format for further data processing for a researcher (Bernstein & Lange, 2013). To make sure that our data processing results are verifiable and correct, we benchmark the eSTREAM framework implementations from previous studies that have been made publicly available and compare the new results against the original research papers.

Since our goal is to obtain comparable results with the earlier research, we measure the encryption time in CPU cycles. As CPU cycle means one clock tick of the processor, it is the basic unit of time for processor when work is done (Stallings, 2009). Processors with higher clock rate can issue CPU cycles faster in the same time. By producing measures scaled to cycles per byte or per block, one can scale the cycles per byte results to the clock rate of given processor to gain the absolute speed of cipher on that specific processor. Great emphasis was put on that the clock rate modifying features of each used processor were turned off for the time of measurement. These clock rate changing features exist to lower the power usage of the processor when the system is on low usage or when extra processing power is needed for a short time. The latter is typically advertised as a turbo feature and it is most critical for the success of our measurements (Intel Corp., 2013*c*). More accurately, when the processor core is in turbo mode, the CPU cycle readings from processor are scaled, instead of clock rate reading, which remains the same. The reason behind this is to avoid breaking old software that uses the CPU cycle reading for time keeping and does not expect the speed of processor to change over time (Intel Corp., 2013*c*). So for our measurements, the turbo features and other CPU frequency scaling

features were turned off in the operating system. Furthermore, the hyper-threading feature that is used for running two threads on the same processor core on Intel processors were turned off to avoid sibling threads from affecting the measurements (Marr et al., 2002). The measurements were also pinned to run on a single core of the processor.

The SUPERCOP framework takes measurements of stream ciphers with message of sizes 0 bytes to 4096 bytes. Each measurement is repeated 15 times for each message size and the median of these measurements are presented as the result. These measurements include key-setup times in all message sizes, thus adding the key-setup overhead for every measurement. The framework also does two slightly different measurements called 'cycles' and 'xor_cycles'. The 'cycles' measurement produces the time to generate a key-stream for stream cipher, whereas the 'xor_cycles' value is for encrypting the plaintext message with this key-stream. Here the 'xor_cycles' value produces the measurement of the block cipher in counter mode for a given message size. The eSTREAM framework implementations re-evaluated in this study give the same result for 'xor_cycles' and 'cycles' because the limitations of eSTREAM framework API compared to the SUPERCOP framework API. Therefore, we use the 'xor_cycles' measurements for our results.

Our measurement runs were repeated five times and the median values from 'xor_cycles' for each message size were used. The use of medians is important to avoid running into issues that exist with cycles measurements in some processors. Matsui and Fukuda (2005) note that some Intel Pentium4 processors can give in very rare cases unrealistically short cycles values. By taking the fastest cycles values as measurements the results would be biased as such cycles values are not achievable in typical run cases. Yet even worse, this issue might be an unexpected processor behavior in which the processor outputs false cycles values in rare cases. Furthermore, the specifics of clock cycle counter on processors with the hyper-threading and the frequency scaling features are complex matter and values returned by the **`rdtsc`** instruction are not always mapped to absolute clock cycles on the hardware (Intel Corp., 2008). As such it is of absolute importance to discard measurements that are clearly faster than the baseline.

From these collected values we plotted a graph for each block cipher on each computer. Figure 20 shows the plot for Camellia implementations on computer `railgun`, showing the 'xor_cycles' measurements from message size range 0 to 4096 bytes. For each implementation, linear regression slope were calculated and gradient is used as our estimate for the speed of implementation on processing long message. Here long message means a message that has size order of magnitude larger than the 4096 bytes. With such large message, we can assume that the initial overhead on encryption becomes insignificant. By taking the slope gradient as derivative measurement, we eliminate the effects of overhead in initial measurements. In the eSTREAM framework, long stream value was gained processing several 4096 byte messages in a set time (De Cannière, 2005). The official SUPERCOP framework results for stream ciphers calculate a long message value by taking gradient between the message lengths of 1024 bytes and 4096 bytes (Bernstein & Lange, 2013). Our analysis uses all measurement data points gathered between 0 and 4096 bytes to reduce variance. We examined also linear regression between data points 2048 to 4096 bytes to see if there is significant change compared to the slope gained from 0 to 4096 bytes data points.

For those interested more specific results, the 'xor_cycles' measurements for each message size from 0 to 4096 bytes with each implementation on all four computers used in this research are made available in a form of graphs in appendix B. Furthermore, for reproducing these results, our implementation source code is available in an easily

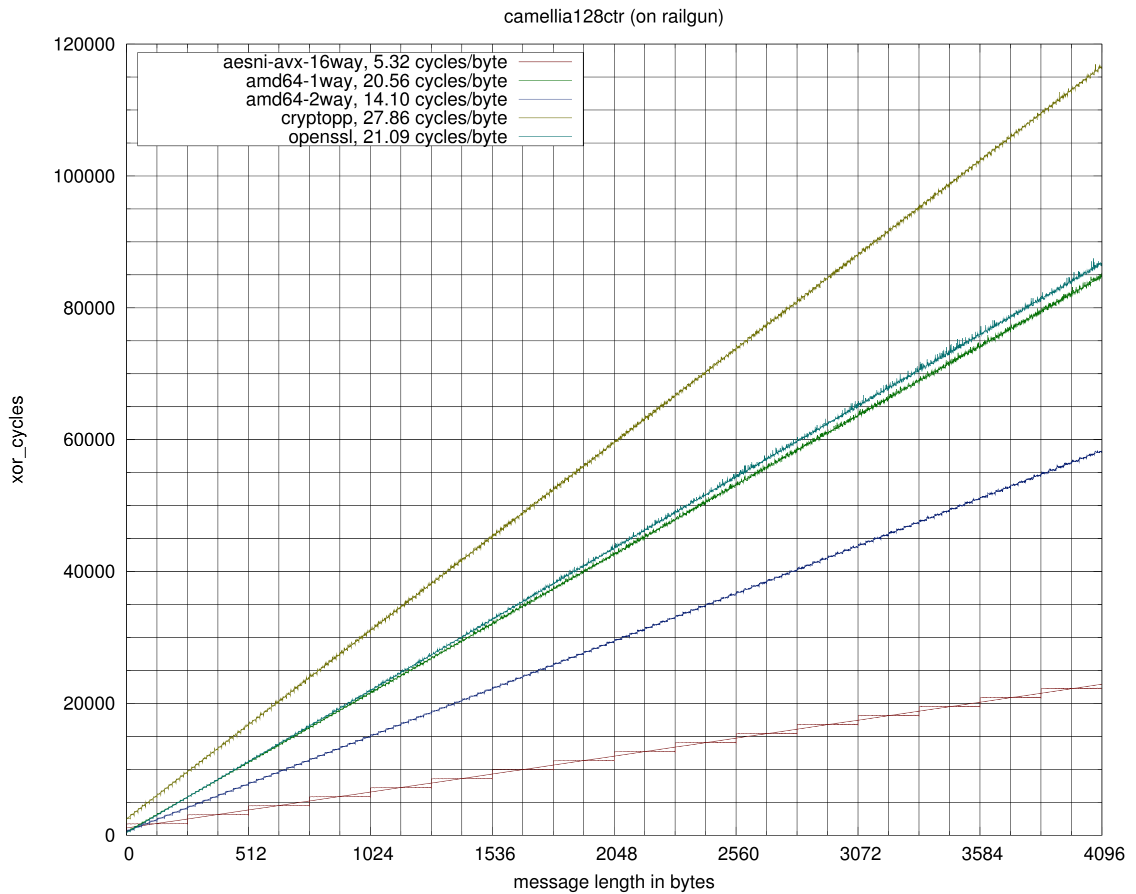


Figure 20. Plot of 'xor_cycles' in function of message length for Camellia implementations on computer `railgun`.

Table 3. Results in cycles per byte for re-evaluated AES implementations compared to the original reported research results.

Implementation	Computer	CPU core	0-4096 bytes	2048-4096 bytes
Table-based, Athlon 64 optimized	`nano`	AMD K8	10.40	10.40
	`homura`	Intel Core2	11.51	11.50
Table-based, Core2 optimized	`nano`	AMD K8	14.83	14.83
	`homura`	Intel Core2	10.55	10.55
Bit-sliced, SSSE3	`homura`	Intel Core2	7.58	7.56

	CPU core	Original result	Reference
Table-based, Athlon 64 optimized	AMD K8	10.43	Bernstein & Schwabe, 2008
	Intel Core2	11.54	
Table-based, Core2 optimized	AMD K8	14.77	Bernstein & Schwabe, 2008
	Intel Core2	10.57	
Bit-sliced, SSSE3	Intel Core2	7.59	Käsper & Schwabe, 2009

integrable form for the SUPERCOP framework. For references to the source code of the new implementations constructed in this research, see appendix A.

Table 3 shows our results from 0 to 4096 bytes slope and 2048 to 4096 bytes slope for the re-evaluated eSTREAM implementations with the original research results reported. Our results are within $\pm 0.5\%$ from the original results and the slope for message sizes 0 to 4096 bytes is giving results closer to the original results. This gives us confidence that our measurements and the analysis of data is correct and can be compared with the results from the existing literature.

Table 4. Software versions on the computers used for the measurements.

	nano and homura	fate and railgun
Ubuntu	12.04, 64-bit	12.10, 64-bit
GCC compiler	4.6.3-1ubuntu5	4.7.2-2ubuntu1
Crypto++ library	5.6.1-5build1	5.6.1-6
OpenSSL library	1.0.1-4ubuntu5.7	1.0.1c-3ubuntu2.2

For each block cipher, integration into the SUPERCOP framework started by adding a reference implementation of block cipher in counter mode. These reference implementations were based on the AES 128-bit CTR wrapper implementation for Crypto++ library (Dai, 2013). With the Crypto++ library based implementation an initial verification checksum was generated. The framework tests each implementation against this checksum to verify that cipher is correctly implemented and refuses to perform performance measurements on invalid implementations. For most implementations, the counter mode part was implemented in C language with inlined GNU assembly used for the most performance critical parts. The actual assembly implementation of the block cipher is then an ECB-mode function accepting one or multiple input blocks for non-parallel or parallel processing.

In default configuration the SUPERCOP framework tests implementations with large variation of different compilers and compiler optimization flags to find the fastest possible combination for each particular implementation. For our measurements, we limited options to the default GCC version installed on the operating system of each computer and to one set of compiler flags: '-m64 -march=native -mtune=native -O2 -fomit-frame-pointer'. Table 4 shows the compiler versions for each computer. The major reason behind this decision was that the performance critical parts of our implementations were in assembly language where different compiler settings provide no different.

For comparison we measured several other available implementations of the block ciphers in this study. These include the implementations from previous research in Table 3 and the baseline Crypto++ reference implementations used also for verification. We also measured the OpenSSL library (The OpenSSL Project, 2013) performance using wrapper implementation for Camellia and Blowfish block ciphers. For Serpent we measured the C-language implementation from the Linux Kernel by Osvik (2002, 2000) and for Twofish cipher the assembly language implementation by Fritschi (2006) from the Linux Kernel was measured. Furthermore, the word-sliced AVX implementations of Blowfish, Twofish and Serpent by Götzfried (2012*a*, 2012*c*, 2012*b*, 2012*d*) were used in the measurements.

6 Results

In this section, we present the performance results of our block cipher implementations and other implementations for comparison. The measured values are gathered using the method described in the Evaluation section. Each block cipher implementation was measured with the key length of 128-bits. We also gather here results from previous research and compare our results with results published in prior research and with results from our measurements of selected implementations.

Results reported in the literature differ from each other depending on the focus of study. There are studies that explore novel ideas for implementing particular block ciphers. While these new techniques might not necessarily give faster implementation on the processors at the time of the research, on the next generation of processors these techniques might

be successful. For example, Matsui and Fukuda (2005) presented a four block parallel word-sliced implementation of Serpent on x86-32 architecture that was not faster than earlier implementations. In our study, our similar word-sliced implementation of Serpent is over three times faster than typical C-language implementations on Intel Sandy-Bridge processor.

Another topic in the literature has been to find faster implementations for popular block ciphers, such as AES and Camellia. Bernstein and Schwabe (2008) examined the performance of table look-up implementations of AES on various processors of different architectures, including the x86-64 architecture. They report their implementation to reach the speed of 10.43 cycles per bytes on AMD K8 processor. Käsper and Schwabe (2009) presented a bit-sliced implementation of AES using SSSE3 instruction set. On Intel Nehalem processor, they reported their implementation reaching the speed of 6.92 cycles per byte and being 30% faster than any previous software implementation of AES on the x86-64 processors. The research on less popular and less standardized block ciphers appears to be limited, mainly restricted to a few year period after block cipher is first introduced.

6.1 Blowfish

Table 5 shows results of our performance measurements with Blowfish cipher on all four computers used in this study (see Table 2 for computer and processor details). For comparison, we measured the performance of Blowfish cipher in two common cryptographic libraries, Crypto++ and OpenSSL.

Table 5. The performance of various implementations of Blowfish in counter mode. References are given for implementations originating from prior research.

Implementation	Result	Reference
`nano`, AMD K8		
Our 4-way table look-up	*8.80 cycles/byte*	
Our 1-way table look-up	20.49 cycles/byte	
Crypto++	24.80 cycles/byte	
OpenSSL	27.74 cycles/byte	
`homura`, Intel Core2		
Our 4-way table look-up	*9.70 cycles/byte*	
Our 1-way table look-up	24.42 cycles/byte	
Crypto++	29.12 cycles/byte	
OpenSSL	31.68 cycles/byte	
`fate`, AMD K10		
Our 4-way table look-up	*8.67 cycles/byte*	
Our 1-way table look-up	21.90 cycles/byte	
Crypto++	25.09 cycles/byte	
OpenSSL	27.78 cycles/byte	
`railgun`, Intel Sandy-Bridge		
Our 16-way word-sliced with table look-ups, AVX	*9.53 cycles/byte*	
16-way word-sliced with table look-ups, AVX	11.53 cycles/byte	Götzfried, 2012*a*, 2012*b*
Our 4-way table look-up	11.62 cycles/byte	
Our 1-way table look-up	27.33 cycles/byte	
OpenSSL	30.35 cycles/byte	
Crypto++	31.03 cycles/byte	

As seen from the table, our one-way table look-up implementation storing block state in one 64-bit register is 10% to 20% faster than regular implementations in Crypto++ and OpenSSL libraries utilizing typical arraignment of splitting block state to two 32-bit registers. Our 4-way table look-up implementation, taking four parallel input blocks for processing, is over two times faster than our one-way implementation on all tested computers. This shows that there is not great deal of parallelism available within Blowfish. By introducing multiple blocks for parallel processing and more instruction level parallelism, we can utilize processor execution ports to much greater extent and thus gain significant increase in cipher performance.

The computer `railgun` has support for AVX instruction set available and was used to measure performance of 16-way word-sliced table look-up based implementation by Götzfried (2012*a*, 2012*b*) and our improvement of this implementation. The original implementation gains less than 1% performance improvement compared to our 4-way table look-up implementation. However, with our improvements 16-way word-sliced implementation is 22% faster than our 4-way implementation.

As summary, we note that the highest performance for Blowfish was attained with our 4-way table look-up on computer `fate`. This computer with AMD K10 CPU reached the speed of 8.67 cycles per byte, which is 2.89 times faster than our reference implementation in Crypto++. Highest relative performance improvement was seen on `railgun`. With this computer, containing Intel Sandy-Bridge CPU, improved 16-way word-sliced table look-up got the speed of 9.53 cycles per byte and was 3.18 times faster than Crypto++ implementation.

6.2 AES

Table 6 shows results of our performance measurements with AES cipher on all four computers used in this study (see Table 2 for computer and processor details). For comparison, we measured the performance of AES cipher in Crypto++ library and implementations of Bernstein and Schwabe (2008) and Käsper and Schwabe (2009) that had been published in previous research.

Our one-way implementation using two 64-bit registers to store block state is as expected slower than other non-parallel table look-up based implementation. This is because additional computation required for merging results of 8×32-bit table lookups. However, the reduced register usage allowed us to extend non-parallel implementation to two block parallel two-way implementation to introduce more instruction level parallelism. Our two-way table look-up implementation is faster than compared table look-up based implementation on three of used computers. The speed improvement however is smallish, ranging from 9% on `fate` and `nano` to 16% on `railgun`. On all three computers, two-way table look-up implementation pushes through 10 cycles per byte barrier.

On `homura` with Intel Core2 CPU, Core2 optimized table look-up implementation by Bernstein and Schwabe (2008) was 5% faster than our two-way implementation. Presumably the reason to this difference is the fact that Core2 processor has only one memory load port whereas other processors in study have two load ports (Fog, 2012*a*). Thus Core2 is unable to benefit from the additional parallelism introduced with two block parallel implementation. This is well known limitation of Core2 processors that was taken into count when Bernstein and Schwabe (2008) designed their Core2 optimized implementation.

Table 6. The performance of various implementations of AES in counter mode. References are given for implementations originating from prior research.

Implementation	Result	Reference
`nano`, AMD K8		
Our 2-way table look-up	*9.51 cycles/byte*	
Athlon 64 optimized table look-up	10.40 cycles/byte	Bernstein & Schwabe, 2008
Crypto++	10.65 cycles/byte	
Our 1-way table look-up	11.92 cycles/byte	
Core2 optimized table look-up	14.83 cycles/byte	Bernstein & Schwabe, 2008
`homura`, Intel Core2		
8-way bit-sliced SSSE3	*7.58 cycles/byte*	Käsper & Schwabe, 2009
Core2 optimized table look-up	10.55 cycles/byte	Bernstein & Schwabe, 2008
Our 2-way table look-up	11.05 cycles/byte	
Athlon 64 optimized table look-up	11.51 cycles/byte	Bernstein & Schwabe, 2008
Crypto++	12.06 cycles/byte	
Our 1-way table look-up	13.78 cycles/byte	
`fate`, AMD K10		
Our 2-way table look-up	*9.20 cycles/byte*	
Crypto++	10.04 cycles/byte	
Athlon 64 optimized table look-up	10.35 cycles/byte	Bernstein & Schwabe, 2008
Our 1-way table look-up	10.80 cycles/byte	
Core2 optimized table look-up	12.22 cycles/byte	Bernstein & Schwabe, 2008
`railgun`, Intel Sandy-Bridge		
Crypto++ (AES-NI)	*1.35 cycles/byte*	
Our 8-way bit-sliced AVX	5.83 cycles/byte	
8-way bit-sliced SSSE3	6.77 cycles/byte	Käsper & Schwabe, 2009
Our 2-way table look-up	9.78 cycles/byte	
Core2 optimized table look-up	11.36 cycles/byte	Bernstein & Schwabe, 2008
Athlon 64 optimized table look-up	11.62 cycles/byte	Bernstein & Schwabe, 2008
Our 1-way table look-up	12.80 cycles/byte	

Hamburg (2009) presented a way to implement AES using vector permutation instructions, reaching the speed of 10.0 cycles per byte in counter mode on Intel Nehalem processor. However it should be noted that our table look-up implementations utilize counter mode caching, whereas Hamburg's implementation does not. The performance difference for counter mode cached implementation is estimated to be 17% (Hamburg, 2009). Thus with counter mode caching the vector permute implementation could be estimated to reach the speed 8.3 cycles per byte.

Bit-sliced implementations were tested on computers `homura` and `railgun`. On `homura`, we got the result 7.58 cycles per byte for bit-sliced implementation by Käsper and Schwabe (2009) using SSSE3 instruction set, which is 39% faster than the Core2 optimized table look-up implementation. This is practically the same result as in original research, were performance for 4096 bytes long message was measured to be 7.59 cycles per byte. With Intel Nehalem processor, they reported performance of 6.92 cycles per byte. On `railgun`, with a newer Intel Sandy-Bridge processor, this bit-sliced SSSE3 implementation reached the speed of 6.77 cycles per byte, being 44% faster than our two-way table look-up implementation. Our improvement to bit-sliced SSSE3 implementation is the following: The bit-sliced AVX implementation reaches the performance of 5.83 cycles per byte, being 16% faster than with SSSE3.

Neither of these bit-sliced implementations utilizes counter mode caching to be constant time implementations. If the counter mode caching is utilized, the performance of AVX implementation on Intel Sandy-Bridge processors can be expected to reach 4.8 cycles per byte.

However, bit-sliced AVX implementation is not the fastest measured realization on `railgun`. The implementation in Crypto++ library utilizes AES-NI instruction set and gains speed lead with 1.35 cycles per byte. Thus Crypto++ library with AES-NI implementation is 4.3 times faster than our improved bit-sliced implementation utilizing AVX instruction set.

6.3 Camellia

Table 7 shows results of our performance measurements with Camellia cipher on all four computers used in this study (see Table 2 for computer and processor details). For comparison, we measured the performance of Camellia cipher in two common cryptographic libraries, Crypto++ and OpenSSL.

Table 7. The performance of various implementations of Camellia in counter mode.

Implementation	Result
`nano`, AMD K8	
Our 2-way table look-up	*10.38 cycles/byte*
Our 1-way table look-up	13.74 cycles/byte
OpenSSL	17.43 cycles/byte
Crypto++	25.76 cycles/byte
`homura`, Intel Core2	
Our 2-way table look-up	*12.11 cycles/byte*
Our 1-way table look-up	18.55 cycles/byte
OpenSSL	21.64 cycles/byte
Crypto++	27.71 cycles/byte
`fate`, AMD K10	
Our 2-way table look-up	*10.35 cycles/byte*
Our 1-way table look-up	13.72 cycles/byte
OpenSSL	17.67 cycles/byte
Crypto++	25.73 cycles/byte
`railgun`, Intel Sandy-Bridge	
Our 16-way byte-sliced, AVX & AES-NI	*5.32 cycles/byte*
Our 2-way table look-up	14.10 cycles/byte
Our 1-way table look-up	20.56 cycles/byte
OpenSSL	21.09 cycles/byte
Crypto++	27.86 cycles/byte

The performance of our one-way table look-up implementation storing block state in two 64-bit registers is from 4.8% to 29% faster than assembly implementation in OpenSSL. The two-way table look-up implementation, processing two parallel blocks at time, gains further speed, thanks to exposing more instruction level parallelism to out-of-order scheduling. Our two-way table look-up implementation reaches 10.38 cycles per byte on `nano` with AMD K8 CPU, which 32% faster than our one-way implementation. In the research by Matsui (2006), on which our implementation is based, performance of their two-way Camellia implementation on Athlon 64 processor was measured to be 10.9 cycles per byte. The 5% difference can originate from slight differences in implementation details,

the use of different measurement platform and the use of different block cipher mode. On AMD K10 processor, on `fate`, performance of our two-way implementation is 10.35 cycles per byte. This result is fastest measurement for the two-way table look-up implementation and it is 33% faster than the one-way implementation on the same computer.

With Intel Core2 processor on `homura` and with Intel Sandy-Bridge processor on `railgun`, our two-way implementation reaches the speed of 12.11 cycles per byte and 14.10 cycles per byte, respectively. In the research by Matsui and Nakajima (2007), bit-sliced implementation of Camellia has reported the speed of 8.44 cycles per byte on Intel Core2, being 43% faster than our two-way table look-up implementation.

On `railgun`, we measured our 16-way byte-sliced implementation utilizing AVX and AES-NI instruction sets. The AVX/AES-NI accelerated implementation reaches 5.32 cycles per byte on Intel Sandy-Bridge CPU, being 2.65 times faster than our two-way table look-up implementation on the same computer. Compared to the performance of OpenSSL implementation on the same computer, our 16-way byte-sliced implementation is 3.96 times faster. Compared to previous result with bit-sliced implementation, by Matsui and Nakajima (2007), our byte-sliced implementation is 59% faster.

6.4 Serpent

Table 8 shows results of our performance measurements with Serpent cipher on all four computers used in this study (see Table 2 for computer and processor details). For comparison, we measured the performance of Serpent cipher in Crypto++ library and also measured the performance of Serpent implementation from Linux Kernel, which is based on research and work by Osvik (2000, 2002).

Table 8. The performance of various implementations of Serpent in counter mode. References are given for implementations originating from prior research.

Implementation	Result	Reference
`nano`, AMD K8		
Our 8-way word-sliced, SSE2	*30.73 cycles/byte*	
Osvik's in Linux Kernel	35.14 cycles/byte	Osvik, 2002, 2000
Crypto++	39.38 cycles/byte	
`homura`, Intel Core2		
Our 8-way word-sliced, SSE2	*12.22 cycles/byte*	
Osvik's in Linux Kernel	47.16 cycles/byte	Osvik, 2002, 2000
Crypto++	50.45 cycles/byte	
`fate`, AMD K10		
Our 8-way word-sliced, SSE2	*13.54 cycles/byte*	
Osvik's in Linux Kernel	35.30 cycles/byte	Osvik, 2002, 2000
Crypto++	39.40 cycles/byte	
`railgun`, Intel Sandy-Bridge		
8-way word-sliced, AVX	*10.30 cycles/byte*	Götzfried, 2012*a*, 2012*c*
Our 8-way word-sliced, SSE2	11.00 cycles/byte	
Osvik's in Linux Kernel	42.72 cycles/byte	Osvik, 2002, 2000
Crypto++	46.89 cycles/byte	

Our 8-way word-sliced SSE2 implementation is faster than the reference implementations from Linux Kernel and Crypto++ on all processors. The smallest difference is seen on

nano, with AMD K8 processor, where 8-way word-sliced implementation reaches the speed of 30.73 cycles per byte, when Linux Kernel implementation gets 35.14 cycles per byte. In other words, our implementation is only 14% faster on this computer. The performance issue with nano and AMD K8 processor is that internal SIMD implementation in these processors is 64-bit. In other words the 128-bit SSE2 instructions are split to two 64-bit operations to 64-bit wide execution units, whereas on other processors used in this study, the 128-bit SSE2 operations are executed in dedicated 128-bit wide execution units (Fog, 2012*a*).

On other processors, the 8-way word-sliced SSE2 implementation sees much a greater benefit for performance. On railgun with Intel Sandy-Bridge, we get 11.00 cycles per byte which is 3.88 times higher speed than with Osvik's implementation. A similar speed up is seen with homura, the word-sliced implementation being 3.85 times faster than the old one. On fate, speed up is slightly smaller but still respectful 2.61 times faster. The previously reported 8-way SSE2 implementation by Lloyd (2009*b*), reached the speed of 14.3 cycles per byte on Intel Core2 processor. On our Intel Core2 computer homura, our implementation gets the speed of 12.22 cycles per byte, which is 17% faster than Lloyd's result. The word-sliced AVX implementation by Götzfried (2012*a*), shows small 6.8% speed improvement compared to our word-sliced SSE2 implementation on railgun. This is thanks to reduced instruction count by utilizing three-operand instruction format of the AVX instruction set. The AVX implementation is 4.15 times than Osvik's implementation from the Linux Kernel.

6.5 Twofish

Table 9 shows results of our performance measurements with Twofish cipher on all four computers used in this study (see Table 2 for computer and processor details). For comparison, we measured the performance of Twofish cipher in Crypto++ library and also the performance of Twofish x86-64 assembly implementation from Linux Kernel by Fritschi (2006).

Our one-way table look-up implementation using two 64-bit registers to store block state is slower than the more typical Twofish implementation from Linux Kernel using four 32-bit registers for holding block state. Fritschi's implementation is measured to be from 11% on homura to 20% on nano faster than our one-way table look-up implementation. However, reduced register usage allows building two and three block parallel implementations from our one-way table look-up implementation. Our two-way parallel table look-up implementation is faster than three-way on nano, while on the other computers the three-way parallel implementation beats the two-way technique. On nano with AMD K8, our two-way table look-up implementation reaches the speed of 12.43 cycles per byte, being 17% faster than the reference implementation from Linux Kernel. The highest performance that our three-way table look-up implementation reaches is on fate, with result 12.06 cycles per byte. The highest relative improvement in speed was recorded on homura, where three-way parallel implementation is 35.7% faster than the Linux Kernel implementation.

The computer railgun with Intel Sandy-Bridge processor had a support for AVX instruction set available and was used to measure the performance of 8-way word-sliced table look-up based implementation by Götzfried (2012*a*, 2012*d*) as well as our improvement. Götzfried's implementation gains 34.8% performance improvement compared to our three-way table look-up implementation. However, with our improvements to the 8-way

Table 9. The performance of various implementations of Twofish in counter mode. References are given for implementations originating from prior research.

Implementation	Result	Reference
`nano`, AMD K8		
Our 2-way table look-up	*12.43 cycles/byte*	
Our 3-way table look-up	13.26 cycles/byte	
Linux Kernel, assembly	14.51 cycles/byte	Fritschi, 2006
Our 1-way table look-up	17.39 cycles/byte	
Crypto++	19.68 cycles/byte	
`homura`, Intel Core2		
Our 3-way table look-up	*13.40 cycles/byte*	
Our 2-way table look-up	14.35 cycles/byte	
Linux Kernel, assembly	18.18 cycles/byte	Fritschi, 2006
Our 1-way table look-up	20.21 cycles/byte	
Crypto++	20.51 cycles/byte	
`fate`, AMD K10		
Our 3-way table look-up	*12.06 cycles/byte*	
Our 2-way table look-up	12.17 cycles/byte	
Linux Kernel, assembly	15.19 cycles/byte	Fritschi, 2006
Our 1-way table look-up	17.31 cycles/byte	
Crypto++	18.39 cycles/byte	
`railgun`, Intel Sandy-Bridge		
Our 8-way word-sliced, AVX	*9.93 cycles/byte*	
8-way word-sliced, AVX	11.28 cycles/byte	Götzfried, 2012*a*, 2012*d*
Our 3-way table look-up	15.20 cycles/byte	
Our 2-way table look-up	15.36 cycles/byte	
Linux Kernel, assembly	19.11 cycles/byte	Fritschi, 2006
Our 1-way table look-up	21.45 cycles/byte	
Crypto++	21.90 cycles/byte	

word-sliced implementation, the measured performance is increased by 13.6% compared to Götzfried's original implementation. Our improved 8-way word-sliced implementation reaches the speed of 9.93 cycles per byte and it is 53.1% faster than our three-way parallel table look-up implementation.

7 Conclusions

This thesis was about reviewing different optimization techniques used on the x86-64 architecture when implementing different block ciphers, and applying these techniques when constructing new faster implementations of the selected block ciphers. The use of different techniques on particular block cipher algorithm depends greatly on the actual design of the algorithm. Typically block ciphers are designed so that fast software implementations can utilize the use of large look-up tables for the substitution and permutation phases found in round function. A counterexample to this is the Serpent cipher that is designed so that the cipher can be implemented in software in a bit-sliced manner. The bit-slicing has been used to construct faster software implementations of other block ciphers, such as AES and Camellia. One complication for the use of bit-slicing technique is the need to process multiple plaintext blocks in parallel, unless the cipher is specially designed for bit-slicing such as Serpent.

The parallel processing can be accomplished with specific block cipher modes of operation, such as counter mode, that allow it. Modes of operation that have ciphertext block feedback, require the previous plaintext block to be encrypted before, serializing the encryption process. Such modes of operation cannot benefit from the additional performance introduced by parallel processing. For measurements we used the SUPERCOP framework, where block ciphers were tested in counter mode. Therefore most of our implementations process blocks in parallel to reach higher performance.

Parallel processing can be applied to the table look-up based implementations. The out-of-order scheduling, that is available in most x86-64 processors today, can find instruction level parallelism and execute independent instruction streams in parallel. Matsui (2006) presented this technique for implementing the Camellia cipher in two-way parallel manner, interleaving two block encryption operations to introduce more parallelism for processor to exploit and yield higher performance. We implemented the Camellia cipher in the same manner and were able to reproduce the positive results of this treatment. We successfully applied this technique to table look-up based implementations Blowfish, AES and Twofish. The 4-way parallel implementation of Blowfish was 2.89 times faster than reference implementation on AMD K10 processor. For AES and Twofish implementations, we introduced additional overhead in storing block state in 64-bit registers and as a result the non-parallel table-lookup implementations were slower than reference implementations. However, reduced register usage in these constructs allowed us to build parallel implementations that were able to regain the lost performance and get additional speed. Our two-way parallel table look-up based AES implementation reaches 9.51 cycles per byte on AMD K8 processor.

The bit-slicing technique was experimented with the AES cipher. We converted the previous 8-way parallel bit-sliced implementation utilizing SSSE3 instruction set by Käsper and Schwabe (2009) to newer AVX instruction set which then reached the speed of 5.83 cycles per byte on Intel Sandy-Bridge processor.

Other slicing techniques, such as byte-slicing and word-slicing, were also examined. The substitution function in Camellia cipher is very closely related to the substitution function in AES cipher. The AES-NI instruction set provided the building blocks of AES cipher for us to construct byte-sliced implementation of Camellia cipher. Our 16-way parallel Camellia implementation utilized the AES-NI and AVX instruction sets and it is measured to have the speed of 5.32 cycles per byte on Intel Sandy-Bridge processor. The word-slicing technique was applied to Serpent cipher using SSE2 instruction in 4-way parallel manner. Two 4-way instruction streams for out-of-order processing were introduced to create 8-way parallel Serpent implementation which reached speed 11.00 cycles per byte. Combining the table look-up technique with word-slicing using AVX instruction set presented by Götzfried (2012*a*) was also investigated and previous implementations of this combined technique were improved.

There is still room for improvement in our implementations. For example, counter mode caching was only used on AES implementations. This optimization technique might be possible to apply to other ciphers. We chose not to examine this technique on other ciphers than AES as it can be applied with one mode of operation. For AES, we applied this technique to get comparable results with previous research.

The x86-64 architecture contains a vast amount of different instruction sets and possibilities for applying different optimization techniques. Some instruction sets are made obsolete by introduction of new instruction sets that contain the old instructions in better form. Such instruction set is AVX which extended the previous SSE and SSSE instruction sets to

three-operand format. These new instruction sets that are introduced in every few years, make the art and science of constructing fast software implementations on the x86-64 architecture a rapidly developing field. This thesis was completed in May 2013, one month before the new Intel Haswell processors were about to be released. These fresh processors include new AVX2 instruction set that widens the integer vector registers from 128-bits to 256-bits, doubling the peak integer vector performance. Furthermore, the AVX2 instruction set includes new exciting instructions that can be used to construct even faster software implementation of block ciphers. One such group of instructions in AVX2 is the vector gather instructions that allow the executing of parallel table look-ups from vector registers. We have produced implementations that utilize the new features of AVX2 instruction set and submitted some of these for inclusion to Linux Kernel (Kivilinna, 2013). We chose not to include these implementations in this thesis since we did not have a processor that supports AVX2 available for measurements.

Some of the processors used in this research were quite old, and we did not have the latest generation of processors available for measurements. It would have been interesting to get measurements on new AMD processors, such AMD Bulldozer or AMD Piledriver, as these processors have microarchitecture that is significantly different than previous generation AMD K8 and K10 that were used in this research. The newer Intel Ivy-Bridge is closely related to the Intel Sandy-Bridge and differences in measurements between two can be expected to be small. On the other hand, the use of older processors allowed us to compare our results with previous research and verify that our measure methods gave identical results.

An obvious future research subject is to continue work on future instruction set extensions. As the field moves forward, our implementations presented here are soon becoming obsolete. The AVX2 instruction set in addition of vector gather instructions, contain other interesting new possibilities for cryptographic implementations, such as bit permutation instructions. Another direction for further research is to examine application of optimization techniques used on the x86-64 architecture to other current and future architectures such as ARM and AArch64. The AArch64 architecture is new 64-bit architecture based on previous 32-bit ARM architecture. As modern architecture, it contains vector register instructions and processors are expected to have out-of-order scheduling capabilities. Furthermore, cryptographic instructions such as in AES-NI are expected to be included in future AArch64 processors making it possible to implement Camellia and other algorithms in fast byte-sliced manner.

References

Akdemir, K., Dixon, M., Feghali, W., Fay, P., Gopal, V., Guilford, J., … Zohar, R. (2010). *Breakthrough AES Performance with Intel AES New Instructions* (Vol. 10TB24; Tech. Rep.). Intel Corporation.

Anderson, R. (2008). *Security engineering : a guide to building dependable distributed systems* (2nd ed.). Indianapolis: Wiley.

Anderson, R., Biham, E., & Knudsen, L. (1998). Serpent: A Proposal for the Advanced Encryption Standard. In *First AES Candidate Conference (AES1).* National Institute of Standard and Technology.

Aoki, K., Ichikawa, T., Kanda, M., Matsui, M., Moriai, S., Nakajima, J., & Tokita, T. (2001*a*). Camellia: A 128-Bit Block Cipher Suitable for Multiple Platforms — Design and Analysis. In D. R. Stinson & S. Tavares (Eds.), *Lecture Notes in Computer Science* (Vol. 2012, pp. 39–56). Springer Berlin Heidelberg. doi: 10.1007/3-540-44983-3_4

Aoki, K., Ichikawa, T., Kanda, M., Matsui, M., Moriai, S., Nakajima, J., & Tokita, T. (2001*b*). *Specification of Camellia — a 128-bit Block Cipher (version 2.0).* Retrieved December 28, 2012, from `http://info.isl.ntt.co.jp/crypt/eng/camellia/specifications.html`

Aoki, K., Matusiewicz, K., Roland, G., Sasaki, Y., & Schläffer, M. (2012). Byte Slicing Grøstl: Improved Intel AES-NI and Vector-Permute Implementations of the SHA-3 Finalist Grøstl. In M. S. Obaidat, J. Sevillano, & J. Filipe (Eds.), *Communications in Computer and Information Science* (Vol. 314, pp. 281–295). Springer Berlin Heidelberg. doi: 10.1007/978-3-642-35755-8_20

Bernstein, D. J. (2008). *Stream-cipher timings.* Retrieved March 21, 2013, from `http://cr.yp.to/streamciphers/timings.html`

Bernstein, D. J., & Lange, T. (Eds.). (2013). *eBACS: ECRYPT Benchmarking of Cryptographic Systems.* Retrieved March 21, 2013, from `http://bench.cr.yp.to`

Bernstein, D. J., & Schwabe, P. (2008). New AES Software Speed Records. In D. R. Chowdhury, V. Rijmen, & A. Das (Eds.), *Lecture Notes in Computer Science* (Vol. 5365, pp. 322–336). Springer Berlin Heidelberg. doi: 10.1007/978-3-540-89754-5_25

Bhaskar, R., Dubey, P. K., Kumar, V., & Rudra, A. (2003). Efficient galois field arithmetic on SIMD architectures. In *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures* (pp. 256–257). New York, NY, USA: ACM. doi: 10.1145/777412.777458

Biham, E. (1997). A fast new DES implementation in software. In E. Biham (Ed.), *Lecture Notes in Computer Science* (Vol. 1267, pp. 260–272). Springer Berlin Heidelberg. doi: 10.1007/BFb0052352

Bryant, R. E., & O'Hallaron, D. R. (2005, September). *x86-64 Machine-Level Programming.* Retrieved May 03, 2013, from `http://www.cs.cmu.edu/~fp/courses/15213-s07/misc/asm64-handout.pdf`

Buxton, M. (2011). *Haswell New Instruction Descriptions Now Available!* Retrieved March 8, 2013, from `http://software.intel.com/en-us/blogs/2011/06/13/haswell-new-instruction-descriptions-now-available`

Daemen, J., & Rijmen, V. (2000). The Block Cipher Rijndael. In J.-J. Quisquater & B. Schneier (Eds.), *Lecture Notes in Computer Science* (Vol. 1820, pp. 277–284). Springer Berlin Heidelberg. doi: 10.1007/10721064_26

Daemen, J., & Rijmen, V. (2002). *The Design of Rijndael: AES - The Advanced Encryption Standard*. Berlin: Springer.

Dai, W. (2013). *Crypto++ Library.* Retrieved from `http://www.cryptopp.com`

De Cannière, C. (2005). *eSTREAM Optimized Code HOWTO.* Retrieved March 21, 2013, from `http://www.ecrypt.eu.org/stream/perf`

Erdelsky, P. J. (2002). *Rijndael Encryption Algorithm.* Retrieved March 8, 2013, from `http://www.efgh.com/software/rijndael.htm`

Ferguson, N., & Schneier, B. (2003). *Practical Cryptography*. New York: Wiley.

Fog, A. (2012*a*). *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers.* Retrieved March 8, 2013, from `http://www.agner.org/optimize/microarchitecture.pdf`

Fog, A. (2012*b*). *Optimizing subroutines in assembly language: An optimization guide for x86 platforms.* Retrieved March 8, 2013, from `http://www.agner.org/optimize/optimizing_assembly.pdf`

Fritschi, J. (2006). *Twofish cipher algorithm (x86_64).* Retrieved March 21, 2013, from `http://lxr.linux.no/linux+v3.0/arch/x86/crypto/twofish-x86_64-asm_64.S`

Fruhwirth, C. (2011). *LUKS On-Disk Format Specification (Version 1.2.1).* Retrieved from `http://wiki.cryptsetup.googlecode.com/git/LUKS-standard/on-disk-format.pdf`

Gladman, B. (2000). *Serpent S Boxes.* Retrieved March 8, 2013, from `http://gladman.plushost.co.uk/oldsite/cryptography_technology/serpent/index.php`

GNU Project. (2013). *GCC, The GNU Compiler Collection.* Retrieved from `http://gcc.gnu.org`

Götzfried, J. (2012*a*). *Advanced Vector Extensions to Accelerate Crypto Primitives.* Bachelor's Thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen, Germany. Retrieved March 16, 2013, from `http://www1.cs.fau.de/avx.crypto`

Götzfried, J. (2012*b*). *Blowfish Cipher 16-way parallel algorithm (AVX/x86_64).* Retrieved February 14, 2013, from `http://www1.informatik.uni-erlangen.de/filepool/projects/avx.crypto/blowfish-avx-x86_64-asm_64.S`

Götzfried, J. (2012*c*). *Serpent Cipher 8-way parallel algorithm (x86_64/AVX).* Retrieved February 14, 2013, from `http://www1.informatik.uni-erlangen.de/filepool/projects/avx.crypto/serpent-avx-x86_64-asm_64.S`

Götzfried, J. (2012*d*). *Twofish Cipher 8-way parallel algorithm (AVX/x86_64).* Retrieved February 14, 2013, from `http://www1.informatik.uni-erlangen.de/filepool/projects/avx.crypto/twofish-avx-x86_64-asm_64.S`

Gueron, S. (2009). Intel's New AES Instructions for Enhanced Performance and Security. In O. Dunkelman (Ed.), *Lecture Notes in Computer Science* (Vol. 5665, pp. 51–66). Springer Berlin Heidelberg. doi: 10.1007/978-3-642-03317-9_4

Hamburg, M. (2009). Accelerating AES with Vector Permute Instructions. In C. Clavier & K. Gaj (Eds.), *Lecture Notes in Computer Science* (Vol. 5747, pp. 18–32). Springer Berlin Heidelberg. doi: 10.1007/978-3-642-04138-9_2

Intel Corp. (2008). *Intel Pentium 4 Processor – Specification Update (249199-071).* Santa Clara, CA, USA. Retrieved from `http://download.intel.com/design/intarch/specupdt/24919969.pdf`

Intel Corp. (2011). *Intel Advanced Vector Extensions Programming Reference (319433-*

011). Santa Clara, CA, USA. Retrieved from `http://software.intel.com/file/36945`

Intel Corp. (2013*a*). *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture.* Santa Clara, CA, USA. Retrieved from `http://www.intel.com/products/processor/manuals`

Intel Corp. (2013*b*). *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z.* Santa Clara, CA, USA. Retrieved from `http://www.intel.com/products/processor/manuals`

Intel Corp. (2013*c*). *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 3A, 3B, and 3C: System Programming Guide, Parts 1 and 2.* Santa Clara, CA, USA. Retrieved from `http://www.intel.com/products/processor/manuals`

International Organization for Standardization. (2010). *ISO/IEC 18033-3:2010 — Information technology – Security techniques – Encryption algorithms – Part 3: Block ciphers.*

Irvine, K. R. (2010). *Assembly Language for x86 Processors* (6th ed.). New Jersey: Prentice Hall.

Käsper, E., & Schwabe, P. (2009). Faster and Timing-Attack Resistant AES-GCM. In C. Clavier & K. Gaj (Eds.), *Lecture Notes in Computer Science* (Vol. 5747, pp. 1–17). Springer Berlin Heidelberg. doi: 10.1007/978-3-642-04138-9_1

Katz, J., & Lindell, Y. (2008). *Introduction to Modern Cryptography*. Boca Raton: Chapman & Hall/CRC.

Kerckhoffs, A. (1883). La cryptographie militaire. *Journal des sciences militaires*, *IX*, 5–38. Retrieved from `http://www.petitcolas.net/fabien/kerckhoffs`

Kivilinna, J. (2013, April). *Add AVX2 accelerated implementations for Blowfish, Twofish, Serpent and Camellia.* Retrieved May 03, 2013, from `http://marc.info/?l=linux-kernel&m=136585000432342&w=2`

Lloyd, J. (2009*a*). *Programming trivia: 4x4 integer matrix transpose in SSE2.* Retrieved April 23, 2013, from `http://www.randombit.net/bitbashing/2009/10/08/integer_matrix_transpose_in_sse2.html`

Lloyd, J. (2009*b*). *Speeding up Serpent: SIMD Edition.* Retrieved December 28, 2012, from `http://randombit.net/bitbashing/2009/09/09/serpent_in_simd.html`

Marr, D. T., Binns, F., Hill, D. L., Hinton, G., Koufaty, D. A., Miller, J. A., & Upton, M. (2002). Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, *6*(1), 4–15.

Mathews, M., & Hunt, R. (2007). Evolution of Wireless LAN Security Architecture to IEEE 802.11I (WPA2). In *Proceedings of the Fourth IASTED Asian Conference on Communication Systems and Networks* (pp. 292–297). Anaheim, CA, USA: ACTA Press.

Matsui, M. (2006). How Far Can We Go on the x64 Processors? In M. Robshaw (Ed.), *Lecture Notes in Computer Science* (Vol. 4047, pp. 341–358). Springer Berlin Heidelberg. doi: 10.1007/11799313_22

Matsui, M., & Fukuda, S. (2005). How to Maximize Software Performance of Symmetric Primitives on Pentium III and 4 Processors. In H. Gilbert & H. Handschuh (Eds.), *Lecture Notes in Computer Science* (Vol. 3557, pp. 398–412). Springer Berlin Heidelberg. doi: 10.1007/11502760_27

Matsui, M., & Nakajima, J. (2007). On the Power of Bitslice Implementation on Intel Core2 Processor. In P. Paillier & I. Verbauwhede (Eds.), *Lecture Notes in Computer Science* (Vol. 4727, pp. 121–134). Springer Berlin Heidelberg. doi:

10.1007/978-3-540-74735-2_9

Mollin, R. A. (2005). *Codes: The Guide to Secrecy from Ancient to Modern Times*. Boca Raton: Chapman & Hall/CRC.

Moona, R. (2007). *Assembly Language Programming in GNU/Linux for IA32 Architectures*. New Delhi: Prentice-Hall of India.

NESSIE consortium. (2003). *Portfolio of recommended cryptographic primitives.* Retrieved April 5, 2013, from `https://www.cosic.esat.kuleuven.be/nessie`

Neves, S., & Aumasson, J.-P. (2012). BLAKE and 256-bit Advanced Vector Extensions. In *The Third SHA-3 Candidate Conference.* National Institute of Standards and Technology.

Osvik, D. A. (2000). Speeding up Serpent. In *Third AES Candidate Conference (AES3)* (pp. 317–329). New York, New York, USA: National Institute of Standards and Technology.

Osvik, D. A. (2002). *Cryptographic API: Serpent Cipher Algorithm.* Retrieved March 8, 2013, from `http://www.ii.uib.no/~osvik/serpent/serpent.c`

Padgett, N., & Kerner, M. (2007, February). *The History of Modern 64-bit Computing*. Retrieved May 03, 2013, from `http://www.cs.washington.edu/education/courses/csep590/06au/projects/history-64-bit.pdf`

Potlapally, N. R., Ravi, S., Raghunathan, A., & Jha, N. K. (2003). Analyzing the energy consumption of security protocols. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design* (pp. 30–35). New York, NY, USA: ACM. doi: 10.1145/871506.871518

Rebeiro, C., Selvakumar, D., & Devi, A. S. L. (2006). Bitslice Implementation of AES. In D. Pointcheval, Y. Mu, & K. Chen (Eds.), *Lecture Notes in Computer Science* (Vol. 4301, pp. 203–212). Springer Berlin Heidelberg. doi: 10.1007/11935070_14

Robshaw, M. J. B. (1995, August). *Block Ciphers* (Tech. Rep. No. TR-601 version 2.0). RSA Laboratories.

Rudra, A., Dubey, P. K., Jutla, C. S., Kumar, V., Rao, J. R., & Rohatgi, P. (2001). Efficient Rijndael Encryption Implementation with Composite Field Arithmetic. In c. K. Koç, D. Naccache, & C. Paar (Eds.), *Lecture Notes in Computer Science* (Vol. 2162, pp. 171–184). Springer Berlin Heidelberg. doi: 10.1007/3-540-44709-1_16

Satoh, A., & Morioka, S. (2003). Unified Hardware Architecture for 128-Bit Block Ciphers AES and Camellia. In C. D. Walter, c. K. Koç, & C. Paar (Eds.), *Lecture Notes in Computer Science* (Vol. 2779, pp. 304–318). Springer Berlin Heidelberg. doi: 10.1007/978-3-540-45238-6_25

Schneier, B. (1994*a*). *The Blowfish algorithm.* Retrieved March 9, 2013, from `https://www.schneier.com/sccd/BFSH-SCH.ZIP`

Schneier, B. (1994*b*). Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish). In R. Anderson (Ed.), *Lecture Notes in Computer Science* (Vol. 809, pp. 191–204). Springer Berlin Heidelberg. doi: 10.1007/3-540-58108-1_24

Schneier, B. (2013*a*). *Products that Use Blowfish.* Retrieved April 24, 2013, from `https://www.schneier.com/blowfish-products.html`

Schneier, B. (2013*b*). *Products that Use Twofish.* Retrieved April 24, 2013, from `https://www.schneier.com/twofish-products.html`

Schneier, B., Kelsey, J., Whiting, D., Wagner, D., Hall, C., & Ferguson, N. (1998). Twofish: A 128-Bit Block Cipher. In *First AES Candidate Conference (AES1).* National Institute of Standard and Technology.

Schwabe, P. (2013). *cryptojedi.org – Cryptography.* Retrieved March 21, 2013, from `http://cryptojedi.org/crypto/index.shtml`

Shannon, C. E. (1949). Communication Theory of Secrecy Systems. *Bell Systems Technical Journal*, *28*(4), 656–715.

Stallings, W. (2009). *Computer Organization and Architecture: Designing for Performance* (8th ed.). Upper Saddle River, NJ: Prentice Hall.

Stallings, W. (2010). *Cryptography and Network Security: Principles and Practice* (5th ed.). Boston: Prentice Hall.

The CRYPTREC Advisory Committee. (2003). *e-Government Recommended Ciphers List.* Retrieved April 5, 2013, from `http://www.cryptrec.go.jp/english/list.html`

The OpenSSL Project. (2013). *OpenSSL.* Retrieved from `https://www.openssl.org`

Tran, T. H., Cho, H.-M., & Cho, S.-B. (2009). Performance enhancement of motion estimation using SSE2 technology. In *Proceedings of World Academy of Science, Engineering and Technology* (Vol. 40, pp. 168–171).

TrueCrypt Foundation. (2013). *TrueCrypt.* Retrieved from `http://www.truecrypt.org`

Wright, C., Dave, J., & Zadok, E. (2003). Cryptographic File Systems Performance: What You Don't Know Can Hurt You. In *Security in Storage Workshop, 2003. SISW '03. Proceedings of the Second IEEE International* (p. 47). doi: 10.1109/SISW.2003.10005

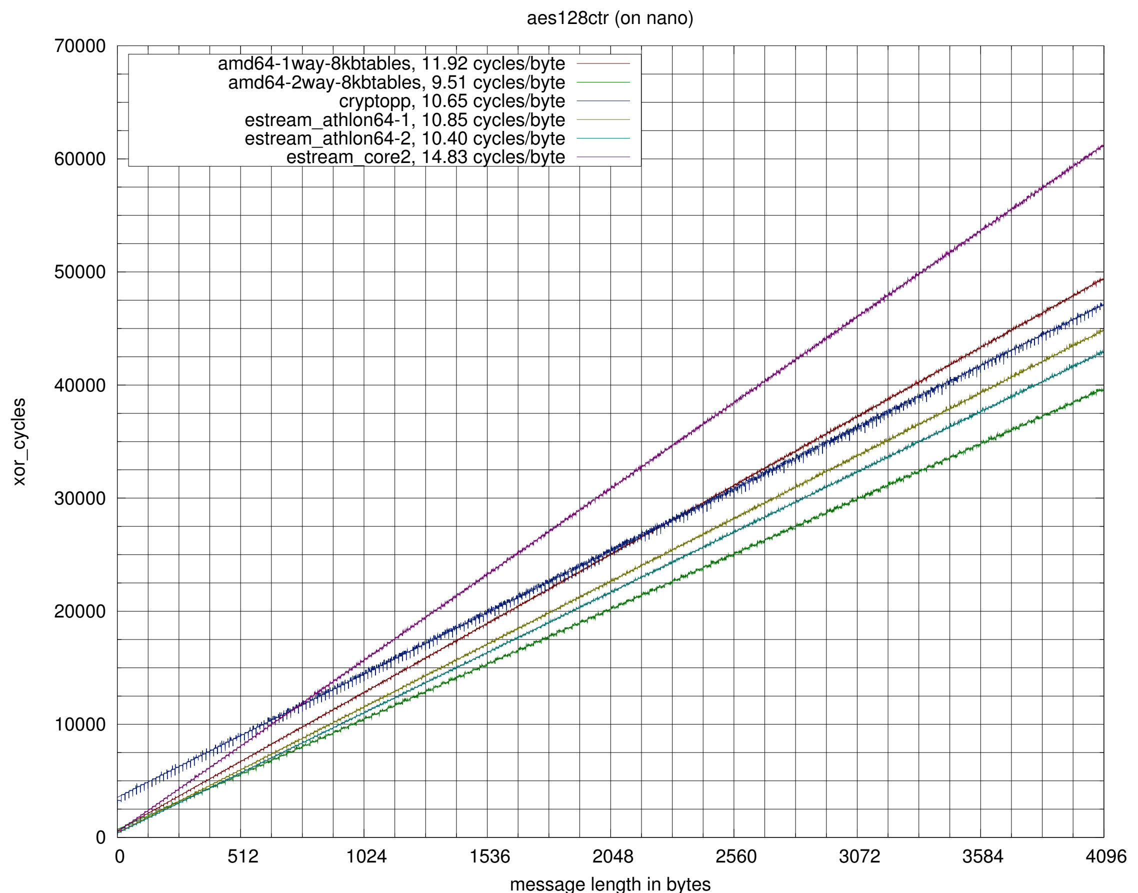
Appendix A Links to source code produced in this research

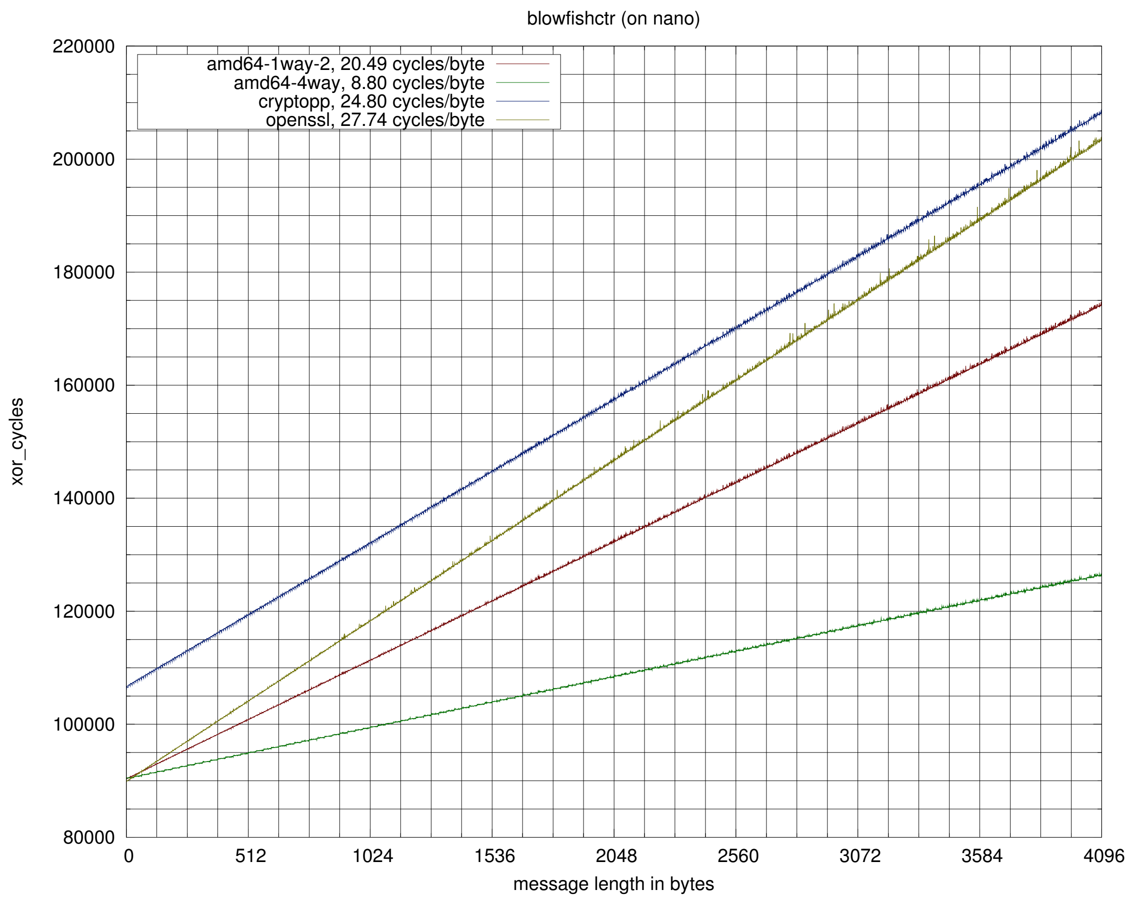
The implementations constructed and measured in this research are available at `https://github.com/jkivilin/supercop-blockciphers`. This git-repository contains the implementations prepared for easy direct integration to SUPERCOP framework.

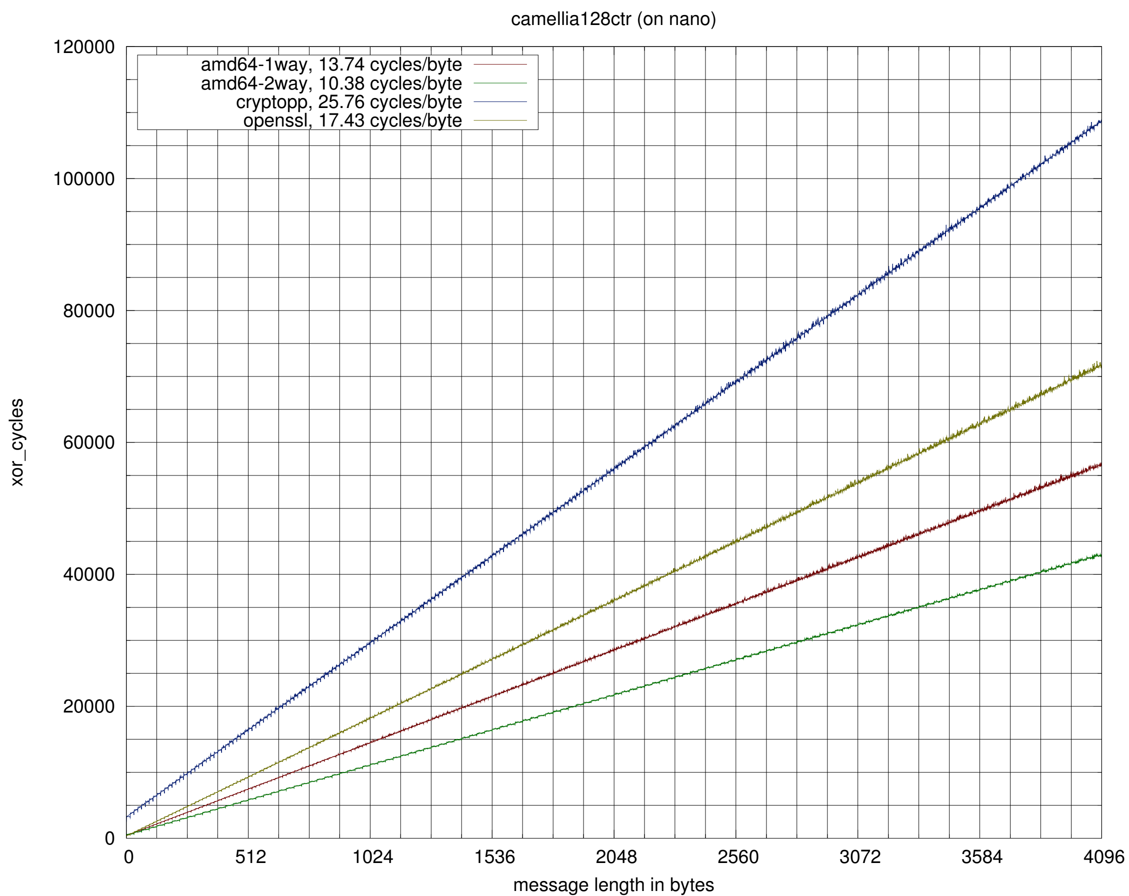
The author also has committed the 4-way parallel Blowfish, 8-way parallel SSE2 Serpent, 3-way parallel Twofish, 2-way parallel Camellia and 16-way parallel AES-NI/AVX Camellia implementations to the Linux Kernel (see `http://www.kernel.org/`).

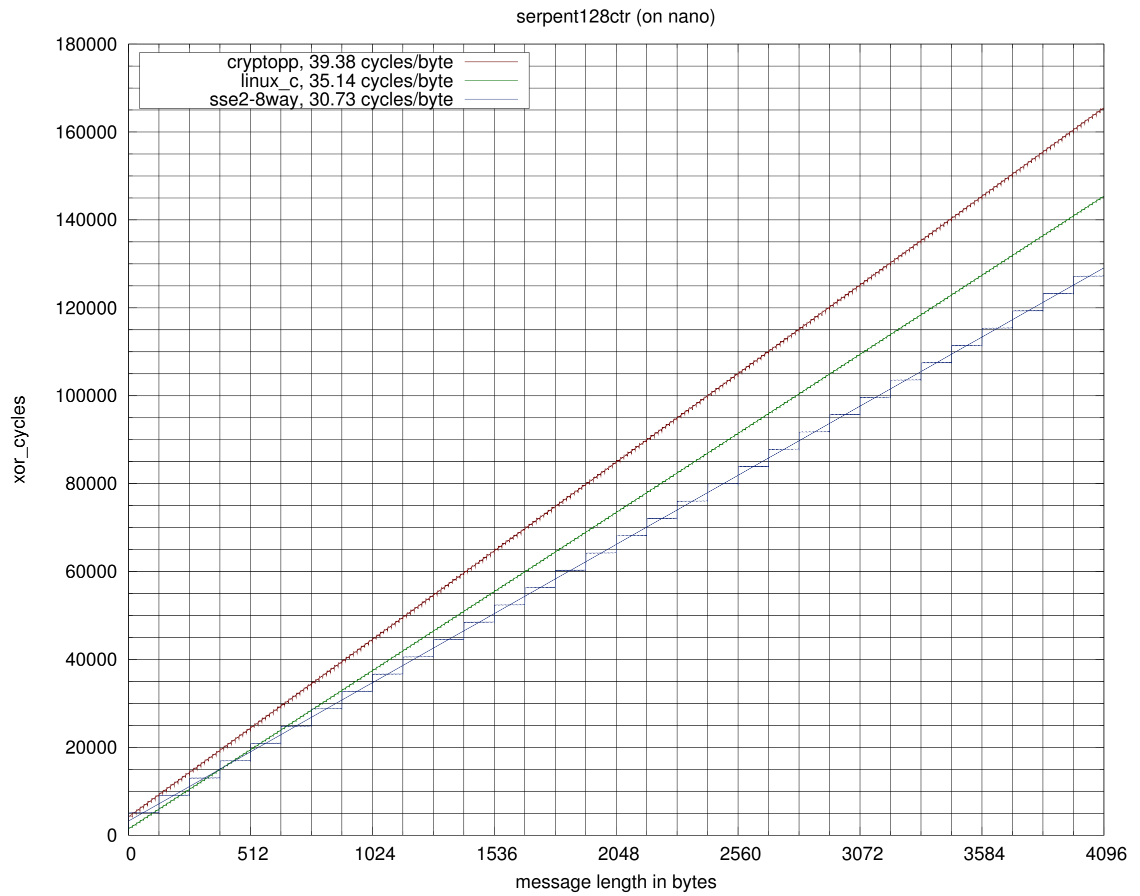
Appendix B Graphs from the SUPERCOP measurements

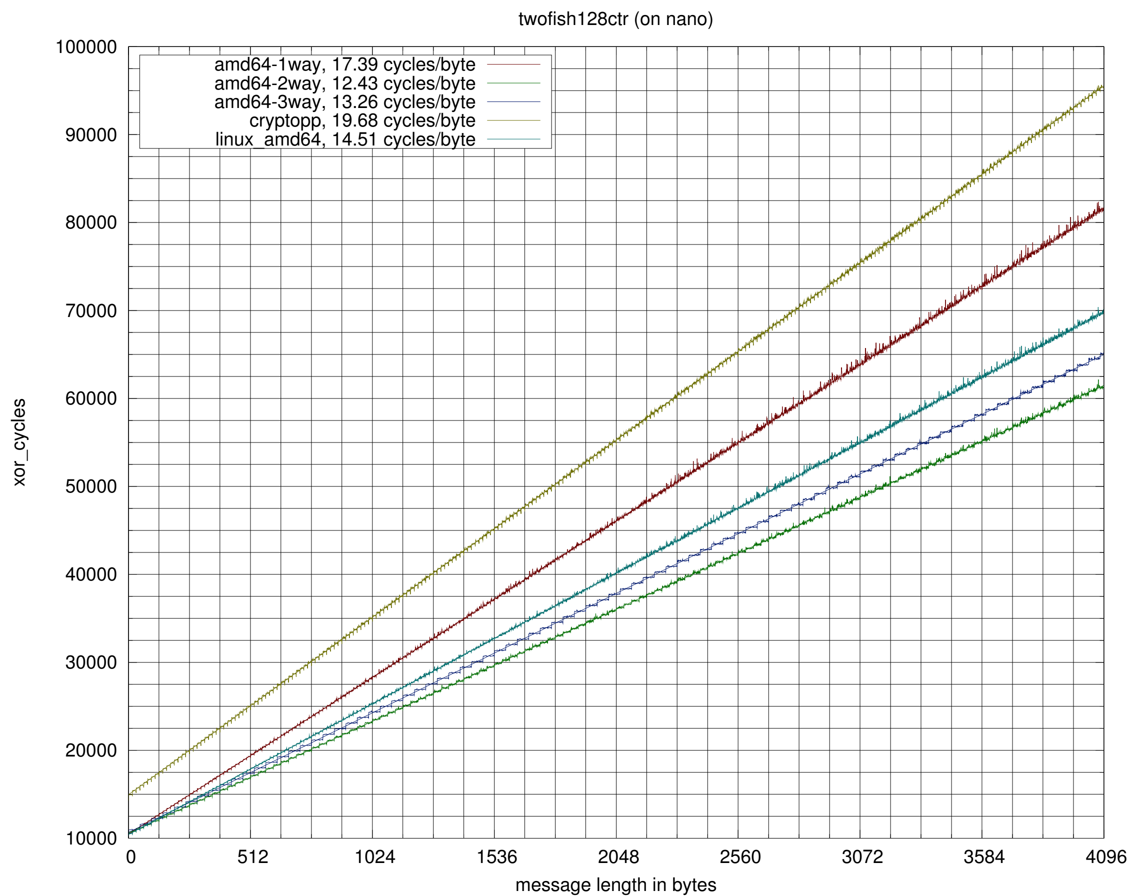
This section contains additional graphs that were generated from the raw measurement data in this study. Graphs are grouped by used computers, starting with `nano` and continuing with `homura`, `fate` and `railgun`. For processor details of each computer see Table 2.

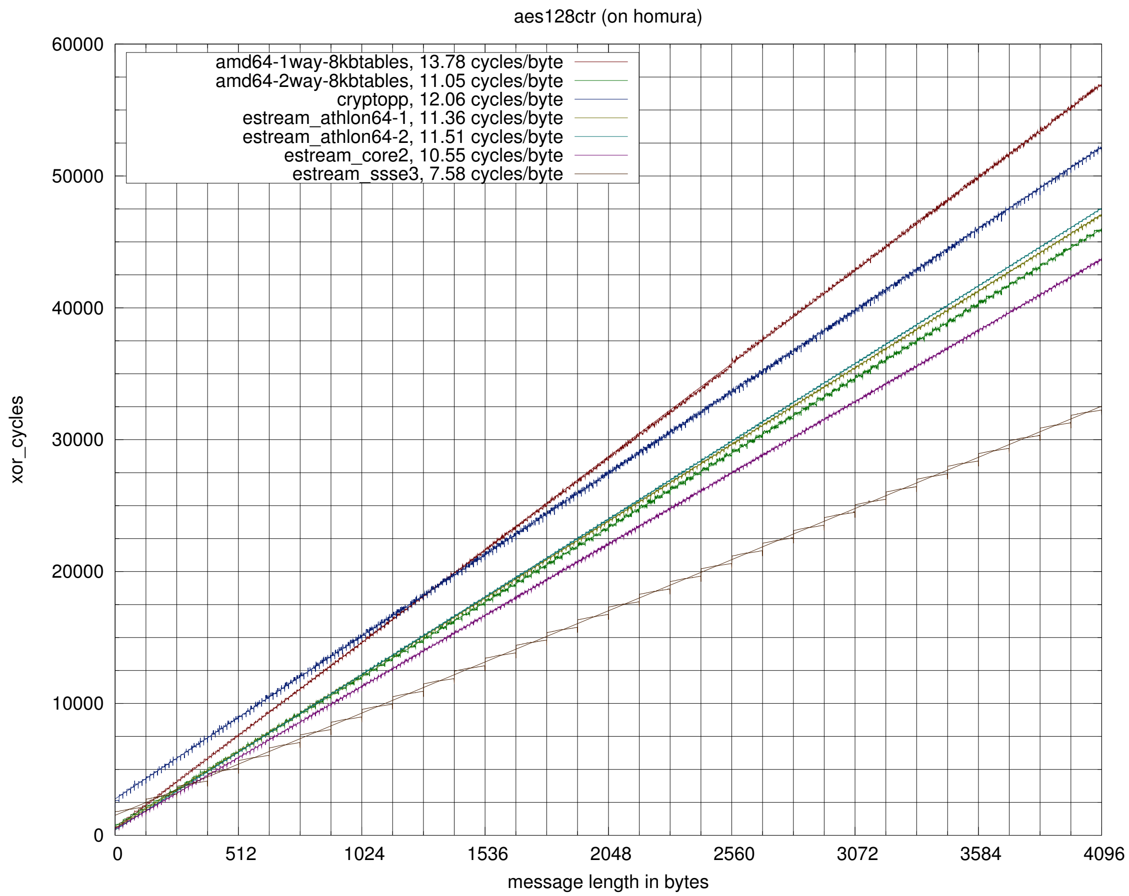


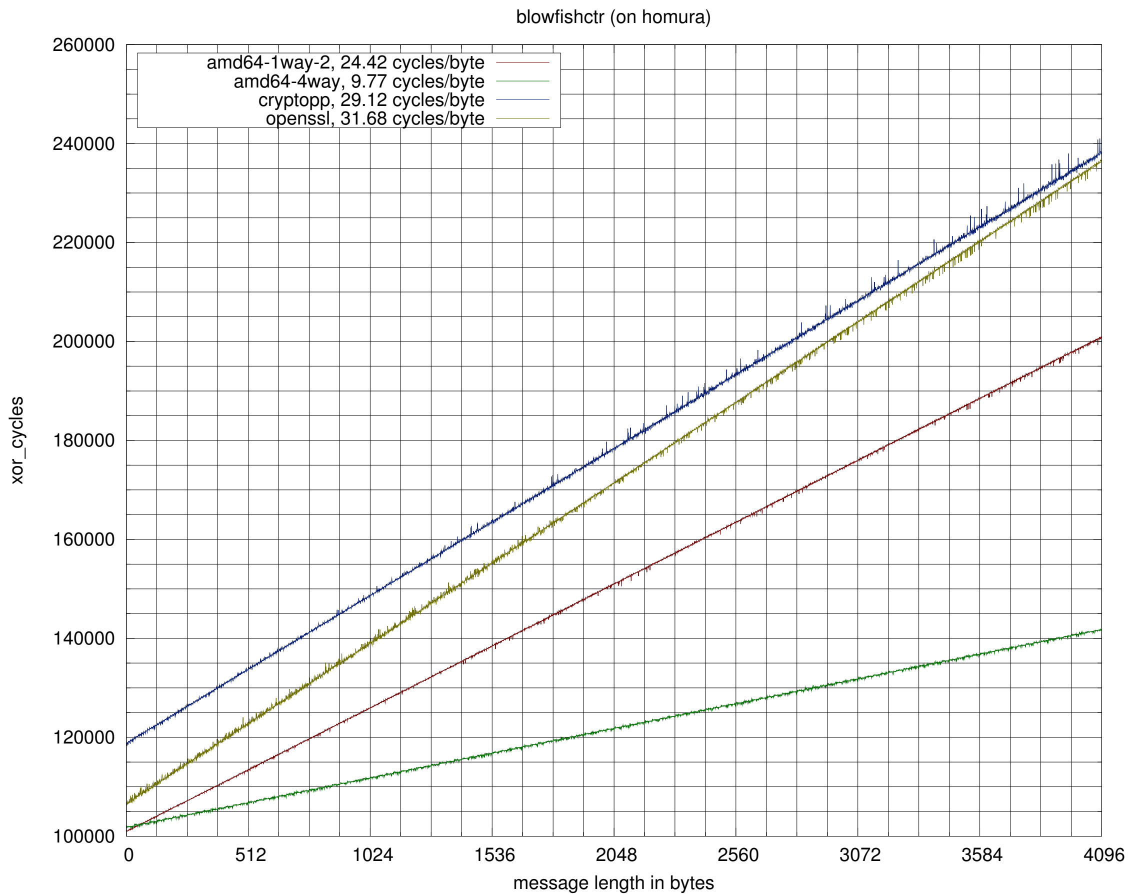

blowfishctr (on nano)
amd64-1way-2, 20.49 cycles/byte
amd64-4way, 8.80 cycles/byte
cryptopp, 24.80 cycles/byte
openssl, 27.74 cycles/byte
xor_cycles
message length in bytes
220000
200000
180000
160000
140000
120000
100000
80000
0
512
1024
1536
2048
2560
3072
3584
4096

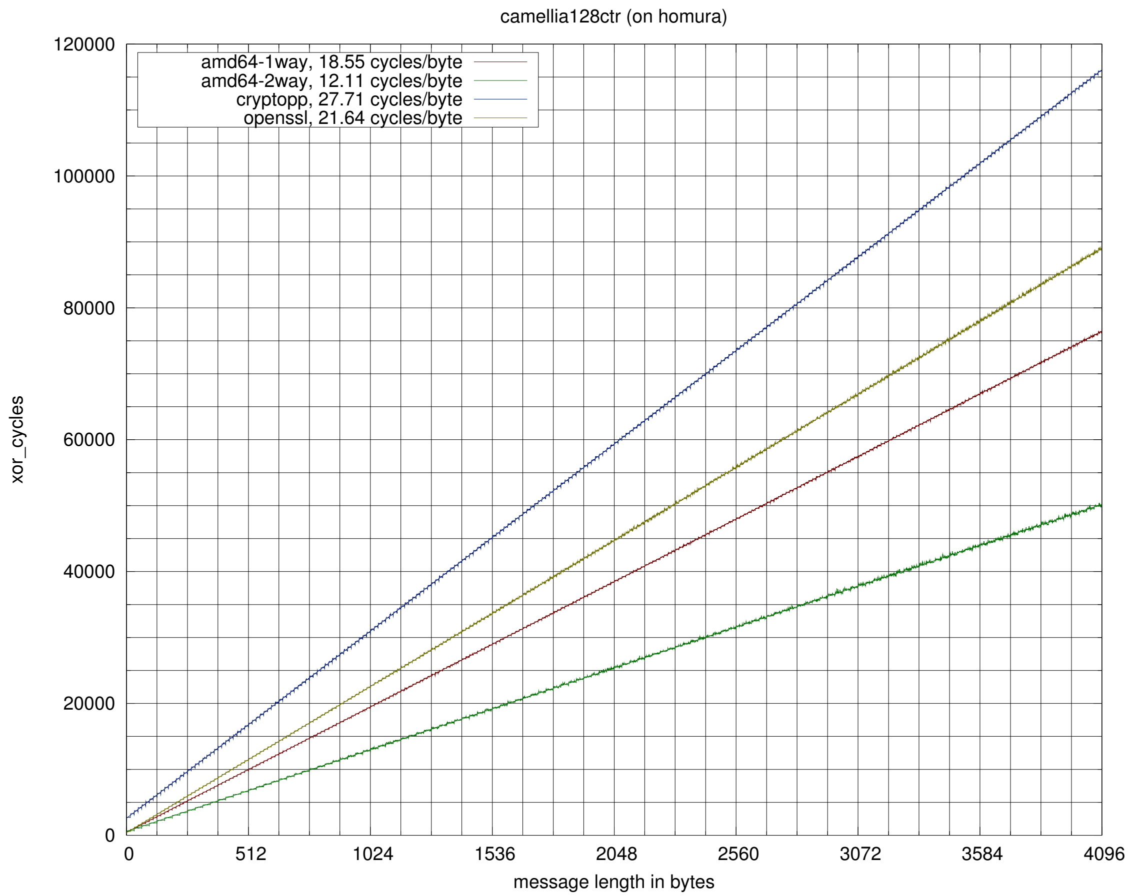

camellia128ctr (on nano)
amd64-1way, 13.74 cycles/byte
amd64-2way, 10.38 cycles/byte
cryptopp, 25.76 cycles/byte
openssl, 17.43 cycles/byte
xor_cycles
message length in bytes
120000
100000
80000
60000
40000
20000
0
0
512
1024
1536
2048
2560
3072
3584
4096

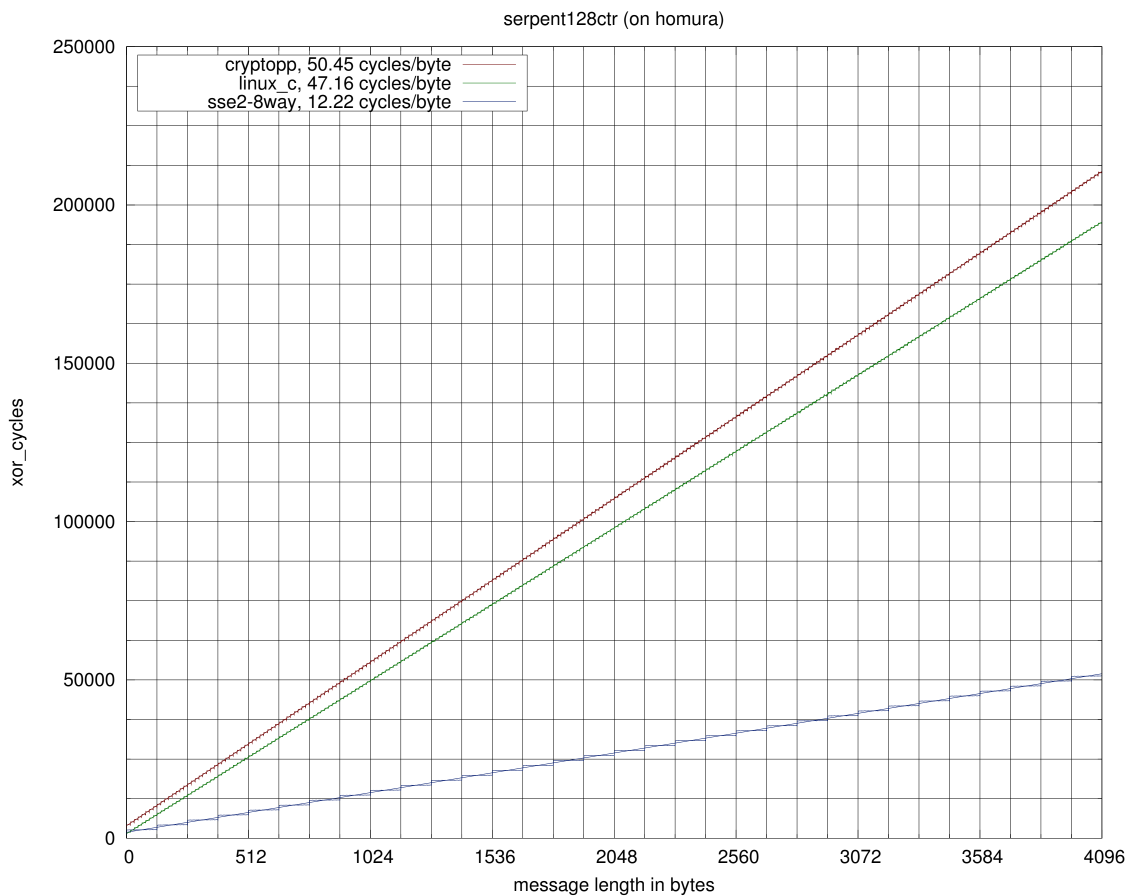

serpent128ctr (on nano)
cryptopp, 39.38 cycles/byte
linux_c, 35.14 cycles/byte
sse2-8way, 30.73 cycles/byte
xor_cycles
message length in bytes
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
512
1024
1536
2048
2560
3072
3584
4096

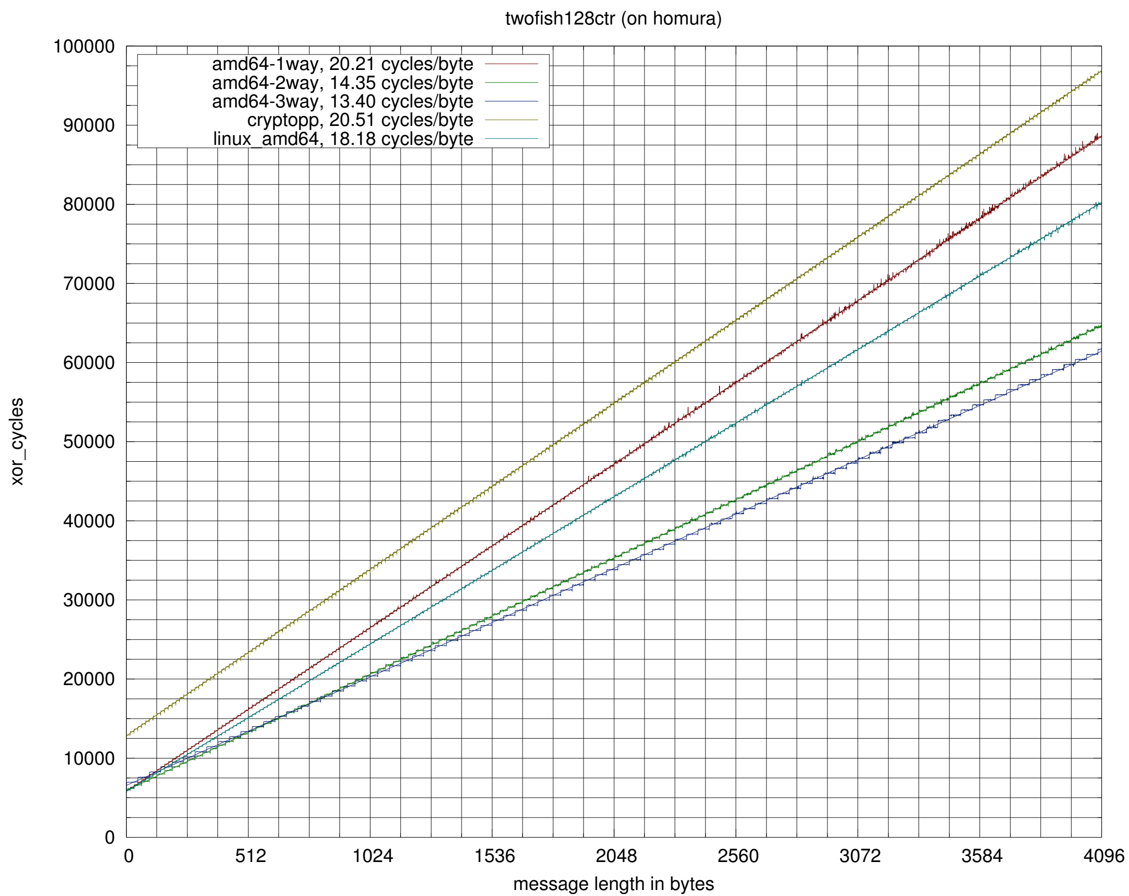

twofish128ctr (on nano)
amd64-1way, 17.39 cycles/byte
amd64-2way, 12.43 cycles/byte
amd64-3way, 13.26 cycles/byte
cryptopp, 19.68 cycles/byte
linux_amd64, 14.51 cycles/byte
xor_cycles
message length in bytes
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
0
512
1024
1536
2048
2560
3072
3584
4096

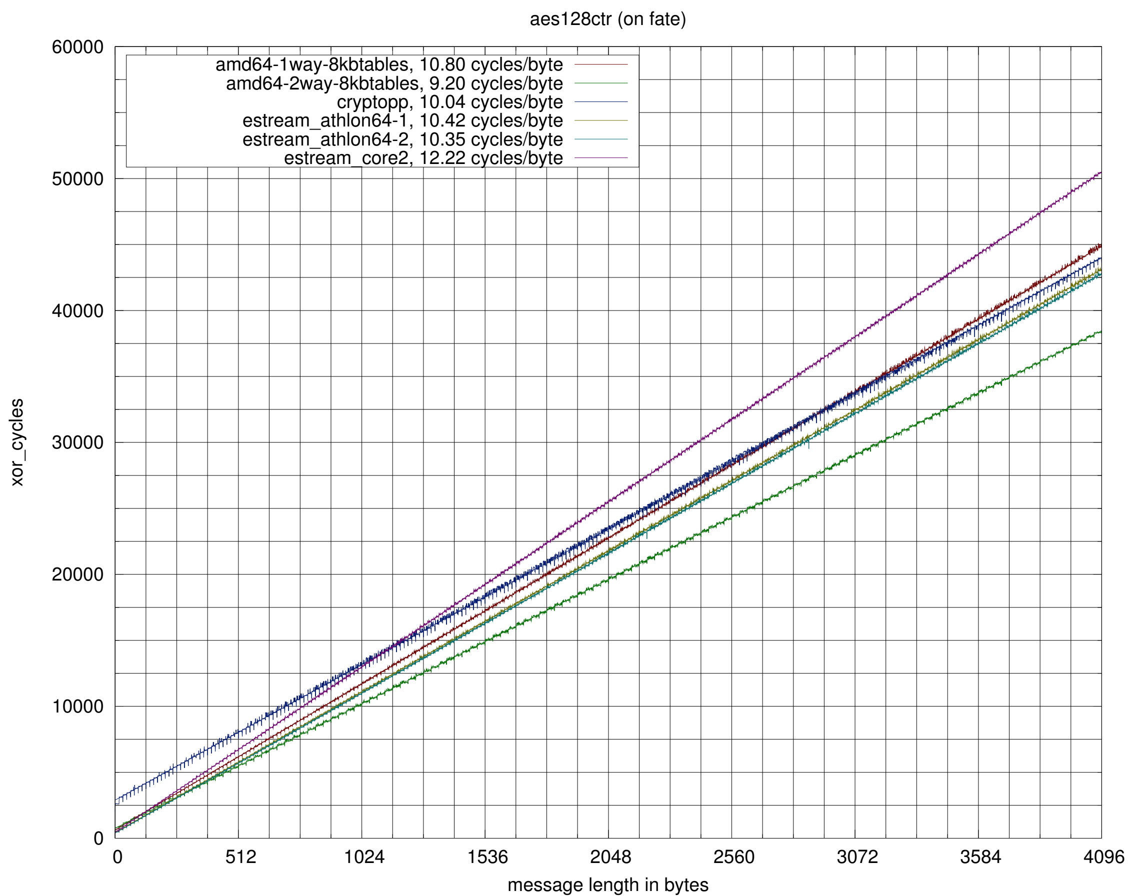

aes128ctr (on homura)
amd64-1way-8kbtables, 13.78 cycles/byte
amd64-2way-8kbtables, 11.05 cycles/byte
cryptopp, 12.06 cycles/byte
estream_athlon64-1, 11.36 cycles/byte
estream_athlon64-2, 11.51 cycles/byte
estream_core2, 10.55 cycles/byte
estream_ssse3, 7.58 cycles/byte
xor_cycles
message length in bytes
0
10000
20000
30000
40000
50000
60000
512
1024
1536
2048
2560
3072
3584
4096

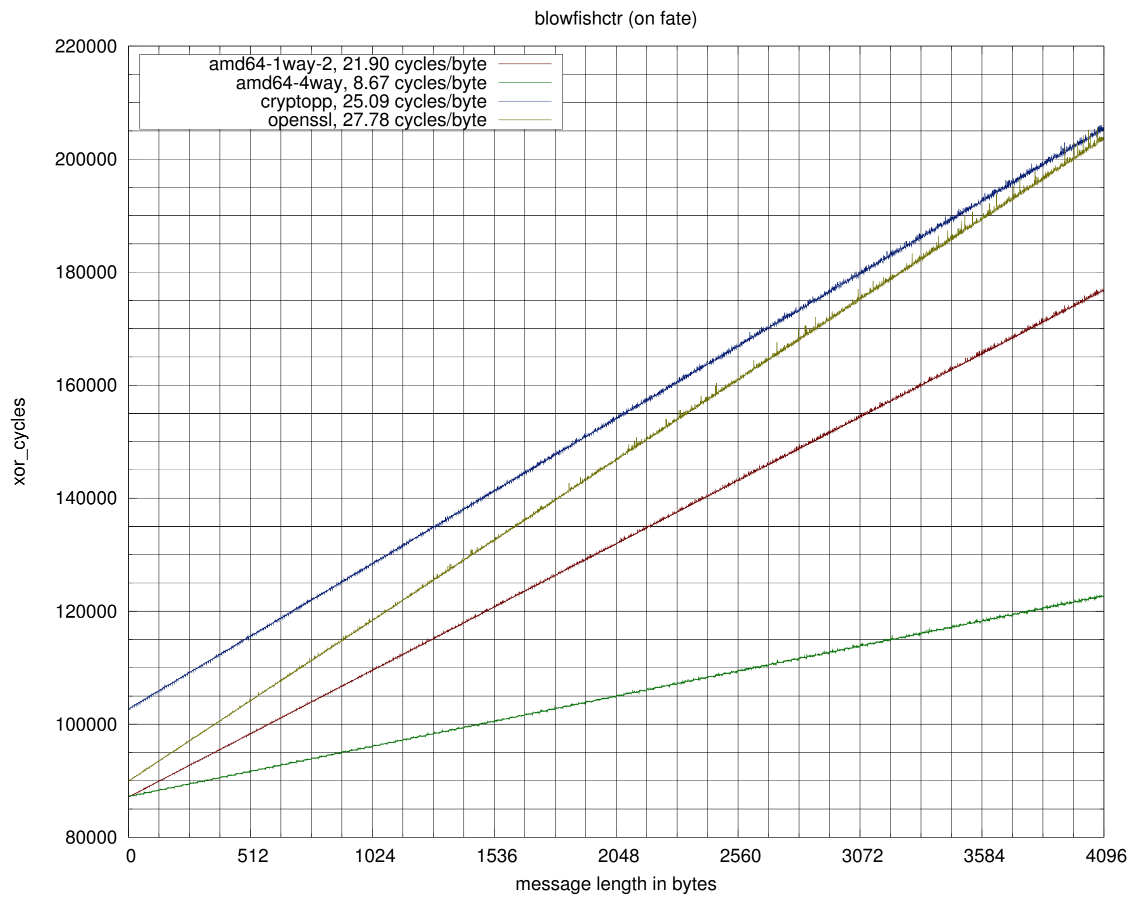

blowfishctr (on homura)
amd64-1way-2, 24.42 cycles/byte
amd64-4way, 9.77 cycles/byte
cryptopp, 29.12 cycles/byte
openssl, 31.68 cycles/byte
xor_cycles
message length in bytes
100000
120000
140000
160000
180000
200000
220000
240000
260000
0
512
1024
1536
2048
2560
3072
3584
4096

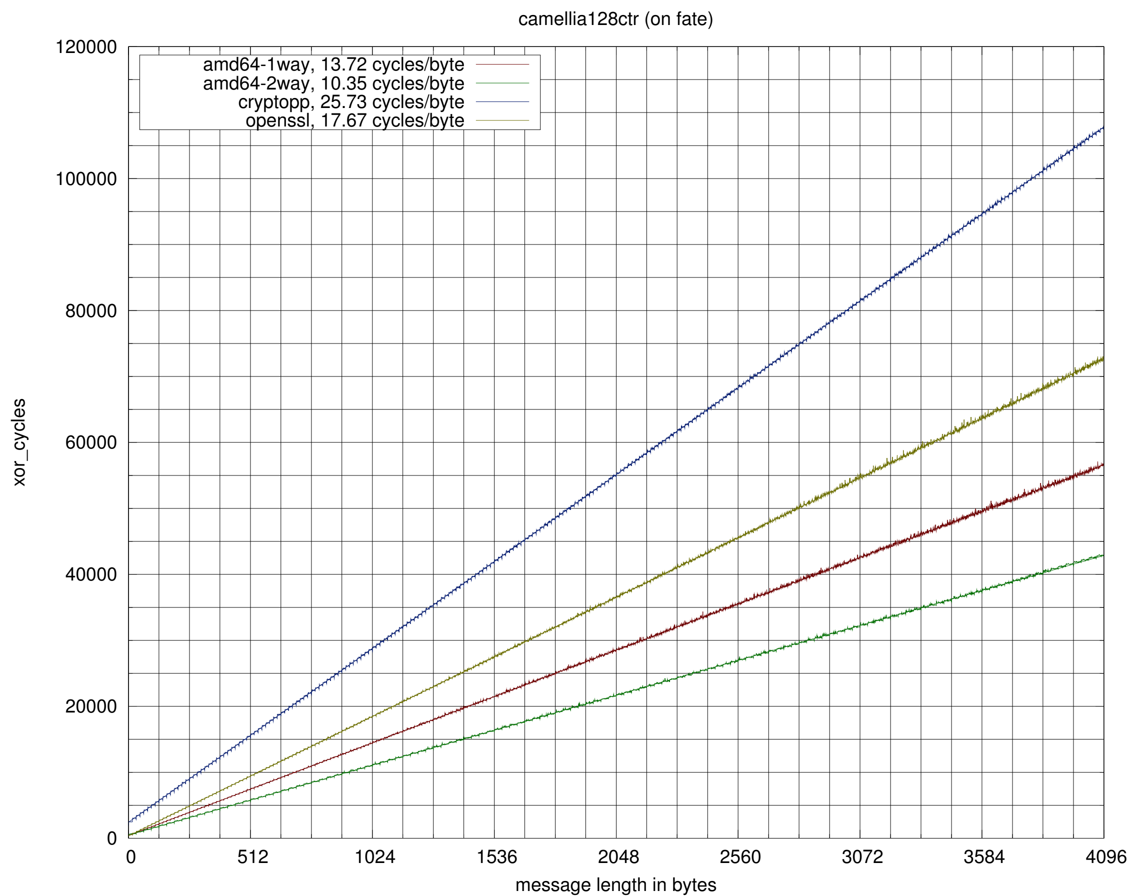

camellia128ctr (on homura)
amd64-1way, 18.55 cycles/byte
amd64-2way, 12.11 cycles/byte
cryptopp, 27.71 cycles/byte
openssl, 21.64 cycles/byte
xor_cycles
message length in bytes
0
20000
40000
60000
80000
100000
120000
0
512
1024
1536
2048
2560
3072
3584
4096

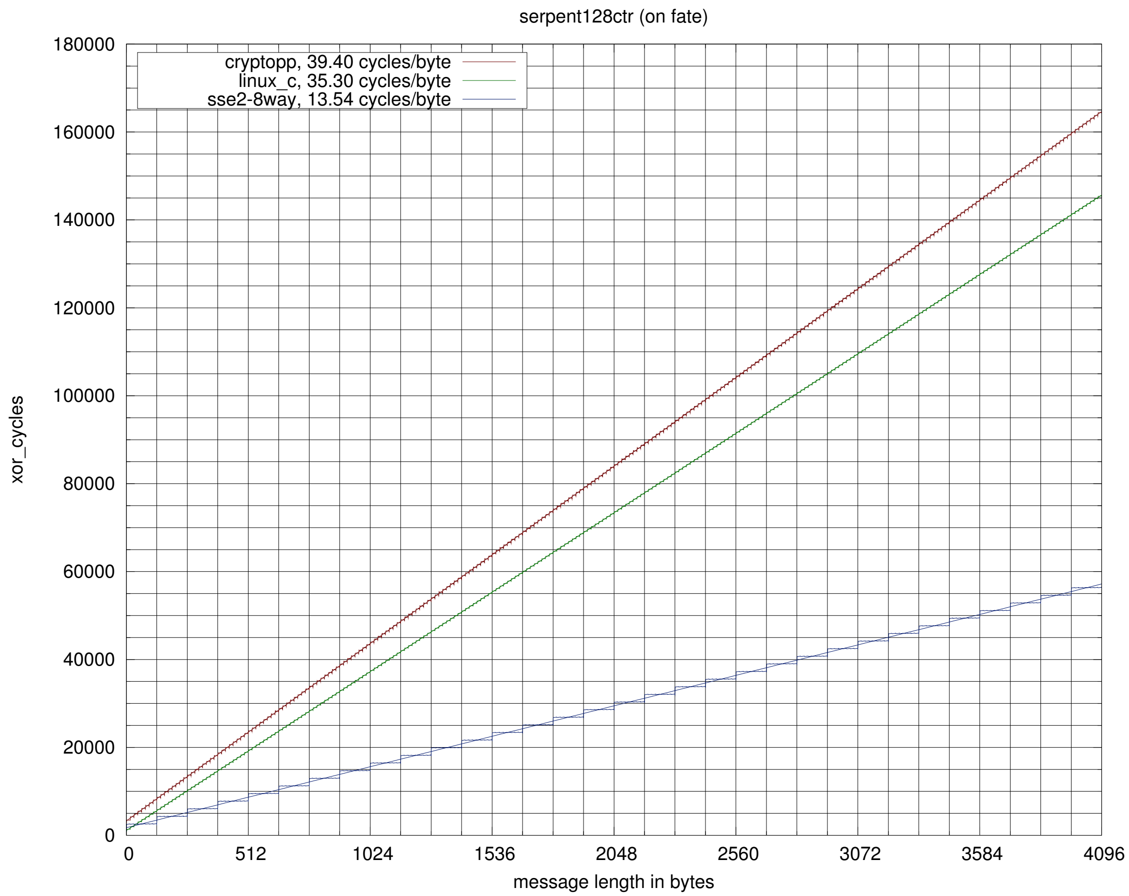

serpent128ctr (on homura)
cryptopp, 50.45 cycles/byte
linux_c, 47.16 cycles/byte
sse2-8way, 12.22 cycles/byte
xor_cycles
message length in bytes
0
50000
100000
150000
200000
250000
0
512
1024
1536
2048
2560
3072
3584
4096

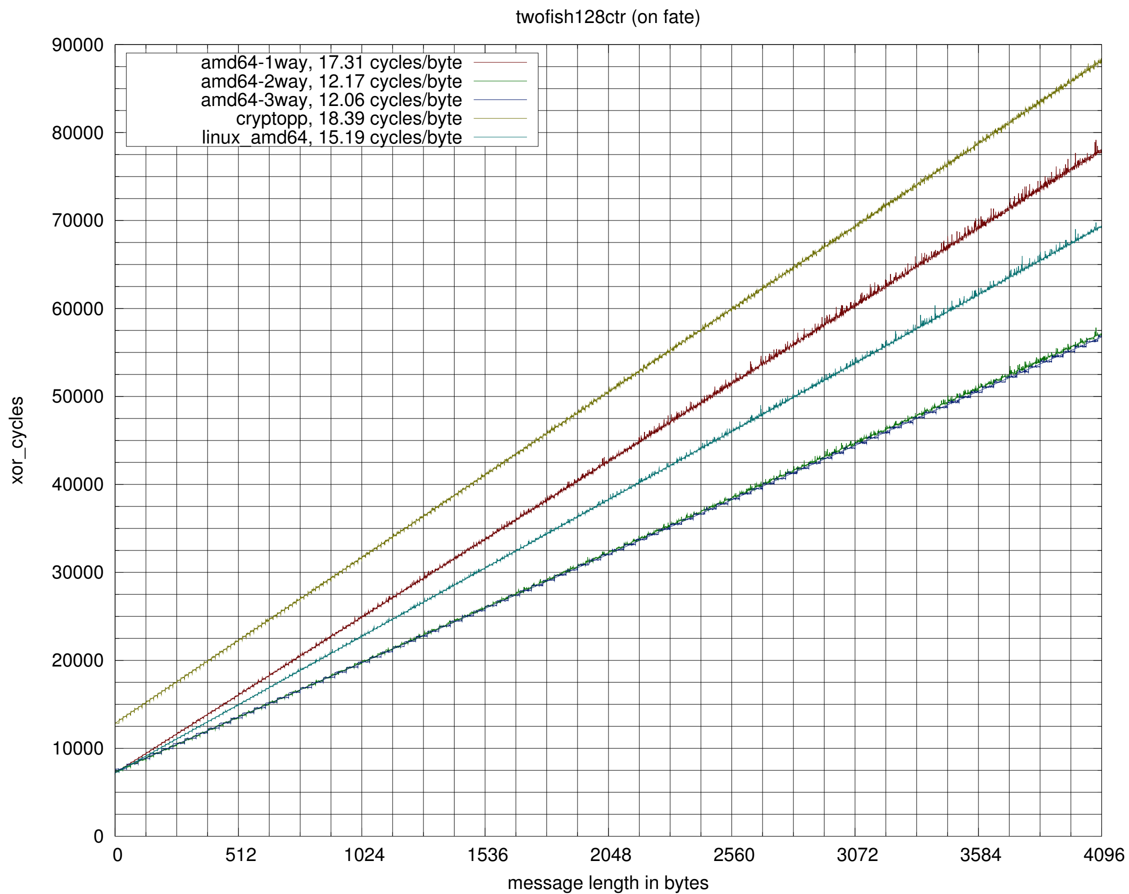

twofish128ctr (on homura)
amd64-1way, 20.21 cycles/byte
amd64-2way, 14.35 cycles/byte
amd64-3way, 13.40 cycles/byte
cryptopp, 20.51 cycles/byte
linux_amd64, 18.18 cycles/byte
xor_cycles
message length in bytes

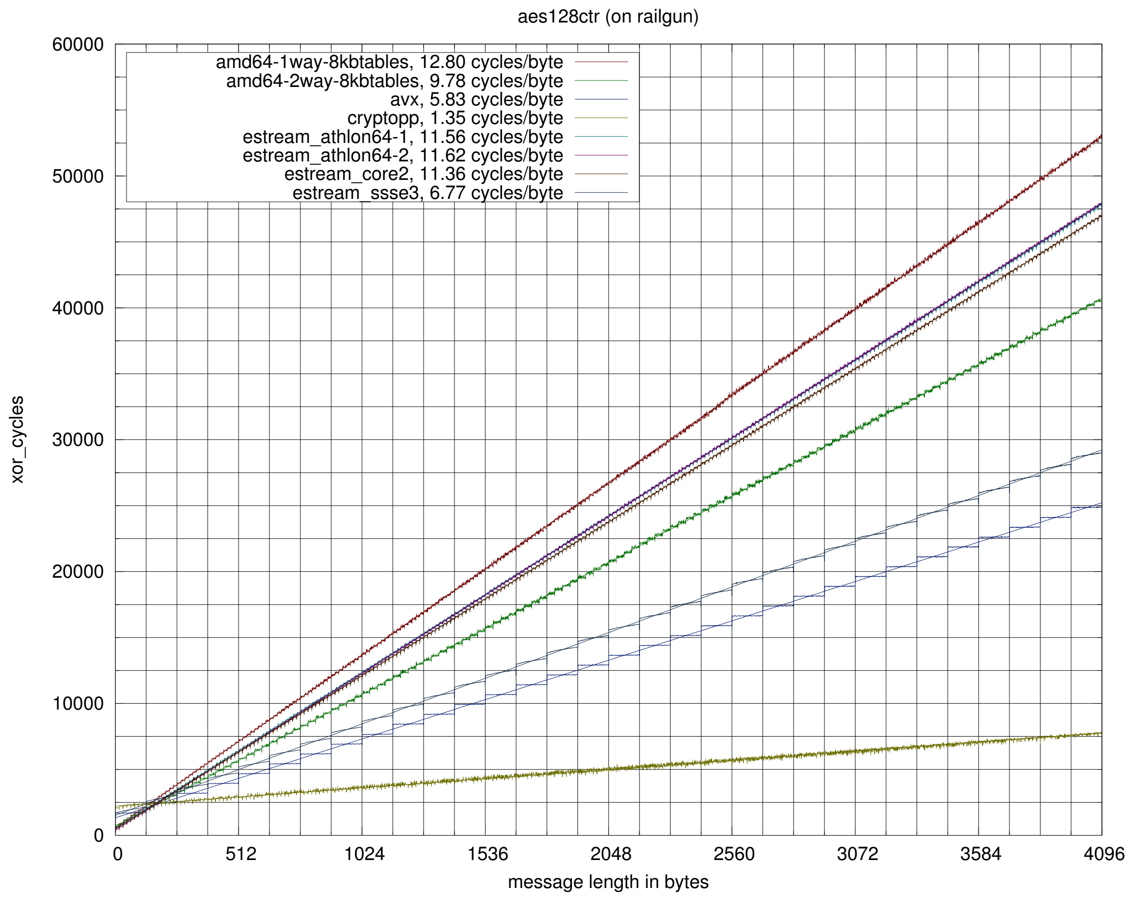

aes128ctr (on fate)
amd64-1way-8kbtables, 10.80 cycles/byte
amd64-2way-8kbtables, 9.20 cycles/byte
cryptopp, 10.04 cycles/byte
estream_athlon64-1, 10.42 cycles/byte
estream_athlon64-2, 10.35 cycles/byte
estream_core2, 12.22 cycles/byte
xor_cycles
message length in bytes

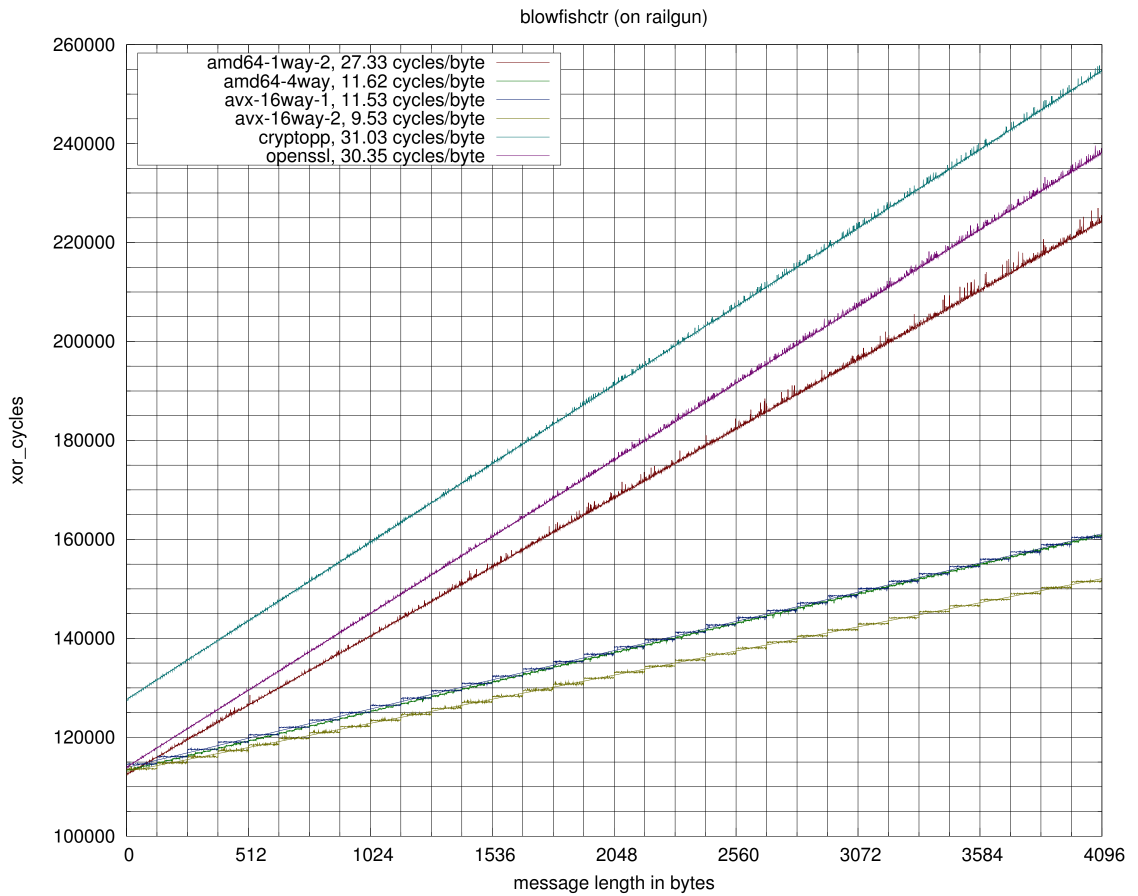

blowfishctr (on fate)
amd64-1way-2, 21.90 cycles/byte
amd64-4way, 8.67 cycles/byte
cryptopp, 25.09 cycles/byte
openssl, 27.78 cycles/byte
xor_cycles
220000
200000
180000
160000
140000
120000
100000
80000
0
512
1024
1536
2048
2560
3072
3584
4096
message length in bytes

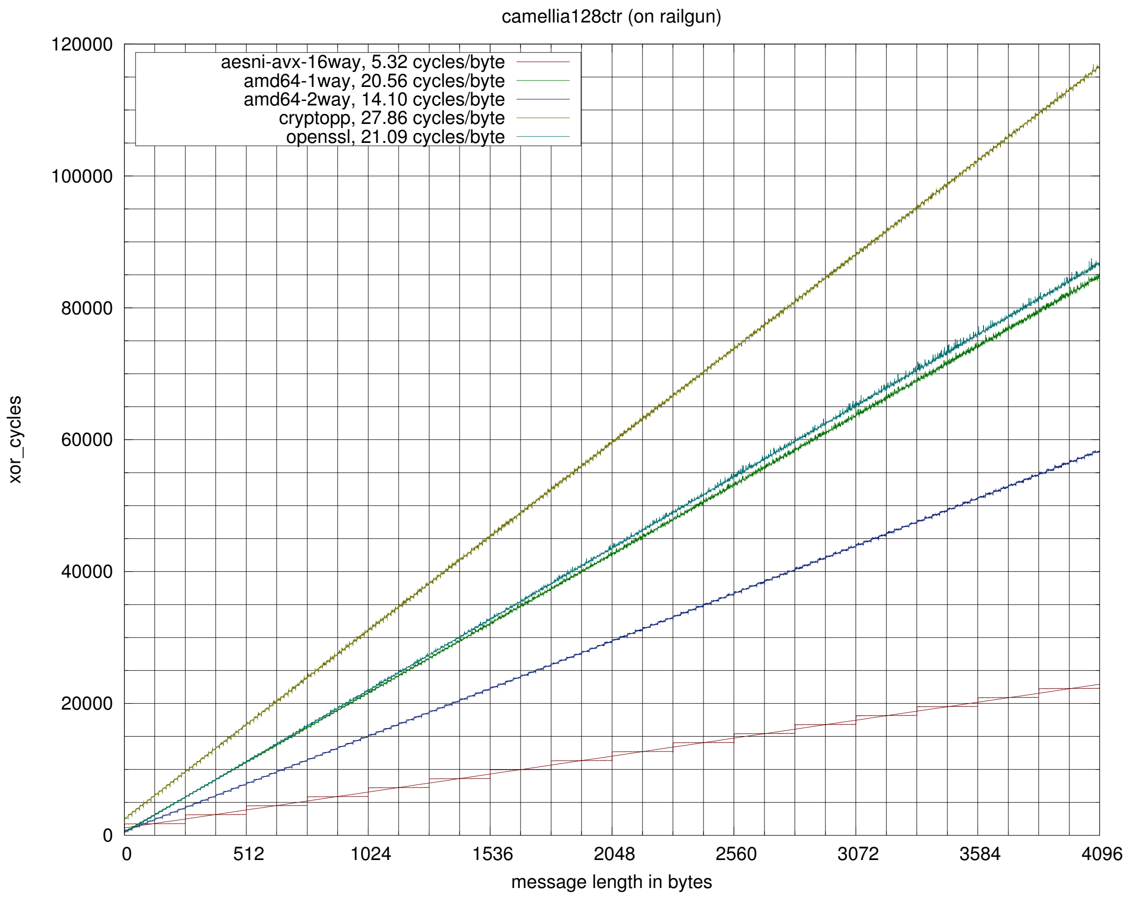

camellia128ctr (on fate)
amd64-1way, 13.72 cycles/byte
amd64-2way, 10.35 cycles/byte
cryptopp, 25.73 cycles/byte
openssl, 17.67 cycles/byte
xor_cycles
120000
100000
80000
60000
40000
20000
0
0
512
1024
1536
2048
2560
3072
3584
4096
message length in bytes

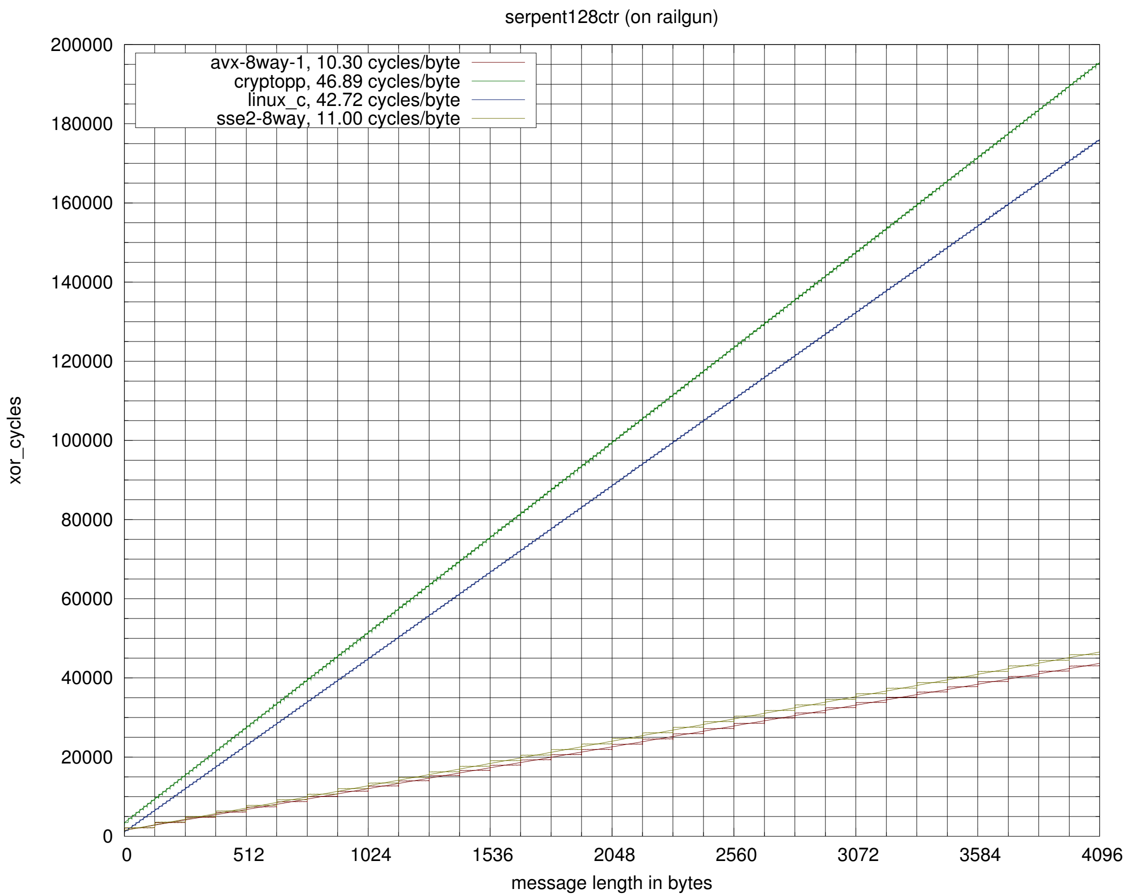

serpent128ctr (on fate)
cryptopp, 39.40 cycles/byte
linux_c, 35.30 cycles/byte
sse2-8way, 13.54 cycles/byte
xor_cycles
message length in bytes
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
0
512
1024
1536
2048
2560
3072
3584
4096

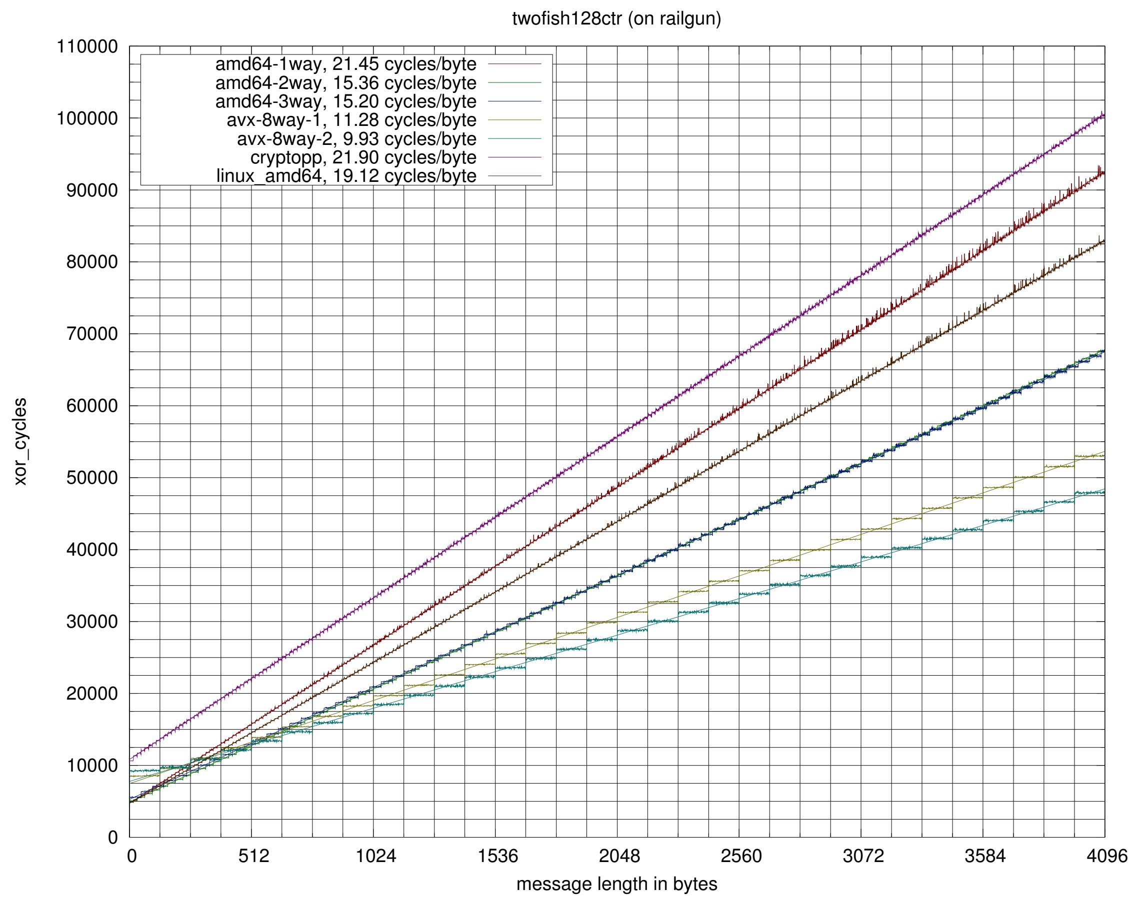

twofish128ctr (on fate)
amd64-1way, 17.31 cycles/byte
amd64-2way, 12.17 cycles/byte
amd64-3way, 12.06 cycles/byte
cryptopp, 18.39 cycles/byte
linux_amd64, 15.19 cycles/byte
xor_cycles
message length in bytes
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
0
512
1024
1536
2048
2560
3072
3584
4096


aes128ctr (on railgun)
amd64-1way-8kbtables, 12.80 cycles/byte
amd64-2way-8kbtables, 9.78 cycles/byte
avx, 5.83 cycles/byte
cryptopp, 1.35 cycles/byte
estream_athlon64-1, 11.56 cycles/byte
estream_athlon64-2, 11.62 cycles/byte
estream_core2, 11.36 cycles/byte
estream_ssse3, 6.77 cycles/byte
xor_cycles
message length in bytes
0
10000
20000
30000
40000
50000
60000
512
1024
1536
2048
2560
3072
3584
4096


blowfishctr (on railgun)
amd64-1way-2, 27.33 cycles/byte
amd64-4way, 11.62 cycles/byte
avx-16way-1, 11.53 cycles/byte
avx-16way-2, 9.53 cycles/byte
cryptopp, 31.03 cycles/byte
openssl, 30.35 cycles/byte
xor_cycles
message length in bytes
100000
120000
140000
160000
180000
200000
220000
240000
260000
0
512
1024
1536
2048
2560
3072
3584
4096


camellia128ctr (on railgun)
aesni-avx-16way, 5.32 cycles/byte
amd64-1way, 20.56 cycles/byte
amd64-2way, 14.10 cycles/byte
cryptopp, 27.86 cycles/byte
openssl, 21.09 cycles/byte
xor_cycles
message length in bytes
0
20000
40000
60000
80000
100000
120000
0
512
1024
1536
2048
2560
3072
3584
4096


serpent128ctr (on railgun)
avx-8way-1, 10.30 cycles/byte
cryptopp, 46.89 cycles/byte
linux_c, 42.72 cycles/byte
sse2-8way, 11.00 cycles/byte
xor_cycles
message length in bytes
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
200000
0
512
1024
1536
2048
2560
3072
3584
4096


twofish128ctr (on railgun)
amd64-1way, 21.45 cycles/byte
amd64-2way, 15.36 cycles/byte
amd64-3way, 15.20 cycles/byte
avx-8way-1, 11.28 cycles/byte
avx-8way-2, 9.93 cycles/byte
cryptopp, 21.90 cycles/byte
linux_amd64, 19.12 cycles/byte
xor_cycles
110000
100000
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
0
512
1024
1536
2048
2560
3072
3584
4096
message length in bytes