KATHOLIEKE UNIVERSITEIT
LEUVEN

Faculteit
Ingenieurswetenschappen

The design and efficient software implementation of S-boxes

Markus Ullrich

Thesis submitted for the degree
of Master of Engineering:
Electrical Engineering

Promotors:
Prof. dr. ir. Bart Preneel
Prof. dr. ir. Vincent Rijmen

Assessors:
Dr. ir. Christophe De Cannière
Prof. dr. ir. Luc Van Eycken

Counsellors:
Dr. ir. Christophe De Cannière
Dr. ir. Sebastiaan Indesteege
M.S. Özgül Küçük
ir. Nicky Mouha

Academic year 2009 – 2010

© Copyright K.U.Leuven

Without written permission of the promotors and the authors it is forbidden to reproduce or adapt in any form or by any means any part of this publication. Requests for obtaining the right to reproduce or utilize parts of this publication should be addressed to Departement Elektrotechniek, Kasteelpark Arenberg 10 postbus 2440, B-3001 Heverlee, +32-16-32 11 30 or by email info@esat.kuleuven.be.

A written permission of the promotor is also required to use the methods, products, schematics and programs described in this work for industrial or commercial use, and for submitting this publication in scientific contests.

Preface

When I arrived in Leuven for the first time almost two years ago, I would never have guessed how fast time can fly. Now, the thesis is finished and my time as a student is nearing its end. The thesis is a worthy end of my career as a student.

When I look back on the last year, I would like to thank everyone who helped me on my way. First of all, I would like to thank my promotors Prof. Bart Preneel and Prof. Vincent Rijmen. I would also like to show my gratitude to my daily supervisors, Christophe De Cannière, Sebastiaan Indesteege, Özgül Küçük and Nicky Mouha. Their support for technical and practical issues was a great help.

Then I would like to thank my family. I am very grateful for my parent's help which made it possible for me to study abroad. I also appreciate my brother Christian for reminding me that a healthy mind requires a healthy body. I am also thankful to my father and Monique for helping with proofreading the thesis. I like to thank my girlfriend Sarah for all the support she gave me and also for the positive distraction from the work. And of course I should not forget to appreciate her great patience which was probably needed towards the deadline.

I also like to thank my fellow students. I made good friends and we had good times. I wish them all the best with their theses and their future.

The community of free software developers deserve my special thank. Not only most of the software tools used for developing or writing the thesis, but also the some of the libraries used in the software tool designed in this work were available under free licenses.

Finally, I also would like to express my gratitude to my assessors, Prof. Luc Van Eycken and Christophe De Cannière, for their efforts to evaluate my thesis.

Markus Ullrich

Contents

Abstract

Recently, more and more secure applications are run on embedded and mobile platforms. This increases the need for highly efficient and lightweight primitives. Substitution boxes are an essential part of many cryptographic primitives and can be found in a lot of applications of symmetric cryptography. This thesis concerns the efficient bit sliced implementation of 4×4-bit s-boxes. In order to find efficient implementations, all possible implementations in form of instruction sequences have been searched, and analysed for their properties as an s-box.

As a method we introduce a combination of algorithms in order to find the efficient implementations. The s-boxes are classified according to non-linear properties and it is searched for the most efficient one per class, the representative of the class. We discuss several methods for speeding up the search. The main focus is on the algorithmic level and on the parallelisation. On the one hand, we show methods that allow to significantly reduce the branching factor of the search tree by using equivalences instead of equalities as a basis for sorting out redundant nodes. On the other hand, we discuss two parallelisation approaches for two different architectures, a multi core computer and a computer cluster. This results in the design of a software tool, which is used to perform the search for the most efficient implementations.

As result we present a list of all optimal implementations found so far, and analyse the relationship between implementation cost and non-linear properties. We discuss specific properties of the representatives that are invariant within the class. We also investigate some characteristics of the implementations such as the instructions needed to reach certain properties. We look into existing primitives using 4-bit s-boxes, e.g., Serpent, Luffa and Noekeon. Their s-boxes are analysed and discussed in comparison with the representatives of the classes. None of the analysed primitives uses a representative.

Finally, we present a new design methodology. The suggested approach allows a designer to benefit from the results of this work when designing highly efficient ciphers.

List of Figures

List of Tables

List of Abbreviations

Abbreviations

ALU	Arithmetic Logic Unit
BFS	Breadth First Search
DFS	Depth First Search
ID-DFS	Iterative Deepening Depth First Search
MDP	Maximum Differential Probability
MLP	Maximum Linear Probability
MOV	Move, name of the copy instruction in many microprocessors
S-BOX	Substitution Box
SPN	Substitution-Permutation-Network also called SP-Network

Chapter 1

Introduction

The history of substitution boxes reaches back to 1949 when Claude Shannon proposed the principles of confusion and diffusion [28]. The method of confusion is to make the relation between the simple statistics of the ciphertext and the simple description of the key a very complex and involved one. Diffusion is a similar method but applies to the relationship between plaintext and ciphertext. This principle is still widely used in many types of symmetric cryptographic primitives such as block ciphers, stream ciphers and hash functions. Many applications such as those in mobile devices or RFID tags require very lightweight and efficient primitives. The major problem with some of the widely used primitives is, that their design strategy was focused mainly on security and that the efficiency of implementations was not always a major goal. This work copes with the problem of efficient implementations of s-boxes as an essential part of cryptographic primitives. S-boxes are non-linear functions applied on small amounts of data, typically of sizes 4 to 8 bit. The basic idea of the approach of this thesis is based on fact that s-boxes can be classified. The search proceeds in well-chosen order through all instruction sequences such that the most efficient implementations are found first. It follows that for each class the best implementation is found first. The work includes the design of a software tool, which actually performs the search for efficient implementations. Below, an overview of the constraints of this thesis is given, the details of which are found in the according chapters:

- The size of the s-boxes has been limited to 4×4-bits. Most of the s-boxes applied in practice have sizes, which are a power of 2, for efficiency reasons. The next larger size, 8×8-bits, is not suited for this type of brute force search because of the immense search space. Even 5×5-bit s-boxes are expected to extend a realistic search space.

- We only look at invertible s-boxes. In principle the approach would work with non-invertible s-boxes as well. But we decided to limit our scope to non-invertible ones because of the different applications. Ciphers based on Substitution-Permutation-networks require invertibility. Some other primitives do not require this property, e.g., Feistel ciphers. Another argument is the

efficiency. The search space for non-invertible s-boxes is larger and some of the used algorithms, e.g., the equivalence algorithm, are less efficient for non-invertible s-boxes.

- Affine equivalence is used as classification criteria for the s-boxes. S-boxes that are affine equivalent have the same properties with respect to linear and differential cryptanalysis. This classification thus reduces s-boxes to their essence with respect to those properties. Details in section 2.2 and 3.2

- The search has been designed for bit sliced software implementations assuming a simple instruction set. The basic approach can be extended to other architectures and hardware implementations. This configuration helps to limit the search space for efficiency reasons. Details in section 3.1.

In chapter 2, the basics about s-box properties, the classification and design strategies of s-boxes are presented. In chapter 3, the different algorithms that are used in the tool are explained. Further, the restrictions of the implementation are introduced. In chapter 4, we explain implementation specific changes that have been applied for efficiency and reliability. The chapter also deals with the parallelisation of the program. Chapter 5 presents the results obtained by applying the software tool developed in this thesis. The results are analysed and compared with those found in the literature. In the last chapter, chapter 6, we conclude the work with a summary and suggestions for future research.

Chapter 2

Theoretical background

In the first section of this chapter the most important properties of s-boxes in reference to cryptanalysis will be introduced. In section 2.2 the properties are applied to classify s-boxes. Section 2.3 copes with a number of the different design methodologies for s-boxes that have been proposed. The methodologies are compared with the new approach of this thesis. In section 2.4 an explanation about bit sliced implementations is given.

2.1 Properties of s-boxes

2.1.1 Differential and linear cryptanalysis

Linear and differential cryptanalysis are considered as being some of the most important techniques for symmetric cryptanalysis [5, 16, 18].

Initially published in the late 80s, differential cryptanalysis gained attention in 1991, when a modified version successfully attacked the full 16-round DES. The basic idea of differential cryptanalysis is to find correlations between differences of input blocks and the differences of the corresponding output blocks.

Before explaining differential properties of s-boxes, some basic principles of ciphers have to be introduced. Ciphers, such as DES or AES, consist of three basic building blocks: the diffusion layer, confusion layer and the key addition. The key addition is, due to the properties of the addition in $GF(2)$, transparent to differences. The diffusion layer is a linear operation, which can influence the differences, but always in a predictable way. This leaves the non-linear s-box to prevent a high correlation between the input and the output differences. From an ideal s-box one would expect that, for any input difference, all output differences are equally probable. In reality this is mathematically not possible. First, $\Delta_i = 0$ implies that the compared input values are equal. Consequently, they will also be mapped to the same values and result in an output difference $\Delta_o = 0$. Second, every difference occurs twice because of the commutativity of the addition in $GF(2)$. This means that, when counting the occurrences of difference pairs, it will always result in even numbers. Table 2.1 shows an example of an s-box with its difference distribution table. It contains the probabilities that a certain input difference Δ_i will result in a certain output

TABLE 2.1: Example s-box with difference distribution table

in	out	Δ_i \ Δ_o	0	1	2	3	4	5	6	7
0	0	0	1	0	0	0	0	0	0	0
1	2	1	0	0	1/2	0	1/2	0	0	0
2	1	2	0	1/2	0	0	0	0	0	1/2
3	5	3	0	0	0	1/2	0	1/2	0	0
4	6	4	0	0	0	0	0	0	1	0
5	4	5	0	0	1/2	0	1/2	0	0	0
6	7	6	0	1/2	0	0	0	0	0	1/2
7	3	7	0	0	0	1/2	0	1/2	0	0

TABLE 2.2: Differential characteristic of the s-box table 2.1

p	1/4	1/2	3/4	1
#	0	12	0	2

difference Δ_o. Attackers are interested in pairs with a high probability, such as $\Delta_i = 4$, which always leads to an output difference of $\Delta_o = 6$. The s-box can then be characterised by the distribution of the probabilities such as in table 2.2.

A few years after differential cryptanalysis, linear cryptanalysis has been developed. Linear cryptanalysis investigates linear relationships between the plaintext and the ciphertext, or parts of them. As for differential cryptanalysis, the non-linear s-box is responsible for reducing the correlation between input and output bits. Binary masking functions (Γ) are used as notation to describe the selected bits. Those functions indicate for the input and the output which bits are take into account for the linear approximation. Formula 2.1 describes how the key can be approximated from the input of the round, X, and the output of the round, Y, using masking functions.

$$\Gamma_x^\top \cdot X \oplus \Gamma_y^\top \cdot Y = \Gamma_k^\top \cdot K \tag{2.1}$$

Equivalent to the difference distribution table, a linear approximation table can be created. The linear approximation table for the example s-box can be found in table 2.3. This table indicates the absolute correlations between the masked input and outputs. The table shows that, for example, the output bit 1 follows input bit 2 for all possible inputs. High correlations offer possibilities for attacks.

It is not sufficient to compare the security of s-boxes by comparing the averages of these properties. The extreme values of the tables are a very important property because attackers try to abuse the worst case. Those values are called maximum linear probability (MLP) and maximum differential probability (MDP). If an attacker finds linear of differential pairs that have a high probability he can estimate part of the cipher and attack the cipher round by round.

An example for an attack using linear and differential cryptanalysis is given in [13].

TABLE 2.3: Linear approximation table of the s-box from table 2.1

Γ_i \ Γ_o	0	1	2	3	4	5	6	7
0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1	0
2	0	1	0	0	0	0	0	0
3	0	0	0	0	0	0	0	1
4	0	0	1/2	1/2	1/2	1/2	0	0
5	0	0	1/2	1/2	1/2	1/2	0	0
6	0	0	1/2	1/2	1/2	1/2	0	0
7	0	0	1/2	1/2	1/2	1/2	0	0

TABLE 2.4: Linear characteristic of the s-box from table 2.1

$\lvert c \rvert$	1/2	1
#	16	4

2.1.2 Other s-box properties

In this section other properties of s-boxes will be introduced. This is only a selection of properties that have been chosen because of their importance for the design strategies of many primitives. These properties are of less importance for this thesis and will be explained more briefly.

2.1.2.1 Branch number

When talking about branch numbers, one has to clarify which definition of branch numbers is referred to. The first definition is the minimum sum of the input difference and output differences word weight. It is a measure of how many input and output words that will have to change minimally. For example, in a cipher with branch number 4, one input word change will change at least 3 output words. Important is that if more input words change the number of output words that have to change is lower. This definition is applicable for whole SP-networks. The non-linear s-boxes as well as the linear diffusion layer can influence on the branch number.

Definition 2.1 $B = min_{a,b \neq a}(w_w(a \oplus b) + w_w(F(a) \oplus F(b))$

Where w_w return the number of non-zero words, and F is the round function of the cipher [10]. It is a measure for how many words have changed.

The other definition, which is applicable for single s-boxes and therefore more suited for this thesis, defines the minimum amount of input and output bits that will be changed by the s-box.

Definition 2.2 $B = min_{a,b \neq a}(w_h(a \oplus b) + w_h(S(a) \oplus S(b))$

Where w_h is the Hamming weight and S is the non-linear function. The second definition is not commonly used. In literature often other terms are used, such as in PRESENT [6], where the s-box is chosen to satisfy the following condition:

$$\begin{aligned} &\forall \Delta_I \in \mathbb{F}_2^4 \neq 0 \text{ and } w_h(\Delta_I) = 1 \\ &\forall \Delta_O \in \mathbb{F}_2^4 \text{ and } w_h(\Delta_O) = 1 \\ &\{x \in \mathbb{F}_2^4 | S(x) + S(x + \Delta_I) = \Delta_O\} = \emptyset \end{aligned} \tag{2.2}$$

This is equivalent to $B > 2$, for the branch number according to definition 2.2.

Generally, it is valid, that the more s-boxes that are active in a cipher, the higher is the complexity of an attack. Both the linear layer and the s-box layer take part of in defining the branch number of an SP-network. If the linear layer consists of permutations only, as for example in PRESENT, it can only be guaranteed that multiple s-boxes get activated if the s-box has a branch number higher than 2 according to definition 2.2.

2.1.2.2 Fixed points

Some design strategies for cryptographic primitives require that the s-box has no fixed point, meaning that there is no value x such that $x = S(x)$. A fixed point is not necessarily a weakness. But there are applications, especially in combination with special linear layers or in some hash functions where fixed points are disadvantageous. The fixed point $S(0) = 0$ can even cause that the whole cipher maps 0 to 0 if the key is zero. This is also not necessarily a weakness. The KATAN family of block ciphers is an example of ciphers with this property. Even though the cipher is not using classical s-boxes, but is inspired by stream cipher design using LFSR (linear feedback shift registers) [19].

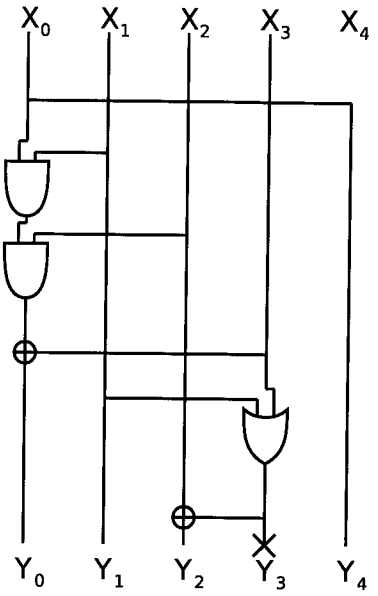
2.1.2.3 Algebraic degree

Every s-box can be described as a set of polynomials. Each polynomial gives one output bit in function of the input bits.

Figure 2.1 shows an example of an s-box with corresponding polynomials describing the output bits. The s-box output consists of $\{y_0, y_1, y_2, y_4\}$. The s-box in the example has large variations in the algebraic degree of the different outputs. While y_0 has a degree of 3, y_1 has a degree of only 1. The degree of the polynomial as well as the amount of different monomials are important properties for algebraic cryptanalysis [3, 12]. The maximum algebraic degree is limited by the s-box size and the implementation constraints. For an invertible 4×4-bit s-box it is limited to 3.

2.2 Classification of s-boxes

In the last section, we introduced different properties that can characterise s-boxes. Having the ability to characterise s-boxes, one can go one step further and classify them in classes with common properties. Before talking about classifications the



$$
\begin{aligned}
y_0 &= x_2 \cdot x_1 \cdot x_0 + x_3 \\
y_1 &= x_1 \\
y_2 &= x_2 + x_3 \cdot x_2 + x_3 + x_2 \\
y_3 &= x_3 \cdot x_2 + x_3 + x_2 \\
y_4 &= x_4 = x_0
\end{aligned}
$$

FIGURE 2.1: Example of the polynomial representation of the output bits of an s-box

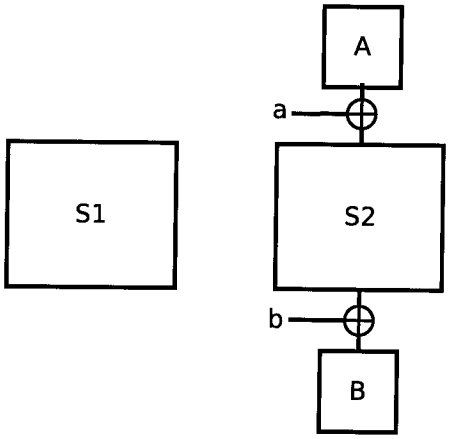


FIGURE 2.2: Schematic of the affine equivalence

structure of ciphers should be recapitulated. Considering an SP-network or a Feistel network, which consist of linear diffusion components and non-linear confusion components. The separation of those components is not strict. Parts of the linear network could be applied to the s-box, which results in a new s-box. Linear transformations can be a part of the linear network or of the s-box. Referring back to the properties, which describe the non-linear behaviour of the s-box, we would like to find groups of s-boxes that share the non-linear properties. The differences between the s-boxes within a class can be moved into the linear layer. For the classification the affine equivalence has been chosen. The affine equivalence is defined by all s-boxes that

can be transformed into each other by applying affine mappings at the input and the output of the s-box as shown in figure 2.2. The algorithm to actually determine an equivalence or to calculate the lexicographically smallest representative of the class are presented in section 3.2. In the last section, it has been shown that the linear layer has no influence on the linear and differential properties. This results in invariant linear approximation and differential distribution tables for all s-boxes of the same class. The affine mappings can only swap the elements in the table but not change values itself. The properties, such as algebraic degree, branch number and fixed points are not invariant within a class. For fixed points, it can easily be shown that every affine equivalence class has elements with and without fixed points. An affine mapping swaps elements in the lookup table according to the affine constant and can thus always create any fixed point. Even though the algebraic degrees of the output bits are not invariant within a class, the maximum degree of the outputs is preserved throughout a class.

2.3 Designing s-boxes

While in the last section the desired properties of s-boxes have been introduced, this section will show several important design methodologies that can be found in the literature. The section concludes with a comparison with the approach of this work.

One of the early methods is the random s-box design. Random lookup tables are created and their properties are compared with the constraints chosen by the designer. This process is repeated until a suitable s-box is found. The problem of this approach is that the focus is on mathematical properties and not on implementation efficiency. For table based implementation the efficiency is independent of the s-box. For bit sliced implementations it is difficult to assess the implementation cost, see [24]. Serpent is an example of a cipher that has been designed by this method. The most efficient implementations have been evaluated long after the design of the cipher by Osvik.

The random approach does not always lead to satisfying results within a practical search time. This is usually the case if specific properties are required. Some designers used special design methodologies that lead to the desired mathematical properties. An example is Rijndael with its wide trail design strategy, which defines criteria for the s-boxes and the linear layer to withstand linear and differential attacks [22]. The s-box is based on the multiplicative inverse, $S(x) = x^{-1}$, but some linear transformation are added [10].

There are many other design strategies, as for example for 6×6 s-boxes [11]. For this work we will not go further into detail about approaches for s-boxes larger than 4×4.

Another approach has been used for the ciphers Noekeon and Luffa. The approach returned to the random trials. The significant difference to the approach of Serpent is that it takes implementation issues into account. Rather than generating random lookup tables, random implementations are created. The implementations are generated as a sequence of assembly instructions. In the next step, it is checked if the

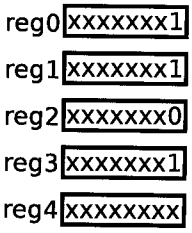


FIGURE 2.3: Example of a bit sliced representation with the value 0xB in the first slice, where register 0 contains the least significant bit

sequence represents a bit sliced implementation of an s-box with the desired properties. The search is repeated until an s-box is found that suffices the requirements. The chances to find invertible s-boxes with this random approach are rather low. It has further been discovered that even most of the valid permutations have only weak non-linear properties or are fully linear [29]. Luffa tries to avoid this by designing random sequences of more complex building blocks that guarantee to result in valid permutations.

In this thesis a new methodology is introduced. This approach focuses on implementation efficiency. Rather than trying random combinations of instructions, we use a brute force approach to search through all possible sequences of instructions. In difference to Osvik, the algorithm is not looking for a specific s-box, but trying to cover the whole range of s-boxes. Using the theory about equivalences, we look for the smallest s-box among all s-boxes with common properties. The s-box is reduced to its non-linear operation and linear operations are moved into the linear layer. The method results in a list of optimal implementations, which can then be used by the designer to work out the trade-off between implementation cost and non-linearity [15].

2.4 Implementing s-boxes

There are principally two methods to implement s-boxes in software. Often s-boxes are stored as constants in the form of lookup tables. The efficiency of this type of implementation is independent of the s-box. When accessing the s-boxes, memory lookups will be performed independent of what value is stored in the memory. The implementation method that is treated in this thesis is called bit sliced implementation. Bit slicing is a technique that enables calculating multiple bit operations in parallel on an architecture with larger word length. Bit slicing can be applied efficiently to all algorithms whose instruction sequence is independent of the data and for instructions which are bitwise, or to be more general, for instruction which do not have a carry over between the different slices. S-boxes which can be decomposed into boolean functions can be implemented in a bit sliced manner. This requires at least one more register than the size of the s-box [24]. The input is stored into a slice over all registers as indicated in figure 2.3. Taking the 8051 processor as an example,

up to eight s-boxes can be calculated in parallel on a 8-bit architecture [2]. After the sequence of bit manipulating operations have been applied, the results can be found in the registers. To compare the efficiency of the bit sliced implementations, we assume a memory access time of two cycles. The lookup table approach will thus need two cycles per s-box, or 16 cycles for eight s-boxes. The cycles needed for an s-box in a bit sliced implementation can not be determined generally because it depends on the s-box itself. But for up to 16 cycles the bit sliced method is more efficient than the lookup table, assuming a parallelism of 8. Many processor architecture allow much higher parallelism. Using SEE, processors can even work with words of 128 bit width, which means that even if the memory access time is only one cycle the break even is at 128 instructions for a bit sliced implementation. The exact implementation costs of the classes representatives can be found in the tables 5.3–5.9.

Another important property that has to be considered when implementing s-boxes is the strength against side channel attacks such as the caching attacks on look-up table implementations [8].

Chapter 3

Algorithms

In the previous chapter different concepts for s-box design, their properties and the impact on cryptanalysis have been discussed. In this chapter the basic concepts and algorithms used in this thesis will be discussed. The search algorithm and the algorithms to determine equivalences between s-boxes is looked at in more detail.

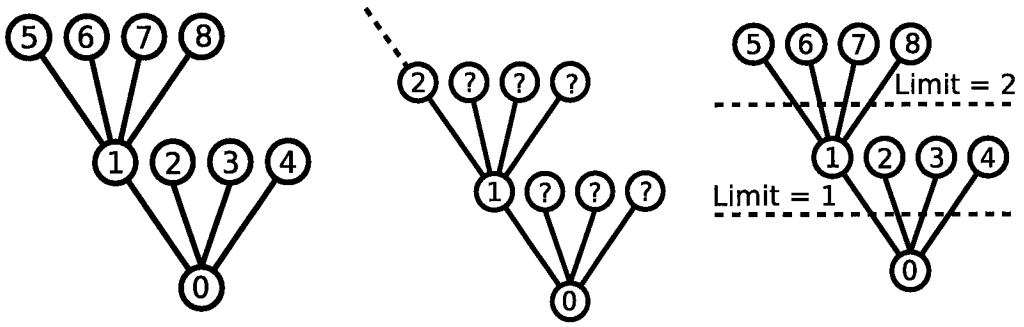
3.1 Search

A core functionality of the program is the search. This algorithm searches through all possible instruction sequences that can be constructed by a given architecture. The instruction set is limited to a subgroup of instructions that can be used for bit slicing. The following instructions have been included: AND, OR, NOT, MOV and XOR. To further limit the search space the number of registers has been limited to 5, which enables to store one intermediate result next to the the 4 bits of the s-box. To find the most efficient search strategy the following techniques have been evaluated [21]:

- breadth first search (BFS)
- depth first search (DFS)
- depth first search with iterative deepening (ID-DFS)

The breadth first search expands all nodes in the same depth first and then advances one iteration to expand all nodes of the following depth, see figure 3.1(a). This has the big advantage of minimising the computational overhead. No node has to be calculated multiple times and not more nodes than necessary have to be calculated to find all solutions. The downside of this approach is that the memory requirement is exponential in the depth of the tree. Every branches endpoint has to be saved to be expanded later on. For large branching factors the number of branch endpoints is relatively close to the total number of nodes.

The depth first search follows a branch as deep as possible. Only if it reaches an end point or a limit it backtracks and tries a different branch, see figure 3.1(b). This has the advantage that the memory usage is only linear in the depth of the tree.



(a) Example of a breadth first search

(b) Example of a depth first search

(c) Example of a iterative deepening depth first search

FIGURE 3.1: Illustrating the differences between the breadth first search, the depth first search and the iterative deepening depth first search

This method is not well suited for applications in which the maximal depth required is not known. The limit has to be guessed. If the guess was too large some branches are searched much deeper than necessary. This may result in a severe performance degradation. In the opposite case, when the depth has been guessed too low, the algorithm will not reach its goal.

We finally decided to use the iterative deepening depth first search. The search performs like a depth first search with the difference that there is a limit that is in the first iteration chosen as small as possible such that at least one node is included. When the search finishes, it is started all over again with a limit that is set to the least costly node that was above the previous limit, see figure 3.1(c). This way the memory efficiency of the depth first search can be combined with the computational efficiency of the breadth first search. The effort is approximately one over the branching factor larger than for the breadth first search, but guarantees that the algorithm does not search deeper than necessary. The decision is based on the uncertainty how the optimal implementations are distributed within the tree.

The branching factor for an instruction set consisting of the basic boolean operations (AND, OR, NOT, XOR and MOV) and 5 registers is 85. NOT has, with one common input/output register, 5 register combinations while the other 4 have each $4 \cdot 5$ combinations. This branching factor makes searches with multiple iterations practically impossible. A rule set has been designed that sorts out nodes which will not lead to valid solutions or redundant nodes. The basis of the rules has been presented by D. A. Osvik [24]. The rules include checks that no uninitialised registers are read, that no register is overwritten without being read and that no register is negated twice. For these rules, the status of the register can have four different values: uninitialised, read, written, inverted. In every node, the status is checked and updated according to the operation. Another constraint is that no information is lost during the operations. We are only considering invertible s-boxes, those require that every input maps to a distinct output value. The four bit input

of the s-box can have 16 different values. During the simulation all 16 values are stored bit sliced next to each other. Information is lost if the five registers do not represent 16 different values. In this case, the s-box is not invertible any more. Lost information can never be reconstructed again and there is no need to follow this branch any further.

Nodes with the same register content will of course have the same child branches. To avoid double nodes a cache is used. Every valid node is saved in the cache. If a node is checked, its register content is looked up in the cache. Before the registers are compared, they are sorted according to their value. For the s-box the order of the registers is not important, only the content is of interest. Removing equal notes will further decrease the branching factor. The implementation of the cache is discussed in section 3.3.

Recalling the argumentation for the search strategies, saving all nodes contradicts with the reasoning not to use breadth first search because of the memory requirements. The important difference is that, while the breadth first search depends on the this storage and can not continue after the memory limit has been exceeded, the cache is only a feature for the iterative deepening depth first search in order to reduce the branching factor. When the simulation runs out of memory, the insertion of elements to the cache is suspended. But the search can continue and meanwhile benefit from the elements stored in the cache. Nodes are still looked up in the cache even though it is not updated any more. This reduces the branching factor although less than the active cache. Next to the cache, there is a cycle check which checks for repeating elements only among all of the node's parents. This check has only limited memory requirements. The check prevents from creating loops in the tree even though the cache insertion has been suspended.

3.2 Linear and affine equivalence algorithm

The other central part of the program is the equivalence algorithm. While the search algorithm is searching through all, or a reduced set, of instruction register combinations, this algorithm has to evaluate if a newly found node is actually a member of a new class of s-boxes.

In this section, we first introduce the linear equivalence algorithm. Then, the classification of s-boxes in affine equivalence classes is explained. There are two definitions for representatives used in this thesis, one is used for the most efficient s-box of class as defined in definition 4.1. The equivalence algorithms use the lexicographically smallest s-box as representative. The second definition (definition 3.1) is used for compatibility with [17].

Definition 3.1 *The representative R is defined as being the lexicographically smallest s-box of the class. This is valid for both the affine and the linear equivalence.*

The linear equivalence algorithm determines from any given s-boxes its linear representative according to the definition 3.1. The algorithm is built after the concept of C. De Cannière.

The basic concept is to build the representative step by step such that it becomes lexicographically minimal. The algorithm has to find linear mappings A and B in order to transform the input s-box into a lexicographical smallest representative $R = B^{-1} \circ S \circ A$, see figure 2.2 (without the affine constants). By selecting the smallest possible value for the representative step by step it is guaranteed that the outcome will be the lexicographically smallest. An example is given in figure 3.2. When the algorithm starts one point is always defined in the linear mappings. Any linear operation has to map $0 \rightarrow 0$. In the example, the s-box maps 0 to 1. The minimal value that can be be assigned to $R(0)$ is thus 1. The linear mapping B can not map 0 to 1. The algorithms continues round by round. In some cases the algorithm does not have a unique solution for a round, as in the third round of the example. In those situations the algorithm has to perform a guess. In order to find the smallest solution all possible values for the guess have to be tried. We omit further details and refer to [17].

To determine if two s-boxes, e.g., S and T, are affine equivalent one has to find affine mappings that transform one s-box into the other. A straight forward way is to compare the linear representatives for every combination of input and output affine constants with the linear representative of the other s-box. This results in 256 calls of the linear equivalence algorithm. To reduce the number of calls of the linear equivalence algorithm one can compare the 16 representatives for input affine constants for one s-box with the linear representatives of the other s-box with output affine constants as shown in the formulas 3.1 and 3.2. If any of the representatives S_a is matching with one of T_b for all a and b the s-boxes are affine equivalent.

$$S_a = \text{lin. representative}[S(x \oplus a)] \tag{3.1}$$

$$T_b = \text{lin. representative}[T(x) \oplus b] \tag{3.2}$$

A s-box is represented by 4 registers. When the search reaches a new node it tests 4 combinations of 4 out of 5 registers for representing a new affine equivalence class. The combination, in which the last changed register is excluded, is neglected. Only one register changed since the last iteration. The s-box which has been chosen without the recently changed register has thus already been check in the parent node. This s-box existed already before. Every combination is treated independently as a candidate for representing a new class. For every candidate the linear representatives for all affine constants at the output, as in formula 3.2, are computed. If any of them matches with a stored representative of previously found s-boxes, it is not representing a new class and to be neglected. If it is not matching a new class has been found and 16 new values calculated according to formula 3.1 are added to the storage.

3.3 Data structures and memory management

This section focuses on the data structure of the cache. This data structure is important because it is responsible for most of the program's memory usage. Simulations

x'	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S_1(x')$	1	B	9	C	D	6	F	3	E	8	7	4	A	2	5	0

$$x \xrightarrow{A} x' \xrightarrow{S} y' \xleftarrow{B} y$$

$$\begin{aligned}
0 &\to 0 && \xrightarrow{S} && 1 \leftrightarrow 1 \\
1 &\to \mathtt{F} && \xleftarrow{S^{-1}} && 0 \leftrightarrow 0 \\
\text{guess} \to 2 &\to \underline{2} && \xrightarrow{S} && 9 \leftrightarrow 2 \\
3 &\to \mathtt{D} && \xrightarrow{S} && 2 \leftrightarrow 4 \\
4 &\to 9 && \xleftarrow{S^{-1}} && 8 \leftrightarrow 3 \\
5 &\to 6 && \xrightarrow{S} && \mathtt{F} \leftrightarrow 8 \\
6 &\to \mathtt{B} && \xrightarrow{S} && 4 \leftrightarrow \mathtt{E}
\end{aligned}$$

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$R(x)$	1	0	2	4	3	8	E									

FIGURE 3.2: Example of the linear equivalence algorithm

showed that it will not be possible to keep saving nodes in the cache throughout the whole search. Nevertheless, the size of the cache is expected to exceed several dozens of gigabytes. To access such large amounts of data efficiently, different data structures are required. The following data structures have been considered:

- binary trees
- hash tables

In a binary tree, every node can have two child nodes. The item to add or retrieve is compared with the top node first. Depending on the outcome the item if forwarded to one of the child nodes. This is continued until the requested node has been found, or the end of a branch has been reached where the new item will be stored. Every cache access has complexity of $\mathcal{O}(\log N)$ (with N the number of elements in the tree), only if the cache is balanced. This is one of the major problems of binary trees. If one keeps adding non random items to the tree, it will, with a high probability, not stay balanced. This means that certain accesses will have a higher complexity than $\mathcal{O}(\log N)$. In the worst case all nodes can be in one single branch and the complexity increases to $\mathcal{O}(N)$. To prevent this, some so called self balancing trees contain a balancing algorithm. Known examples are the red-black tree [4] and the AVL tree [14]. For the scenario of the cache, where the cache queries significantly outnumber the cache insertion, the AVL tree outperforms the red-black tree. It has more strict rules for re-balancing, which adds extra operations for insertions but speeds up the queries [26]. The major disadvantage of all tree data structure is the need of extra data to refer to the child nodes. For self balancing, even more pointers are required for the balancing algorithm. Especially on a 64-bit architecture, these pointers can cause a significant memory overhead.

The other option that is considered are hash tables. The main advantage of hash tables is that the access time is completely independent of the number of nodes. From every item, a key will be calculated with a hash function. This key is then used directly or indirectly as a memory address. This method also reduces the memory overhead because all the items in the data structure are completely independent and do not need to point to each other. The usage of hash table does not completely solve the problem of memory inefficiency. Using the hash as an address does not guarantee that all addresses are used. If two items have the same hash, only the first one will be saved in the table. For the search algorithm, the problem is limited. If a node could not be added to the cache, there is a high probability that its child nodes will then be added to the cache. This will prevent the search from expanding an equivalent branch. The other branch may just be kept one iteration longer. Towards memory saturation it becomes more probable that some entries stay unoccupied, while some items can not be saved. One possibility to improve this would to use a combination of data structures, e.g., using a hash key to distribute the items over many small, for example, linked lists. An alternative would be a so called cuckoo table. Those tables use multiple different hash function to create keys. If a element A collides with the previously stored element B, an alternative key of B is calculated and the element is moved to its new position such that A can be stored where it was initially

meant to be stored. This reordering of elements is continued until one alternative key points to an empty memory location. This increases the memory efficiency but also increases the computational effort if many keys have to be calculated and many elements have to be moved.

Taking the different properties of the data structures into account, the AVL tree has been chosen. The limitations of the hash table caused by collisions were considered more important than the loss of memory due to overhead. The implementation from the Ubuntu source has been chosen [7]. This library has been chosen because its source is publicly available. The library can be added to the source of the program and the program can be compiled on multiple machines without making sure that the correct library binaries are installed. An alternative would have been the AVL library from the GNU project [25]. The GNU project's library is a collection of implementations of many different data structures, it was preferred to limit the code size for better maintenance and debugging.

There was another approach considered for the case of porting the algorithm to a computer cluster rather than on a single machine. The memory management will be different significantly on a cluster compared to a multi core architecture. While in a multi processor core architecture, all processors share the same main memory, every node in cluster has its own main memory, that can not directly be accessed by the other nodes. A possible cluster for this work would be the VSC (Vlaams Supercomputer Centrum). Except for a limited amount of so called 'fat' nodes, the main memory is limited to 8 or 24 GB [23]. In such a scenario, a distributed data structure enables the use of a large cache. The concept for such a data structure could be a binary tree in every node and use a hash function to distribute the items over the nodes. Further explanation about the parallelisation of the algorithm can be found in section 4.3.

3.4 Conclusion

In this chapter the basic algorithms used for the simulation have been introduced. The decision for the different algorithms and libraries have been argued. For the data structure of the cache, the decision fell on the AVL tree. The decisions gave the basis to start the development of the software tool. Some of the algorithms will evolve, be optimised or even replaced during the phase of implementation. The following chapter gives an overview of those changes.

Chapter 4

Parallelisation and performance

This chapter shows how the different algorithms introduced in the previous chapter can be optimised to increase the overall performance. The chapter focuses on algorithm-level optimisation, but also includes code-level optimisation.

4.1 Reducing the branching factor

In the last chapter, we introduced the search algorithm. The most important strategy during the optimisation is to reduce the branching factor. This improves the overall speed exponentially while with code optimisation only a linear improvement can be achieved. In the previous chapter, we proposed a set of rules to reduce the branching factor. In this section, we develop a new caching approach which will achieve a much more significant reduction of the branch number.

The original caching algorithm only dismissed a node if its register content was exactly the same after sorting the registers. The new algorithm uses the same data structure, but verifies if the current node is affine equivalent to one of the previous nodes. The algorithm is inspired by the equivalence algorithm introduced in section 3.2, but there are a few significant changes.

While the general equivalence algorithm operates on 4×4-bit s-boxes, the equivalence algorithm for caching receives 5 output registers as input. This can be interpreted as a 4×5-bit s-box. The algorithm is based on the property that if two not completely finished s-boxes are equivalent, all the s-boxes that result from their child nodes will be equivalent to an s-box of the other branch. In order to visualise the equivalence relation between unfinished s-boxes, one could think of a variant of figure 2.2 where the lower half of the schematic has been cut away from both s-boxes. It is clear that if two nodes are equal, their sub-trees will be equal too. If there exists an affine input transformation that makes two nodes equal, then it can be shown that the sub-trees continue to be equal apart from these affine operations. If an affine operation is the only difference between two s-boxes, they will be member of the same affine equivalence class.

The following explanation focuses on the affine equivalence. The linear equivalence works equivalently, just without the addition of the constant. The linear mapping

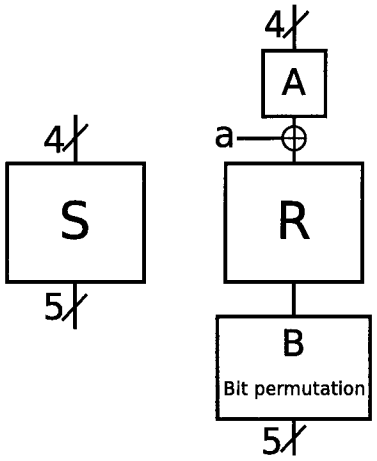


Figure 4.1: Schematic of the modified affine equivalence relationship used in the caching algorithm

at the input can be any linear transformations, as was the case for the normal equivalence algorithm. The linear mapping at the output can not be applied because the s-box has not been completed yet, and therefore the output of the s-box has not been reached yet. One special type of linear transforms, bit permutations, can still be applied. Taking into account bit permutations is equivalent to sorting, which was already applied in the first version of the caching algorithm (see section 3.1). This is always valid because changing the order of the registers does not change anything about the content. Throughout the computation of the s-box, we are only interested in the different values in the register, not in which register these values are stored. The addition of constants is also limited to the input side. There is no such mapping at the output for the same reason as for the linear mapping. Figure 4.1 shows the schematic of the equivalence. The check for complete s-boxes is performed when checking if the s-box represents a new class. It is only at this stage that affine mappings are applied both to the input and to the output.

Figure 4.2 shows the relationship between the number of nodes and the number of iterations. The data points from the basic and the linear equivalence caching are obtained from preliminary test runs, the affine equivalence has been updated during the later simulations. It can be seen that the growth of the tree behaves nearly exponentially. While the normal caching has a branching factor between 10 and 11, the advanced caching has branching factors around 7 for linear equivalence and between 6.4 and 7 for the affine equivalence. The branching factor is even slowly decreasing when advancing into the tree. To decide which algorithm is optimal, one has to consider several properties:

- computational effort per node
- reduction of future computational effort

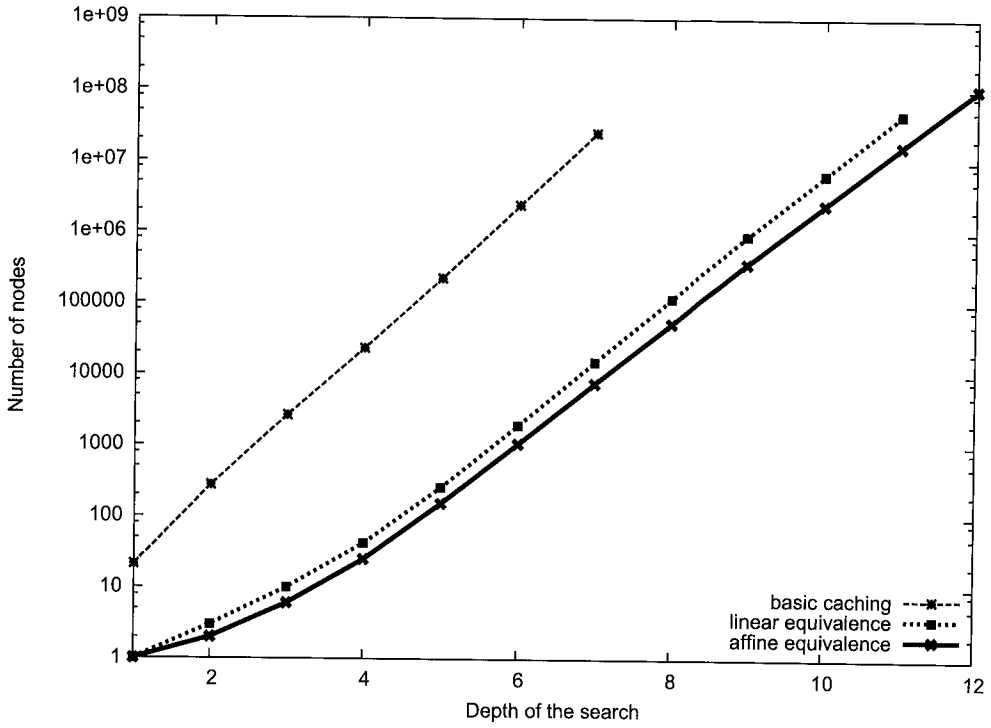


FIGURE 4.2: Comparing the different equivalence algorithms for the cache: number of nodes vs. the depth of the search

- reduction of memory usage

Using the affine equivalence reduces the branching factor and consequently also the memory usage. It also reduces the future computational effort because there will be fewer nodes that have to be checked. At the same time, the computational effort per node will increase. The effort to calculate the 16 lexicographically smallest representatives for the different affine constants is not 16 times larger than calculating a single one. Details about the optimisation are given in section 4.4.1. The trade-off also depends on the distribution of the classes in the tree. If all classes have a relatively low implementation cost, the memory would not play an important role.

It is decided to use the affine equivalence because it causes less cache memory at the expense of calculation effort. Creating less nodes per iteration enables to add elements to the cache for longer time. Without new cache elements, the branching factor increases significantly and the computational effort would be even larger.

4.2 Expected amount of nodes

In [17] an approximative formula for the number of classes is introduced. The approximation is defined by the number of s-boxes divided by the size of the linear and affine group at the input and at the output. The approximation does not take

into account that the number of elements in a class is not always equal to the number of all possible linear and affine mappings. It is an estimation formula to find the order of magnitude. Applying the approximation to affine equivalence classes this results in 201 instead of 302 classes. The number of nodes for the original algorithm with only sorting the register is calculated bellow:

#sboxes: $$S = \frac{32!}{16!}$$

#permutation transforms: $$B_{prm} = 5!$$

#classes: $$n = \frac{S}{B_{prm}} \approx 2^{66.5}$$

After applying the modified affine equivalence algorithm the number of nodes can be calculated as below:

#sboxes: $$S = \frac{32!}{16!}$$

#affine transforms: $$A = 16 \cdot (16-1) \cdot (16-2) \cdot (16-4) \cdot (16-8)$$

#permuation transforms: $$B_{prm} = 5!$$

#classes: $$n = \frac{S}{A \cdot B_{prm}} \approx 2^{48.2}$$

Every node fills at least 28 bytes needed for a data structure and another 53 bytes are required by the AVL tree node. Extra memory required by the dynamically allocated memory to store the nodes program code or the wasted memory due to memory alignment is neglected. This results in memory requirements that can not be provided. Two reasons make this approach possible nevertheless. First, not the complete space of s-boxes has to be searched to find only the optimal implementation per class. Second, even if the caching has filled all available memory, the search algorithm can still continue without adding new elements. This will cause the branching factor to increase, resulting in an even larger node count than the previously calculated $2^{48.2}$. The cache in the actual implementation can hold approximately one out of one million nodes.

4.3 Parallelisation

Another major improvement of the performance can be achieved by parallelisation. The parallel algorithm can benefit from multi-processor architectures. Two possible parallelisation approaches are considered: The first approach keeps the main program sequential. The program is then supported by several worker threads. These workers can perform two types of tasks. The first one is to check if a newly expanded node is not violating any of the earlier defined rules for reducing the branching factor. The other task checks if the register state of a node represents a new class. The sequential implementation of the program used a recursive approach for the search. For a more systematic access to variables within the different nodes, the recursion can be 'unrolled'. The data structure for the temporary data is changed from the

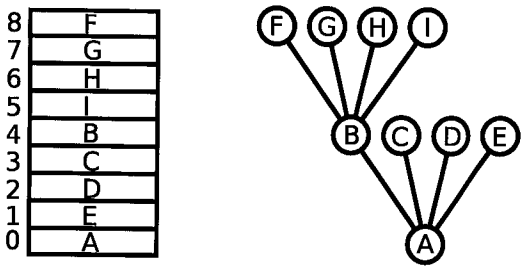


FIGURE 4.3: Example of a node expansion

function stack used by the recursion to a manual stack implemented as an array of structs with an index variable as address. The advantage of this change is that the data in the stack can be easily accessed to update the results from the worker threads, which have their own independent function stack.

Figure 4.3 shows an example graph with branching factor 4 and the corresponding stack on the left. In the first iteration, node A is expanded and nodes B to E are added to the stack (addresses 1 to 4). After the first node has been expanded, the main thread has to wait for the node B to be checked. While it is waiting, it services the worker threads. During the servicing, the main thread checks if a thread is idle. If an idle thread is found, a new job will be assigned. The servicing is expected to react fast because the task that the main thread has to perform is rather small, and it is guaranteed that the threads get serviced in regular intervals. If not all of the worker threads can be kept busy, the efficiency of the program drops. Increasing the number of threads beyond the number of available CPU cores, and thus having several threads that share a single core, would reduce the overhead. Workers that have finished their task will go into sleeping mode and give the core to another worker. This way, part of the servicing task would be done by the operating system's scheduler. Depending on the number of workers, several nodes are handled in parallel. The higher the stack address, the earlier the node will be expanded and the higher the priority for getting assigned to a worker during the servicing. For an efficient servicing of the threads, we will use a max heap data structure. The address of every newly created node is added to the data structure by the main thread. The data structure automatically orders the elements by their size. In the servicing routine, the top element of the data structures, which is by definition always the largest element in the structure, is taken and passed to the idle worker thread. A problem that is generally implied by any parallelisation, is that the calculations and thus the arrival of results may be out of order. If the cache queries are out of order, the outcome of the program is no longer deterministic but depends on the scheduling.

As an example, let us assume that node B in figure 4.3 is equivalent to node E. Depending on the scheduling, either of these equivalent nodes might be expanded first, causing the other one to be dismissed. This lack of determinism does not prevent the algorithm from finding a minimal implementation for each equivalence class. But if several minimal implementations exist for a given class, then it is not possible to predict which one will be found. In order to avoid this, we will make

the search deterministic such that each execution of the parallel algorithm always returns the exact same results as what the sequential program would have produced.

There are two reasons why determinism is a desirable feature. First, this property has clear advantages when it comes to debugging. Second, it allows to precisely define the properties of the representatives returned by the simulation:

Definition 4.1 *The representative is the s-box of an affine equivalence class with the least implementation cost and among equally costly, the first sorted by the assembly code.*

The solution for determinism is to mark cache entries as provisional. If a worker thread tries to insert an item into the cache that is already existing, the procedure depends on the provisional flag. If the colliding cache entry is not marked as provisional, a redundant node has been found and can be dismissed. If the cache element is marked as provisional the stack addresses (the stack addresses have to be saved with every cache element) have to be compared. The item with the higher stack address has priority. The new element will either be replaced or dismissed. In the first case it has to be communicated that the node that created the cache entry should be expanded. Marking the cache entries as non-provisional is done in the main thread. The provisional flag is cleared when the node is expanded. The main thread is still searching through the tree as in the sequential program, which guarantees that no other node could replace the item after it has been marked as non-provisional.

The second task of the threads, which is to analyse if a node represents a new class, is performed by the same workers. The out-of-order execution has the same effect as for the first task: It can interfere with the determinism and in some cases even lead to wrong results if jobs of equivalent s-boxes with different cost are interchanged. For this type of task, we decide to avoid out-of-order execution rather than taking actions to handle it. First, it is of interest to communicate any new found class immediately. The actions used for the caching only work because the data can be updated after it has been initially written. Second, there are far more jobs to check nodes for passing the rules than to check for new classes. We therefore only allow one thread to perform jobs of the second type. To guarantee that the search does not generate more tasks of type two than one thread can handle, the main thread waits for such tasks to finish before continuing. Despite this design decision, the code would still be able to service multiple type-two jobs at a time. A FIFO queue is used to store the jobs to be assigned. The initial assumption was that the waiting will not be lost because the main thread will meanwhile service the other worker threads with jobs of the first type. In early implementations this assumption was valid.

Another way of parallelising the algorithm would be to split the tree after several iterations and to give every thread another branch to search. When scaling, this approach is not limited by a single servicing thread, but this comes at the expense of a higher complexity. There are two main risks that arise from this parallelisation approach. First, the different branches are not necessarily balanced. This means that the different workers have to perform tasks with different calculation efforts. If

no other, more complex, precautions for servicing the threads are taken, the threads will have to wait for each other before another iteration can be started. Second, the nodes are not equally distributed within the thread. This, together with the varying execution time of the check function, causes a more complex out-of-order arrival of items for the cache.

The first method with a sequential main algorithm and supporting worker threads for extensive computational task has been chosen for the implementation. Problems with the efficiency of the parallelised algorithm occurred after the code-level optimisation introduced in section 4.4. Reducing the execution time of the tasks for the worker threads increases the relative servicing overhead. This, in combination with the increased waiting time for the jobs of type two, prevents the main thread from keeping eight cores serviced while waiting for the type-two job. On the computer used for the tests, only between three and four cores can be kept busy. Allowing type-two jobs to be performed out of order would only solve the problem partially. Servicing and expansion in one thread will stay the limiting factor of this approach. The second approach for parallelisation should be considered again. That approach uses full parallelism and is thus not limited by a single servicing thread.

4.4 Code level optimisation

The program code has been revised for efficiency during the work. In this section, the most significant changes to increase the execution speed are introduced. The work has been concentrated on the most crucial parts of the program. The two equivalence algorithms are responsible for more than 90% of the execution time and are discussed in detail in this section.

4.4.1 Caching equivalence

This algorithm has highest priority for optimisation. It contains very computationally intensive parts and it is also executed multiple times for every node. The first step when optimising the algorithm (see section 4.1) is to allow the 16 executions of the linear equivalence algorithm to benefit from each other. The aim of the algorithm is to find the lexicographically smallest of the 16 lookup tables. There is no need to calculate all of them, as done for the basic equivalence algorithm with affine mappings on both sides. Within a single execution, the initial algorithm aborts the calculations for a given guess as soon as it is clear that the result will not lead to the lexicographically smallest representative. This feature can be extended to span all 16 executions, by aborting each of them as soon as the resulting lookup table is larger than the previous ones.

The other improvement concerns the function to determine the next value of the output linear mapping. This mapping has to be a permutation of the registers. Given an input value, the function returns the smallest bit permutation, taking the previously assigned permutation into account. The first version used a 2×5 element array, which represents the 5 registers at the two sides of the permutation. The registers are grouped and then tagged. Equal tags represent how a bit can be moved

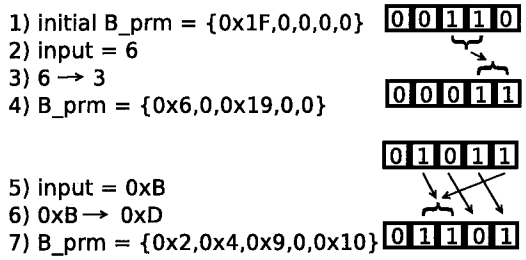


FIGURE 4.4: Example of the algorithm to find the smallest bit permutation

by the permutation. The algorithm starts by assigning the same label to all elements. The linear mapping is defined as soon as 5 different labels are present, and the new location of every bit is defined. The problem of this algorithm is that the labels of all registers have to be checked several times per execution. This is the main bottleneck. This algorithm has been replaced by a more efficient one. The new algorithm requires two pre-calculated tables which defines, for every of the 32 5-bit values, the smallest value that can be created by the same number of '1'-bits and the number of '1'-bits in the input. The permutation is now represented by an one-dimensional array, called `B_prm`. This array containes bit masks. The index of the array indicates the most right position the bits that are selected by a mask can be moved. If now new value has to be permuted to its smallest permutation its overlap with the every value in the mappings array is calculated and shifted according to the index of the array. Combining these intermediate values results in the smallest permutation. In a next step the mapping has to be updated. An example is illustrated in figure 4.4. The resulting speed-up of the algorithm can not the calculated in a general way because it depends on the inputs. Instead, the efficiency improvement has been evaluated by its impact on the search. With a profiler the execution time of a search depth limit of 5 has been measured. While the original takes about 189 s the optimised takes only 16 s.

4.4.2 Linear equivalence

When initially implementing the algorithm, the theory about the algorithm was followed closely. Most variables and data structures were inspired by the pseudo code from [17]. During the analysis, suboptimal implementation decisions have been found. Because of this, the algorithm has been redesigned[1]. The most important changes are summarised below:

- As for the modified affine equivalence algorithm, the handling of the guesses can be improved. If a guess will lead to an s-box that is known not to be lexicographically minimal, the calculations for that guess do not have to be completed. The efficiency gain of this change is data dependent. The earlier

[1]The redesign has been closely followed the advises from C. De Cannière

the lexicographically smallest representation is found, the more guesses can be skipped.

- The data structures from the initial algorithm, including mathematical groups, have been implemented in the first version. Some of the groups are not necessary and others can be replaced by more efficient implementations. Rather than searching the values in the groups, they can be calculated in a more efficient way. Gaussian elimination is used when looking for the input of the linear mapping B ($x \rightarrow y$) for which the representative becomes minimal. It checks if the requested y is already defined by a linear combination of previously mapped values. If so, the input value can simply be calculated using this linear combination. If the requested output is linearly independent, then the smallest remaining linear independent value is the solution.

The total speed-up has been measured by running 10,000 linear representative calculations with randomly generated s-boxes. While the initial algorithm takes about 69μs per call, the optimised algorithm's averaged execution time can be reduced to 7μs. This equals to a factor of 9.8.

4.5 Checkpointing

Even with all the performance enhancements introduced in this chapter the search may take several weeks to complete. A system crash would require a simulation restart in order to rebuild the cache. All the calculation effort of the previous simulation would be lost. Therefore a checkpointing system has been implemented. The checkpointing system enables the program to save its state on the hard disk, and recover from it after a system failure. The check points consist of two parts. One part is continuously updated; the other is saved at regular intervals.

All data, except the cache, is saved in regular intervals, by default every two hours. The file for the cache is called `suspend_cache` and the other file is called `suspend_all`. The second file includes:

- the stack
- the saved lexicographically smallest representatives of already found classes
- the limit of the iterative deepening DFS
- the number of elements in the cache
- the number of nodes (can vary from the number of element in the cache if caching is deactivated because of limited RAM resources)
- the number of classes found

The data structures, containing open jobs for the threads to do, are not saved. The main thread is paused and the worker threads get serviced, until those queues are

emptied before creating a checkpoint. This brings the program into a well defined state.

For the cache entries, another way of scheduling the checkpointing has been chosen. Saving the whole cache of several gigabytes at once would take a large amount of time. It is also not optimal not to save any data while the program is running and leaving the I/O buses idle, while a little later a large writing task slows down the whole program because of the delays of the I/O bus. To avoid this, the cache entries are saved in a file each time a value has been labelled as non-provisional. At this moment it is sure that the entry will never change again. When recovering the data, the two files may not be synchronised. After creating a check point the file `suspend_cache` is still updated, and contains new values by the time the program is terminated. For this reason, the number of cache elements is saved in `suspend_all`. When reading the cache, only the number of cache elements defined in the checkpoint are read, and the rest of the content in `suspend_cache` will be discarded. This results in a loss of part of the data but the loss is limited by the checkpointing interval.

4.6 Conclusion

Thanks to the optimisation, the total execution time was reduced significantly. The already reduced branching factor of ten could be reduced below seven by more advanced caching methods. With code level optimisations, the execution times of the most crucial algorithms have been improved by approximately a factor of ten. Additionally, two approaches to benefit from multi-core architectures and computer clusters have been introduced. The limits of the parallelisation approach have been shown and techniques to further improve the program have been presented. Finally, the reliability against system failures has been enhanced with the checkpointing algorithm.

Chapter 5

Results

In the previous chapters we have introduced the algorithms and tools to search for efficient implementations. In this chapter we present the outcome of the search.

In the first section of this chapter, the best implementations for each class that have been found so far are presented and the outcome are analysed. In section 5.2, we discuss the observation that none of the optimal implementations contains a NOT instruction. The following section 5.3 compares the found representatives with s-boxes used in known primitives such as Serpent, Noekeon and Luffa. In section 5.4, we give an explanation of how to use the table to design new primitives.

5.1 The most efficient implementations

In this section, we give an overview of all optimal implementations found during the simulations. The simulation time is difficult to guess and the simulation takes longer than we expected. At the time of writing, 272 of the 302 classes have been found. Nevertheless, many important equivalence classes are covered and other interesting properties are discovered. The found classes cover 90% of all s-boxes. The results of the simulation are a table containing the representatives, according to definition 4.1. The results are listed in the tables 5.3-5.9. The entries are sorted according to the following conditions:

1. The non-linear properties: The linear and differential properties have been combined. In order to take the different attack complexities into account, the linear correlations have been squared before merging the two tables. The columns of the tables have been sorted according to this value, and summed up in case of equality. This merged table was then used to sort the s-boxes.

2. The s-box classes for which an optimal implementation has been found, have a higher priority than classes with unknown representative.

3. The classes with an optimal implementation have then been sorted according to the implementation cost.

4. The ranking of s-box classes in the thesis of C. De Cannière has been used as sorting condition for classes for which both the non-linear properties and the implementation cost are the same [17].

Figure 5.1 shows the relationship between the figure of merit and the implementation cost. The size of the points indicates the number of classes that have exactly the same figure of merit and implementation cost. The figure of merit is calculated as follows:

$$fom = \sum_c \left(c \cdot \log_2 \frac{1}{c} \right)^2 \cdot LIN(c) + \sum_p p \cdot \log_2 \frac{1}{p} \cdot DIFF(p) \tag{5.1}$$

where LIN and DIFF are the histograms of the linear approximation table 2.4 and the difference distribution table 2.2. The variables c and p are the correlation and probability for which the histograms are defined.

This figure of merit takes a weighted average over the different properties. For some attacks, the existence of single extreme values may cause security flaws. But the figure of merit gives an indication of the non-linearity introduced by an s-box.

The general tendency is that the higher the non-linear requirements are, the more cycles are needed. Nevertheless, not all representatives are on the Pareto optimum. Deviations from the general trend can also be observed for other properties, such as the maximum linear probability and the maximum differential probability. The tables 5.1 and 5.2 show the relationship between the implementation cost and the MDP and the MLP. These variations can be very useful for the design of ciphers. When designing ciphers there is a trade-off between the non-linear properties and the implementation cost of the s-boxes. The complexity of an attack can either be increased by choosing s-boxes that have properties as close as possible to the ideal s-box, or by increasing the number of rounds.

In appendix B we present a table with additional properties. An interesting property is the branch number of the representatives. For all classes found so far it is the minimal possible value of 2. The maximum algebraic degree is also listed. It is important to remember that the algebraic degree of the other output bits may vary.

When choosing a representative, the designer should be aware of the fact that the linear layer can change the branch number and has limited influence on the

TABLE 5.1: Minimum cost required to implement an s-box with a given MLP

MLP = 1/2+ |c|	1/8 1/4	1/4 1/2	3/8 3/4	1/2 1
min. cost	-	9	9	0

TABLE 5.2: Minimum cost required to implement an s-box with a given MDP

MDP	1/8	1/4	3/8	1/2	5/8	3/4	7/8	1
min. cost	-	9	10	6	9	6	-	0

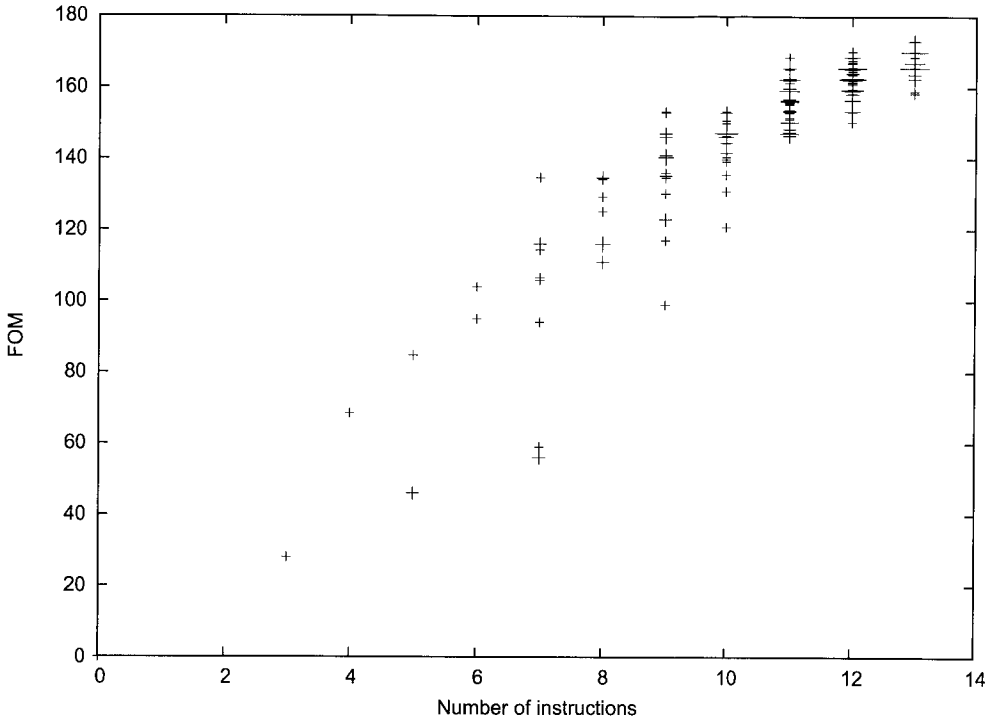


FIGURE 5.1: The relationship between the figure of merit and the implementation cost

algebraic degree of the different output bits. It can merge or cancel out monomials by performing additions between the output bits. However, it can never create new monomials of higher degree than the maximum degree present. Referring to [12] it is important to consider that not only the maximum degree but also the amount and distribution of the monomials is of importance.

5.2 Affine equivalence and the NOT instructions

When analysing the results it has been discovered, that not a single implementation makes use of the NOT instruction. This discovery leads to an important property: the lack of inverters implies the existence of the fixed point $S(0) = 0$. Not all representatives have been found by the simulation so far. Therefore, the implicit proof that all classes have an optimal implementation without NOT instruction is not complete and thus not valid. We would like to introduce a hypothesis: We conjecture that all classes contain at least one s-box that has minimal implementation cost and has the fixed point $S(0) = 0$.

The hypothesis can be verified by completing the search or by a mathematical proof. The existence of optimal implementations with other fixed points or without fixed points could not be shown with this simulation and can thus not be excluded.

For class 13, it is interesting to notice that the best implementation without fixed point found by [29] needs one more instruction than representative with fixed point. The search for the other s-box was slightly different. Parallelism has been taken into account such that the 10 instructions can be executed in 6 cycles. For an example please refer to section 5.3.2.

In the following discussion we introduce an argumentation why the NOT instruction is not needed.

$$\begin{aligned} \overline{A \cup B} &= \overline{A} \cap \overline{B} \\ \overline{A \cap B} &= \overline{A} \cup \overline{B} \end{aligned} \tag{5.2}$$

For the argumentation we will treat two types of implementations separately. The first type resembles a round of an unbalanced Feistel network [27]. The non-linear combination of a subset of the registers (comparable to one side of the Feistel cipher) is combined by a binary addition with another register (comparable to the other side). The non-linear network is limited to make use of only one internal register. All s-boxes based on this type of building blocks are guaranteed to be invertible. The second type includes all other constructions. The difficulty of this type is that many combinations of this type will result in non-invertible s-boxes [29]. Nevertheless, there are invertible s-boxes and even optimal implementations of this type.

Recalling the affine equivalence, it can be shown that every NOT instruction that can be moved to the input or output, can also be removed. The building blocks of type 1 s-boxes guarantee that the De Morgan's transform (see equation 5.2) can move any NOT instruction inside the s-box to either of the ends, where it can be discarded because of the affine mapping. This argumentation is only valid for architectures with only one spare register. Having only one register that can be biased, it is not possible to create a 'De Morgan loop'. This is only valid for type 1 s-boxes. In De Morgan loops, the De Morgan transformation will not be able to move all NOT instructions to either end of the s-box. At least one NOT instruction will always remain inside the loop. Figure 5.2 shows an example of such a loop.

No such closed argumentation has been found for type two. The examples that have been analysed do not have an isolated De Morgan loop. They contain of similar structures, but some of the intermediate values inside the loop are linearly combined with other registers. These linear combinations seem to allow to transform any inserted NOT instruction to one of the ends of the s-box. An example is the representative of class 115 in figure 5.3. It can not be excluded that there does not exist any s-box with minimal implementation cost containing a De Morgan loop. But it has been shown that among the found representatives there exist results without them.

We remain with this argumentation. It does not prove, that there is not an optimal implementation with NOT instructions, but it proves (only if all classes have been found), that there is for every class at least one optimal implementation without NOT instruction. We leave more detailed investigations and a formal proof for further research.

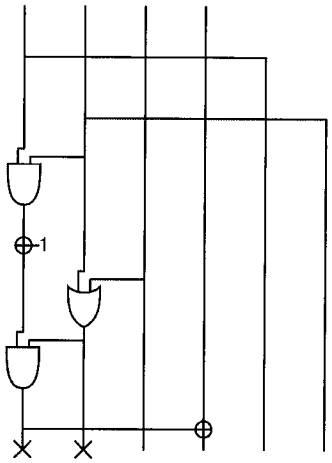


FIGURE 5.2: Example of a s-box with a non-resolvable De Morgan loop

5.3 Comparison with literature

5.3.1 Serpent

The following s-box classes of Serpent have been found: 9 (S_4, S_5), 10 (S_4^{-1}, S_5^{-1}), 14 (S_0^{-1}, S_1), 15 (S_0, S_1^{-1}), 16 (S_2, S_2^{-1}, S_6, S_6^{-1}). The class of S_3, S_3^{-1}, S_7 and S_7^{-1} has not been found yet. Five of the six s-boxes are members of the most efficient classes out of the 16 classes with $MDP = 1/4$ and $MLP = 1/2 + 1/4$, see table 5.3. The s-box whose representative has not been found, can be classified according to its properties. It is in class 12. This means that none of the s-boxes is member of the most efficient class 13.

None of the s-boxes used in Serpent is a most efficient representative as presented in the tables 5.3-5.9. The reason are the variant properties of a class. All representatives fail the second part of condition 1 (see figure 5.4). The MDP is invariant for a class and consequently fulfils the conditions. The position of the elements in the differential distribution table is variant. The condition further requires a probability of zero that a one bit input difference leads to a one bit output difference. This is just another way to define a bitwise branch number to be larger than 2. None of the representatives of classes 9, 10, 15 and 16 passes this condition. By knowing the implementation costs of a representative, one can not make statements about the implementation costs of other members of that class. It is thus not possible to define generally how many instructions have to be added, in order to fulfil other variant requirements.

5.3.2 Luffa

The Luffa s-box $S =$ `0xde015a76b39cf824` is of the same class as S_2, S_2^{-1}, S_6 and S_6^{-1} from Serpent [20]. The representative failed again as a candidate because of

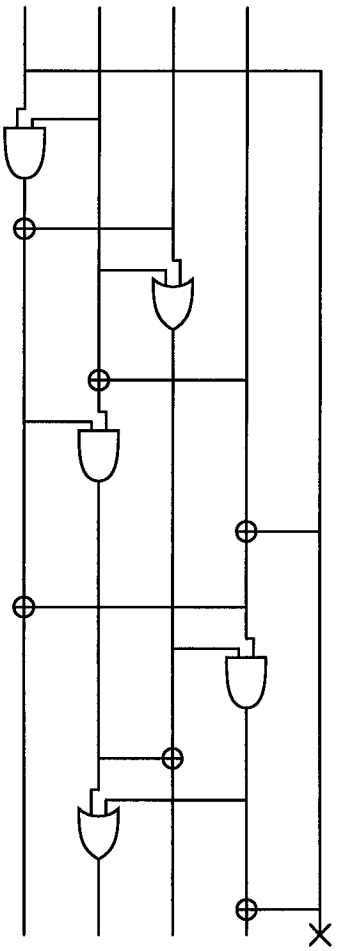
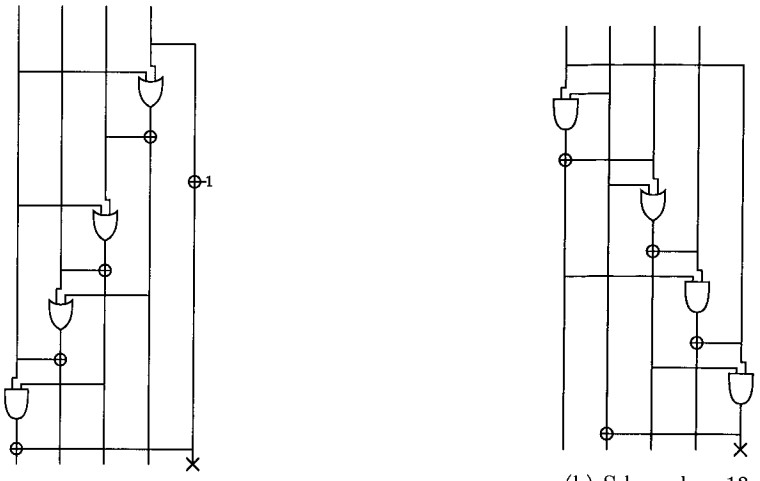


FIGURE 5.3: Example of a s-box of type 2

1. each differential characteristic has a probability of at most $1/4$, and a one-bit input difference will never lead to a one-bit output difference
2. each linear characteristic has a probability in the range $1/2 \pm 1/4$, and a linear relation between one single bit in the input and one single bit in the output has a probability in the range $1/2 \pm 1/8$
3. the non-linear order of the output bits as a function of the input bits is the maximum, namely 3.

FIGURE 5.4: The s-box constraints of Serpent [1]



(a) 'smallest s-box ever' by Watanabe

(b) S-box class 13

FIGURE 5.5: Comparing 'smallest s-box ever' from Luffa with the equivalent found by us

variant properties. The Luffa specifications do not allow fixed points and require that every output bit has a degree of 3.

Dai Watanabe claims in [29] that he found the 'smallest s-box ever' with the optimal MLP of $1/2 + 1/4$, MDP of $1/4$ and no fixed point. Our research could verify this claim. Allowing to have a fixed point results in a slightly different but equivalent result. The s-box found by Watanabe is shown in figure 5.5(a). The smallest s-box with the same properties, except for the fixed point, found by our search is member of the same equivalence class, see figure 5.5(b).

5.3.3 Noekeon

Noekeon is a block cipher based on a substitution/linear transformation network [9]. It shares similarities with the AES candidate Serpent. One important difference is its symmetric structure, which nominates the cipher for efficient bit sliced implementation. The s-box used is member of class 13. The representative can not be used in the cipher because Noekeon requires s-boxes that are involutions. An s-box is called an involution if it is equal to its inverse, $S = S^{-1}$. This property is variant within a class. But not all classes can contain s-boxes with this property. The inverse of an involution has to be member of the same class as the s-box because the inverse is equal to the s-box. Therefore, a class can only contain involutions if its inverses are member of the same class. In Appendix B we present an extended table of s-boxes with some additional properties. The last row of this table refers to the class of the inverses.

5.4 A new design approach

When we compare our results to the s-boxes used in known primitives, it is usually not possible to judge the decisions taken by the designers based on the outcome of our simulations. The reason are the different design work-flows.

The sequence that is often used leaves the design of the s-box to the end. During the design of the other components of the cipher, the specifications of the s-box become more and more detailed.

When trying to replace one of the s-boxes with one of the most efficient representatives, there is a high chance that the specifications are not fulfilled. The specifications contain many properties that are variant in affine equivalence classes. In our approach, the s-box is selected as a first step. The designer therefore fixes only those properties that are invariant within the classes. The optimal s-box of a class can be found with the tool presented in this thesis. In appendix A; we give the optimal implementations for the instruction set defined in section 3.1.

In the next step, the variant properties are treated. The fact that these are variant in affine equivalence classes, implies that these properties can also be affected by the linear layer. The linear layer is then chosen such that the overall specifications are fulfilled.

We expect that this alternative design methodology will result in efficient primitives. Nevertheless, it is not proven that the most efficient members of the classes will also lead to optimal ciphers.

5.5 Conclusion

In this chapter we presented the results of the simulation. Even though the simulation has not finished yet, and there are still some representatives to be found, many interesting properties have been extracted.

We could show that some properties that are variant throughout a class, are the same for all representatives that have been found. Some of these properties may be considered as weak or at least undesirable in some traditional design methodologies. All of the found representatives found so far have a fixed point mapping zero to zero. Furthermore, we could show that all of them have weak mixing properties, as indicated by their branch number of 2.

For the implementations, we have shown that the minimal implementations (of all s-box classes found) are not depending on the NOT instructions. This results in the interesting property that all of them have a fixed point.

We finally presented a design strategy for cryptographic primitives. The design strategy aims for highly efficient primitives by selecting optimal s-boxes and adapting the linear layer in a next step to reach the desired properties.

TABLE 5.3: Implementations of the affine equivalence classes 1–50

	Representative	$\lvert c\rvert = 1/4$	1/2	3/4	1	$p = 1/8$	1/4	3/8	1/2	5/8	3/4	7/8	1	cost
1	?	120	30	0	1	90	15	0	0	0	0	0	1	?
2	?	120	30	0	1	90	15	0	0	0	0	0	1	?
3	?	120	30	0	1	90	15	0	0	0	0	0	1	?
4	?	120	30	0	1	90	15	0	0	0	0	0	1	?
5	?	120	30	0	1	90	15	0	0	0	0	0	1	?
6	?	120	30	0	1	90	15	0	0	0	0	0	1	?
7	?	120	30	0	1	90	15	0	0	0	0	0	1	?
8	?	120	30	0	1	90	15	0	0	0	0	0	1	?
9	0cabf9d4e8635172	112	32	0	1	84	18	0	0	0	0	0	1	11
10	01298bd7cfe654a3	112	32	0	1	84	18	0	0	0	0	0	1	12
11	0a43562edfb1c789	112	32	0	1	84	18	0	0	0	0	0	1	13
12	?	112	32	0	1	84	18	0	0	0	0	0	1	?
13	086d5f7c4e2391ba	96	36	0	1	72	24	0	0	0	0	0	1	9
14	086c7e5f4d21b39a	96	36	0	1	72	24	0	0	0	0	0	1	10
15	0845d7fec6a391b2	96	36	0	1	72	24	0	0	0	0	0	1	10
16	01a2987cdef4563b	96	36	0	1	72	24	0	0	0	0	0	1	11
17	?	120	30	0	1	93	12	1	0	0	0	0	1	?
18	02839b7eca65df14	112	32	0	1	87	15	1	0	0	0	0	1	12
19	04afb6372e81c95d	112	32	0	1	87	15	1	0	0	0	0	1	12
20	02415f3e8bc6a9d7	112	32	0	1	87	15	1	0	0	0	0	1	12
21	0251c6afd7984e3b	112	32	0	1	87	15	1	0	0	0	0	1	13
22	?	112	32	0	1	87	15	1	0	0	0	0	1	?
23	?	120	30	0	1	96	9	2	0	0	0	0	1	?
24	?	120	30	0	1	96	9	2	0	0	0	0	1	?
25	0c69735248af1dbe	96	36	0	1	78	18	2	0	0	0	0	1	11
26	06a953b842c7df1e	96	36	0	1	78	18	2	0	0	0	0	1	11
27	0a2387bfc5de4961	96	36	0	1	78	18	2	0	0	0	0	1	12
28	0a2387bf4d56c1e9	96	36	0	1	78	18	2	0	0	0	0	1	12
29	0913a4bf2e6587dc	96	36	0	1	78	18	2	0	0	0	0	1	12
30	06af7d5e48c391b2	96	36	0	1	81	15	3	0	0	0	0	1	11
31	04598ceb6a72f3d1	96	36	0	1	80	18	0	1	0	0	0	1	11
32	08a319f4c6e5d7b2	64	44	0	1	64	24	0	2	0	0	0	1	9
33	086d5f7e4c2193ba	64	44	0	1	64	24	0	2	0	0	0	1	9
34	?	119	28	1	1	78	21	0	0	0	0	0	1	?
35	?	119	28	1	1	78	21	0	0	0	0	0	1	?
36	?	119	28	1	1	78	21	0	0	0	0	0	1	?
37	03298bd5efc476a1	111	30	1	1	72	24	0	0	0	0	0	1	11
38	03d74f985ec621ab	111	30	1	1	72	24	0	0	0	0	0	1	12
39	0e8952d7ca4b61f3	119	28	1	1	81	18	1	0	0	0	0	1	13
40	0283db4eca769f15	119	28	1	1	81	18	1	0	0	0	0	1	13
41	?	119	28	1	1	81	18	1	0	0	0	0	1	?
42	?	119	28	1	1	81	18	1	0	0	0	0	1	?
43	0c2784fa5961e3db	111	30	1	1	75	21	1	0	0	0	0	1	12
44	0c4d9fba8e635172	111	30	1	1	75	21	1	0	0	0	0	1	12
45	06abc84e79f2d153	111	30	1	1	75	21	1	0	0	0	0	1	12
46	0d9163e5fb7ac842	119	28	1	1	84	15	2	0	0	0	0	1	13
47	?	119	28	1	1	84	15	2	0	0	0	0	1	?
48	?	119	28	1	1	84	15	2	0	0	0	0	1	?
49	?	119	28	1	1	84	15	2	0	0	0	0	1	?
50	?	119	28	1	1	84	15	2	0	0	0	0	1	?

Table 5.4: Implementations of the affine equivalence classes 51–100

	Representative	$\lvert c\rvert = 1/4$	1/2	3/4	1	$p = 1/8$	1/4	3/8	1/2	5/8	3/4	7/8	1	cost
51	0283db7eca659f14	111	30	1	1	78	18	2	0	0	0	0	1	11
52	0285cf4b9a36de71	111	30	1	1	78	18	2	0	0	0	0	1	11
53	0c3e97af86d4512b	111	30	1	1	78	18	2	0	0	0	0	1	12
54	038a75dbcf6e1294	111	30	1	1	78	18	2	0	0	0	0	1	12
55	038a64dbcf7e1295	111	30	1	1	78	18	2	0	0	0	0	1	12
56	0481e37d6afbc952	111	30	1	1	78	18	2	0	0	0	0	1	12
57	04987bcf6ad251e3	111	30	1	1	78	18	2	0	0	0	0	1	12
58	0c2db39a6e857f14	111	30	1	1	78	18	2	0	0	0	0	1	12
59	086e7d5c4f21b39a	111	30	1	1	78	18	2	0	0	0	0	1	12
60	0cf1634b9d25a78e	111	30	1	1	78	18	2	0	0	0	0	1	12
61	0a24193685def7bc	111	30	1	1	78	18	2	0	0	0	0	1	12
62	0a23486519dcfb7e	111	30	1	1	78	18	2	0	0	0	0	1	12
63	04f28d617be3c95a	111	30	1	1	78	18	2	0	0	0	0	1	13
64	0cfbae138594726d	111	30	1	1	78	18	2	0	0	0	0	1	13
65	0debaf129584736c	111	30	1	1	78	18	2	0	0	0	0	1	13
66	07d9af54bec86231	111	30	1	1	78	18	2	0	0	0	0	1	13
67	0ae36592748cdf1b	111	30	1	1	78	18	2	0	0	0	0	1	13
68	086293efc7b5d4a1	111	30	1	1	78	18	2	0	0	0	0	1	13
69	04816aced372f95b	111	30	1	1	78	18	2	0	0	0	0	1	13
70	0281df5bce679a34	111	30	1	1	78	18	2	0	0	0	0	1	13
71	0182cf6ade579b34	111	30	1	1	78	18	2	0	0	0	0	1	13
72	0283db7fca659e14	111	30	1	1	78	18	2	0	0	0	0	1	13
73	0243d6aec7b95f18	111	30	1	1	78	18	2	0	0	0	0	1	13
74	?	111	30	1	1	78	18	2	0	0	0	0	1	?
75	?	111	30	1	1	78	18	2	0	0	0	0	1	?
76	?	111	30	1	1	78	18	2	0	0	0	0	1	?
77	0bf36482759cde1a	111	30	1	1	81	15	3	0	0	0	0	1	13
78	?	111	30	1	1	81	15	3	0	0	0	0	1	?
79	08e42ac1d7f5b396	95	34	1	1	72	18	4	0	0	0	0	1	11
80	04693fd17b52eac8	95	34	1	1	72	18	4	0	0	0	0	1	11
81	09e65cf74d82ba31	95	34	1	1	72	18	4	0	0	0	0	1	11
82	0c6bd9f2e8a51734	95	34	1	1	72	18	4	0	0	0	0	1	11
83	08c52ae1d4f6b397	95	34	1	1	72	18	4	0	0	0	0	1	12
84	04ae8c219fbd5376	63	42	1	1	78	0	14	0	0	0	0	1	10
85	0dbea6372f91c845	111	30	1	1	80	18	0	1	0	0	0	1	12
86	0ea62c93f4d587b1	111	30	1	1	80	18	0	1	0	0	0	1	13
87	0913b2c486ed5a7f	118	26	2	1	72	21	2	0	0	0	0	1	13
88	0d7c2a186e5fb349	118	26	2	1	72	21	2	0	0	0	0	1	13
89	0c6749be1532f8da	118	26	2	1	72	21	2	0	0	0	0	1	13
90	0e2f84acb7d65319	118	26	2	1	72	21	2	0	0	0	0	1	13
91	0329d7e8f4c51ba6	118	26	2	1	72	21	2	0	0	0	0	1	13
92	?	118	26	2	1	72	21	2	0	0	0	0	1	?
93	0c2dbf16ae497358	110	28	2	1	66	24	2	0	0	0	0	1	11
94	095f18e4a7b3d2c6	110	28	2	1	66	24	2	0	0	0	0	1	11
95	04e8ca639bfd5172	110	28	2	1	66	24	2	0	0	0	0	1	12
96	0bd57f243a18e6c9	118	26	2	1	75	18	3	0	0	0	0	1	13
97	0913b24ca6ed587f	118	26	2	1	75	18	3	0	0	0	0	1	13
98	0425be968fdc731a	118	26	2	1	75	18	3	0	0	0	0	1	13
99	0724ae85bfdc6319	118	26	2	1	75	18	3	0	0	0	0	1	13
100	01b2c5e3f7d689a4	118	26	2	1	75	18	3	0	0	0	0	1	13

TABLE 5.5: Implementations of the affine equivalence classes 101–150

	Representative	$\lvert c\rvert = 1/4$	1/2	3/4	1	$p = 1/8$	1/4	3/8	1/2	5/8	3/4	7/8	1	cost
101	0ac5d736f4b912e8	118	26	2	1	75	18	3	0	0	0	0	1	13
102	09657cade4831bf2	118	26	2	1	75	18	3	0	0	0	0	1	13
103	0821e7ca6fd593b4	118	26	2	1	75	18	3	0	0	0	0	1	13
104	012bd4e3c7f598a6	118	26	2	1	75	18	3	0	0	0	0	1	13
105	?	118	26	2	1	75	18	3	0	0	0	0	1	?
106	0a387f496edcb125	110	28	2	1	69	21	3	0	0	0	0	1	11
107	0425bd968ecf731a	110	28	2	1	69	21	3	0	0	0	0	1	11
108	03298bd5cfe476a1	110	28	2	1	69	21	3	0	0	0	0	1	11
109	086e5c7d4f2391ba	110	28	2	1	69	21	3	0	0	0	0	1	11
110	06853d942cab71fe	110	28	2	1	69	21	3	0	0	0	0	1	11
111	0bd74f985ec621a3	110	28	2	1	69	21	3	0	0	0	0	1	12
112	06e915dbf37a42c8	110	28	2	1	69	21	3	0	0	0	0	1	12
113	0ec9731dfb52a486	110	28	2	1	69	21	3	0	0	0	0	1	12
114	08a1f356e24db79c	110	28	2	1	69	21	3	0	0	0	0	1	12
115	0942563718fdabec	110	28	2	1	69	21	3	0	0	0	0	1	12
116	0c8962e5fb7a41d3	118	26	2	1	78	15	4	0	0	0	0	1	13
117	?	118	26	2	1	78	15	4	0	0	0	0	1	?
118	?	118	26	2	1	78	15	4	0	0	0	0	1	?
119	048e26c31f957bda	110	28	2	1	72	18	4	0	0	0	0	1	12
120	04cae86bf1d35972	110	28	2	1	72	18	4	0	0	0	0	1	12
121	03a1df79ec658b24	110	28	2	1	72	18	4	0	0	0	0	1	12
122	0281ce6bdf549a37	110	28	2	1	72	18	4	0	0	0	0	1	12
123	0281ce7bdf459a36	110	28	2	1	72	18	4	0	0	0	0	1	12
124	0d14376cfae2958b	110	28	2	1	72	18	4	0	0	0	0	1	12
125	0b12756acfe4938d	110	28	2	1	72	18	4	0	0	0	0	1	12
126	08f5b1e42ac3d697	110	28	2	1	72	18	4	0	0	0	0	1	12
127	02418ae693fcd7b5	110	28	2	1	72	18	4	0	0	0	0	1	12
128	0bd74f91c65ea823	110	28	2	1	72	18	4	0	0	0	0	1	13
129	08e52ac1f4d693b7	110	28	2	1	72	18	4	0	0	0	0	1	13
130	08a2d5e3f6c791b4	118	26	2	1	74	21	0	1	0	0	0	1	13
131	0d91ea6572f3c84b	118	26	2	1	77	18	1	1	0	0	0	1	12
132	0481e37dfa6b59c2	110	28	2	1	71	21	1	1	0	0	0	1	12
133	0c86d352f74e19ba	110	28	2	1	71	21	1	1	0	0	0	1	12
134	0829b71ea64df35c	110	28	2	1	74	18	2	1	0	0	0	1	11
135	0c635172e8abf9d4	110	28	2	1	74	18	2	1	0	0	0	1	11
136	04e935fb71d86ac2	110	28	2	1	74	18	2	1	0	0	0	1	12
137	0ac9f75d1b32e684	110	28	2	1	74	18	2	1	0	0	0	1	12
138	0591e26d7afbc843	110	28	2	1	74	18	2	1	0	0	0	1	12
139	08f43bd6912c7e5a	110	28	2	1	74	18	2	1	0	0	0	1	12
140	0821f396ea45b7dc	110	28	2	1	74	18	2	1	0	0	0	1	12
141	0f415a6b97d2e8c3	110	28	2	1	74	18	2	1	0	0	0	1	12
142	0283df7ace659b14	94	32	2	1	62	24	2	1	0	0	0	1	9
143	0a6d5f7c4e2391b8	94	32	2	1	62	24	2	1	0	0	0	1	9
144	04ac8e239fbd5176	94	32	2	1	62	24	2	1	0	0	0	1	10
145	0821f6dac5e4b397	94	32	2	1	62	24	2	1	0	0	0	1	10
146	0814d5ea7f62b3c9	94	32	2	1	62	24	2	1	0	0	0	1	10
147	0425b796aec1f3d8	94	32	2	1	62	24	2	1	0	0	0	1	10
148	04617b5a8ce3d9f2	94	32	2	1	62	24	2	1	0	0	0	1	10
149	0c63d1fae82597b4	94	32	2	1	62	24	2	1	0	0	0	1	10
150	0e6bd9f2c8a51734	94	32	2	1	62	24	2	1	0	0	0	1	10

TABLE 5.6: Implementations of the affine equivalence classes 151–200

	Representative	$\lvert c\rvert$ = 1/4	1/2	3/4	1	p = 1/8	1/4	3/8	1/2	5/8	3/4	7/8	1	cost
151	04af8d21be9c7356	94	32	2	1	62	24	2	1	0	0	0	1	11
152	0c81bf53d9762ea4	94	32	2	1	62	24	2	1	0	0	0	1	11
153	0a2395b8d7f6c4e1	94	32	2	1	62	24	2	1	0	0	0	1	11
154	0824f6ae5d7391cb	94	32	2	1	62	24	2	1	0	0	0	1	11
155	04639fd2e8cb517a	94	32	2	1	64	24	0	2	0	0	0	1	10
156	0c69a24ef758b31d	117	24	3	1	66	21	4	0	0	0	0	1	12
157	08296c5a4fdeb317	117	24	3	1	66	21	4	0	0	0	0	1	13
158	0bd57f26183ac4e9	117	24	3	1	69	18	5	0	0	0	0	1	13
159	08e5c7a4b391f6d2	117	24	3	1	69	18	5	0	0	0	0	1	13
160	08a9f65db217e3c4	117	24	3	1	69	18	5	0	0	0	0	1	13
161	0a8b46d2ce7f1395	117	24	3	1	69	18	5	0	0	0	0	1	13
162	08235cfae64719bd	117	24	3	1	69	18	5	0	0	0	0	1	13
163	0ac46e251b39f7d8	109	26	3	1	63	21	5	0	0	0	0	1	11
164	08e4c6a591b3d7f2	109	26	3	1	63	21	5	0	0	0	0	1	11
165	0ce53b91d7f284a6	109	26	3	1	63	21	5	0	0	0	0	1	11
166	046abec9f8d27351	109	26	3	1	63	21	5	0	0	0	0	1	11
167	0823b79ad5f64ce1	109	26	3	1	63	21	5	0	0	0	0	1	11
168	0a4e86c13bd5f792	109	26	3	1	63	21	5	0	0	0	0	1	12
169	06e842c397f5b1da	109	26	3	1	63	21	5	0	0	0	0	1	12
170	05bf8d349cae6217	109	26	3	1	63	21	5	0	0	0	0	1	12
171	0291e8a3f6d5b7c4	117	24	3	1	72	15	6	0	0	0	0	1	12
172	068cea2db7951f34	109	26	3	1	66	18	6	0	0	0	0	1	12
173	08235cfae74619bd	109	26	3	1	66	18	6	0	0	0	0	1	12
174	0219d7e3c6f48ab5	109	26	3	1	66	18	6	0	0	0	0	1	12
175	03a1ec69df748b25	109	26	3	1	66	18	6	0	0	0	0	1	12
176	0283ca6edb749f15	109	26	3	1	66	18	6	0	0	0	0	1	12
177	032476a5cfe98bd1	109	26	3	1	66	18	6	0	0	0	0	1	12
178	0289f75a6c4d3b1e	109	26	3	1	69	15	7	0	0	0	0	1	12
179	0ea3c84671f2d95b	117	24	3	1	65	24	1	1	0	0	0	1	12
180	072634bc58f9e1ad	117	24	3	1	65	24	1	1	0	0	0	1	13
181	06b3e9c1fa4d2785	117	24	3	1	65	24	1	1	0	0	0	1	13
182	09324debf75618ac	117	24	3	1	68	21	2	1	0	0	0	1	13
183	0e42a6c3fbd97158	109	26	3	1	62	24	2	1	0	0	0	1	11
184	047bafe9d8c36251	109	26	3	1	62	24	2	1	0	0	0	1	11
185	041dbea5267fc983	109	26	3	1	62	24	2	1	0	0	0	1	11
186	0481eb7d62f3c95a	109	26	3	1	62	24	2	1	0	0	0	1	11
187	0ce53b91f7d4a286	109	26	3	1	62	24	2	1	0	0	0	1	12
188	0a3b29c5fe4d6781	109	26	3	1	62	24	2	1	0	0	0	1	12
189	0ea342c6fb78d951	117	24	3	1	80	9	6	1	0	0	0	1	12
190	0285ca4e9f36db71	109	26	3	1	74	12	6	1	0	0	0	1	11
191	06c18a4edf329b75	109	26	3	1	74	12	6	1	0	0	0	1	11
192	08e64c29d7f51b3a	93	30	3	1	65	15	7	1	0	0	0	1	10
193	08e6c4a1d7f593b2	93	30	3	1	65	15	7	1	0	0	0	1	10
194	08a1e2c6f7d4b395	116	22	4	1	60	21	6	0	0	0	0	1	12
195	0e6c84a17fd5b392	116	22	4	1	60	21	6	0	0	0	0	1	13
196	0d786e5c2a1fb349	116	22	4	1	60	21	6	0	0	0	0	1	13
197	0938e64bf75ca21d	116	22	4	1	63	18	7	0	0	0	0	1	12
198	08a1e6c3f7d4b295	116	22	4	1	66	15	8	0	0	0	0	1	12
199	08a1e6d3f7c5b294	116	22	4	1	66	15	8	0	0	0	0	1	12
200	0c6348aef71259bd	116	22	4	1	59	24	3	1	0	0	0	1	12

TABLE 5.7: Implementations of the affine equivalence classes 201–250

	Representative	$\lvert c\rvert = 1/4$	1/2	3/4	1	$p = 1/8$	1/4	3/8	1/2	5/8	3/4	7/8	1	cost
201	08e64c2bd7f5193a	108	24	4	1	56	24	4	1	0	0	0	1	11
202	08ce4623f5d791ba	108	24	4	1	56	24	4	1	0	0	0	1	11
203	024ce6a97f5d1b38	108	24	4	1	56	24	4	1	0	0	0	1	11
204	04a8ec23dbf95176	108	24	4	1	56	24	4	1	0	0	0	1	11
205	0ac46e2d1b397f58	108	24	4	1	62	18	6	1	0	0	0	1	11
206	08a17b95f3d2c6e4	108	24	4	1	62	18	6	1	0	0	0	1	11
207	0759aec8fbd26341	108	24	4	1	62	18	6	1	0	0	0	1	11
208	08297d5a4cefb316	108	24	4	1	62	18	6	1	0	0	0	1	11
209	02a846c397b1f5de	108	24	4	1	62	18	6	1	0	0	0	1	12
210	06ac24e397b5d1f8	108	24	4	1	62	18	6	1	0	0	0	1	12
211	0ac62e83b795f1d4	108	24	4	1	62	18	6	1	0	0	0	1	12
212	0d98ea65fb7341c2	116	22	4	1	73	12	5	2	0	0	0	1	12
213	0283de7bcf659a14	108	24	4	1	67	15	5	2	0	0	0	1	11
214	0392ce7bdf648a15	108	24	4	1	67	15	5	2	0	0	0	1	11
215	0281df7ace459b36	92	28	4	1	48	30	0	3	0	0	0	1	9
216	086f5d7e4c29b31a	92	28	4	1	48	30	0	3	0	0	0	1	9
217	0283de7bcf459a16	115	20	5	1	56	21	6	1	0	0	0	1	12
218	0829f75ae64db31c	107	22	5	1	52	24	4	2	0	0	0	1	9
219	0823b79ad5f6c4e1	107	22	5	1	52	24	4	2	0	0	0	1	10
220	08a2d5f3e7c691b4	115	20	5	1	64	15	6	2	0	0	0	1	12
221	08a2c4e597b1d3f6	107	22	5	1	58	18	6	2	0	0	0	1	11
222	08e6c4a197f5d3b2	107	22	5	1	58	18	6	2	0	0	0	1	11
223	08a64ce75df1b293	114	18	6	1	63	6	15	0	0	0	0	1	12
224	0462e8c3715bf9da	114	18	6	1	63	6	15	0	0	0	0	1	13
225	08465dbce7a291f3	114	18	6	1	54	21	4	3	0	0	0	1	12
226	08a1d5f3c4e6b297	114	18	6	1	54	21	4	3	0	0	0	1	12
227	0abd4ef9823657c1	106	20	6	1	54	18	6	3	0	0	0	1	10
228	048c62e315f97bda	114	18	6	1	69	6	9	3	0	0	0	1	13
229	0823d7fa4ce591b6	106	20	6	1	63	9	9	3	0	0	0	1	11
230	0c69a24ef718b35d	113	16	7	1	62	9	8	4	0	0	0	1	11
231	0938e65bf74ca21d	110	10	10	1	65	0	5	10	0	0	0	1	11
232	092e7456cdfb83a1	94	32	2	1	66	23	1	0	1	0	0	1	11
233	03fc56ed47928a1b	94	32	2	1	66	23	1	0	1	0	0	1	11
234	097e4d6f5c38a21b	109	26	3	1	66	23	1	0	1	0	0	1	12
235	06ac8e239fbd5174	92	28	4	1	60	17	7	0	1	0	0	1	10
236	06ac8e219fbd7354	92	28	4	1	60	17	7	0	1	0	0	1	10
237	08e64c29f7d51b3a	107	22	5	1	48	29	3	0	1	0	0	1	11
238	0921b3d5f7a86e4c	107	22	5	1	48	29	3	0	1	0	0	1	12
239	08e64c29d5f71b3a	107	22	5	1	54	23	5	0	1	0	0	1	11
240	0ac46e29f7d53b18	107	22	5	1	60	17	7	0	1	0	0	1	11
241	08a719b35df6e2c4	105	18	7	1	58	11	9	2	1	0	0	1	11
242	0a23f7d86ec591b4	105	18	7	1	58	11	9	2	1	0	0	1	11
243	086d5f7ec4291b3a	62	40	2	1	48	33	0	0	0	1	0	1	9
244	08e64c2bf7d5193a	90	24	6	1	42	27	6	0	0	1	0	1	10
245	084a6e1d5c397f2b	112	28	0	2	57	21	7	0	0	0	0	1	12
246	08ab193246cf7d5e	96	32	0	2	51	21	9	0	0	0	0	1	10
247	0ba981234fe6dc57	112	28	0	2	50	30	2	1	0	0	0	1	11
248	0c2f1d7a48693b5e	96	32	0	2	44	30	4	1	0	0	0	1	9
249	012b89f7cde654a3	96	32	0	2	44	30	4	1	0	0	0	1	9
250	082b195d4f6e7c3a	96	32	0	2	44	30	4	1	0	0	0	1	10

TABLE 5.8: Implementations of the affine equivalence classes 251–300

	Representative	$\lvert c\rvert = 1/4$	1/2	3/4	1	$p = 1/8$	1/4	3/8	1/2	5/8	3/4	7/8	1	cost
251	024513768ecbf9da	96	32	0	2	44	30	4	1	0	0	0	1	10
252	0c7b3e6a2f4d5918	96	32	0	2	44	30	4	1	0	0	0	1	10
253	086f5d7e4c293b1a	64	40	0	2	32	36	0	4	0	0	0	1	7
254	086f5d7e4c2391ba	64	40	0	2	32	36	0	4	0	0	0	1	7
255	082b197c4e6d5f3a	64	40	0	2	32	36	0	4	0	0	0	1	8
256	046173528cebd9fa	64	40	0	2	32	36	0	4	0	0	0	1	8
257	086f5d7ec4a1b392	64	40	0	2	32	36	0	4	0	0	0	1	8
258	08a319f6c4e7d5b2	0	56	0	2	64	0	0	14	0	0	0	1	7
259	09f75d26183ac4eb	110	24	2	2	45	21	11	0	0	0	0	1	11
260	08a357dfb192c4e6	110	24	2	2	50	18	10	1	0	0	0	1	11
261	08a71df395b2c4e6	94	28	2	2	40	24	8	2	0	0	0	1	9
262	0459afebd8c16273	94	28	2	2	40	24	8	2	0	0	0	1	9
263	0812b3a95d46f7ce	94	28	2	2	40	24	8	2	0	0	0	1	9
264	04369ca78def512b	110	24	2	2	48	24	4	3	0	0	0	1	11
265	032547618bacfe9d	108	20	4	2	44	12	16	1	0	0	0	1	10
266	08297f5a6e4d3b1c	92	24	4	2	28	30	4	5	0	0	0	1	7
267	082b193a4ce5f7d6	92	24	4	2	28	30	4	5	0	0	0	1	8
268	082b3f1a5d7e4c69	92	24	4	2	28	30	4	5	0	0	0	1	8
269	0461dbf28ce9537a	56	28	8	1	0	56	0	0	0	0	0	2	9
270	092e1436cdfb85a7	96	32	0	2	48	29	3	0	1	0	0	1	11
271	082b1a6d5f7e4c39	96	32	0	2	48	29	3	0	1	0	0	1	11
272	046f953b1d7ec82a	110	24	2	2	36	35	3	0	1	0	0	1	11
273	0a28c6e53b91f7d4	92	24	4	2	40	17	11	2	1	0	0	1	9
274	082a4ce51b39d7f6	92	24	4	2	40	17	11	2	1	0	0	1	9
275	0a28c4e53b91f7d6	106	16	6	2	32	23	7	4	1	0	0	1	10
276	082ac4e519b3d7f6	104	12	8	2	42	11	5	9	1	0	0	1	10
277	082b7c5a496e3f1d	108	20	4	2	58	16	0	5	2	0	0	1	11
278	08a75db391f6c4e2	92	24	4	2	46	22	0	5	2	0	0	1	9
279	086e4c295d7f1b3a	90	20	6	2	42	9	15	0	3	0	0	1	9
280	082a4ce7193bf5d6	104	12	8	2	24	32	0	3	4	0	0	1	10
281	082a4ce5193bd7f6	104	12	8	2	48	8	8	3	4	0	0	1	10
282	082ac4e319b7d5f6	100	4	12	2	54	0	0	9	6	0	0	1	10
283	086e4c295d7f3b1a	62	36	2	2	36	21	12	0	0	1	0	1	8
284	04ae8c239dbf5176	62	36	2	2	36	21	12	0	0	1	0	1	8
285	08a35df2c4e791b6	60	32	4	2	32	27	0	7	0	1	0	1	7
286	08e64c2bd5f7193a	90	20	6	2	24	27	8	3	0	1	0	1	8
287	0823d5fa4ce791b6	88	16	8	2	38	13	8	4	2	1	0	1	8
288	046153728ce9dbfa	0	56	0	2	0	56	0	0	0	0	0	2	7
289	046b59728ce3d1fa	0	56	0	2	0	56	0	0	0	0	0	2	7
290	0463d9f28ceb517a	56	24	8	2	0	44	0	6	0	0	0	2	7
291	0127456389aedcbf	96	24	0	4	24	6	24	3	0	0	0	1	8
292	081b2a394c5e7f6d	64	32	0	4	0	36	0	12	0	0	0	1	6
293	0c2f1d7b483a596e	96	24	0	4	12	38	0	3	4	0	0	1	9
294	082ac4e719b3d5f6	88	8	8	4	12	20	0	9	4	2	0	1	7
295	086e4c2b5d7f193a	60	24	4	4	16	9	16	5	0	3	0	1	6
296	082b5d7f193e4c6a	84	0	12	4	36	3	0	0	12	3	0	1	7
297	082b197e4c6f5d3a	0	48	0	4	0	32	0	12	0	0	0	2	5
298	046351728cebd9fa	0	48	0	4	0	32	0	12	0	0	0	2	5
299	082b5d7a4c6f193e	56	16	8	4	0	26	0	12	0	2	0	2	5
300	082a4c6f193b5d7e	56	0	8	8	0	14	0	0	0	14	0	2	4

TABLE 5.9: Implementations of the affine equivalence classes 300–302

	Representative	$\lvert c\rvert = 1/4$	1/2	3/4	1	$p = 1/8$	1/4	3/8	1/2	5/8	3/4	7/8	1	cost
301	`082b193a4c6f5d7e`	0	32	0	8	0	0	0	24	0	0	0	4	3
302	`0123456789abcdef`	0	0	0	16	0	0	0	0	0	0	0	16	0

Chapter 6

Conclusion

We conclude this thesis by summarising the main achievements and results. We also suggest ideas for further research.

6.1 Achievements of this thesis

In this master thesis the problem of designing and efficiently implementing s-boxes has been treated.

- We presented the algorithms used to search through all possible sequences of instructions, to classify the s-boxes and to manage large amounts of data.
- We introduced concepts to parallelise the algorithm on different platforms. We decided for a sequential approach which could be speeded up by the support of worker threads. The efficiency of the method has been discussed and improvements haven been proposed.
- The efficiency has been further increased by optimising on the algorithmic level. A modified equivalence algorithm has been introduced which could reduce the branching factor by more advanced caching. All nodes that are equivalent with a previous one will be skipped from further investigations. Reducing the branching factor will speed up the program exponentially.
- The algorithms have been applied for a basic architecture with 5 registers. The time after finishing the code was not sufficient to find all 302 classes. By the time of handing in the thesis 272 classes have been found. The classes found cover about 90% of all s-boxes. The results have been compared with literature and s-boxes of selected primitives. We also investigated properties of the representatives, that are variant within a class. Further we looked at the implementations of the representatives and could show that none of them makes use of the NOT instruction.
- A new approach for designing cryptographic primitives has been introduced. We propose to choose the s-box, other than in earlier approaches, in a first

step. Therefore an optimal implementation of an s-box with certain, non-linear properties is selected from a table as created in this thesis. The linear layer is then designed such that the other specifications are fulfilled.

6.2 Further work

We believe that the approach introduced has a huge potential. There are many ideas how to further extend and improve the algorithm.

- The search for the best implementation for s-boxes is still not completely finished even though it is already advanced. The simulation is still running and we will update our results as soon as it will be finished.
- In section 4.3 problems of the chosen parallelisation strategy have been shown. Redesigning the parallelisation could be combined with porting the program on a cluster architecture. This task may be required for certain extensions proposed in this chapter.
- The simulation of this work was limited to a basic instruction set. Modern processors support many more advanced features such as , multiple ALUs, pipelining and extended instruction sets. All those features could result different trade-offs when designing s-boxes.
- The definition of instructions could be made in a much more generic way. Rather than explicitly defining the instruction in the code, it could be defined as generic lookup table for any 5×5-bit function. This could result in an interesting new approach for hardware implementations. An already implemented s-box can be reused as one single instruction and the search algorithm can find ways to implement an s-box by reusing another s-box on the same chip.
- It has been shown that the relationship between implementation cost and non-linear properties is not monotone is. This can be used to find optimal trade-offs between implementations cost, non-linear properties and the number of rounds in a cipher.
- In section 5.4, we propose a design methodology for cryptographic primitives. The problem of interaction between linear and substitution layer has not been fully investigated. It is not proven if the most efficient members, which may have same some lacks in variant properties, can be compensated by the linear layer such that the cipher is more efficient than if one would have designed according to traditional design strategies. We suggest to investigate refinements of the design strategy.

Appendices

Appendix A

Most efficient implementations

S-box 9	S-box 10	S-box 11	S-box 13	S-box 14	S-box 15
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2
3 OR r2 r1	3 AND r2 r1	3 AND r2 r0	3 OR r2 r1	3 OR r2 r1	3 OR r2 r1
4 XOR r2 r3	4 XOR r2 r3	4 XOR r2 r3	4 XOR r2 r3	4 XOR r1 r0	4 XOR r2 r3
5 OR r3 r0	5 AND r3 r2	5 OR r3 r0	5 AND r3 r0	5 XOR r2 r3	5 OR r3 r0
6 XOR r3 r4	6 XOR r3 r0	6 XOR r3 r1	6 XOR r3 r4	6 AND r3 r0	6 AND r1 r3
7 AND r4 r3	7 AND r0 r2	7 AND r1 r2	7 AND r4 r2	7 XOR r3 r4	7 XOR r3 r4
8 XOR r4 r2	8 XOR r0 r4	8 XOR r1 r4	8 XOR r1 r4	8 AND r4 r2	8 AND r4 r2
9 AND r2 r3	9 OR r4 r0	9 OR r4 r2	r0 r1 r2 r3	9 XOR r0 r4	9 XOR r1 r4
10 XOR r1 r2	10 AND r4 r3	10 XOR r4 r0		r0 r1 r2 r3	r0 r1 r2 r3
r0 r1 r3 r4	11 XOR r1 r4	11 AND r0 r1			
	r0 r1 r2 r3	12 XOR r0 r2			
		r0 r1 r3 r4			

S-box 16	S-box 18	S-box 19	S-box 20	S-box 21	S-box 25
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2
3 AND r2 r1	3 AND r2 r0	3 OR r2 r1	3 OR r2 r1	3 AND r2 r0	3 OR r2 r1
4 XOR r2 r3	4 XOR r2 r3	4 XOR r1 r0	4 XOR r1 r3	4 XOR r2 r3	4 XOR r1 r3
5 OR r3 r1	5 OR r3 r0	5 XOR r2 r3	5 AND r1 r0	5 OR r3 r0	5 AND r1 r0
6 XOR r3 r0	6 XOR r3 r1	6 AND r3 r0	6 XOR r2 r1	6 XOR r3 r1	6 XOR r1 r2
7 OR r0 r2	7 AND r1 r2	7 XOR r3 r4	7 XOR r1 r4	7 AND r1 r0	7 XOR r3 r4
8 XOR r0 r4	8 XOR r1 r4	8 AND r4 r2	8 AND r4 r2	8 XOR r1 r4	8 XOR r2 r3
9 AND r4 r2	9 OR r4 r0	9 XOR r4 r1	9 XOR r4 r3	9 AND r4 r3	9 AND r3 r1
10 XOR r1 r4	10 AND r4 r3	10 AND r1 r3	10 AND r3 r1	10 XOR r4 r2	10 XOR r3 r4
r0 r1 r2 r3	11 XOR r2 r4	11 XOR r0 r1	11 XOR r0 r3	11 OR r2 r3	r0 r1 r2 r3
	r0 r1 r2 r3	r0 r2 r3 r4	r0 r1 r2 r4	12 XOR r0 r2	
				r0 r1 r3 r4	

S-box 26	S-box 27	S-box 28	S-box 29	S-box 30	S-box 31
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 XOR r0 r2	2 OR r1 r4	2 OR r1 r4	2 OR r1 r4	2 XOR r0 r2	2 XOR r0 r2
3 OR r2 r1	3 XOR r0 r2	3 XOR r0 r2	3 XOR r0 r2	3 OR r2 r1	3 AND r2 r1
4 XOR r2 r3	4 AND r2 r4	4 AND r2 r4	4 AND r2 r4	4 XOR r2 r4	4 XOR r2 r3
5 AND r3 r0	5 XOR r0 r3	5 XOR r4 r0	5 XOR r4 r0	5 OR r4 r0	5 OR r3 r1
6 XOR r3 r1	6 XOR r4 r0	6 XOR r0 r3	6 XOR r0 r3	6 XOR r3 r4	6 XOR r3 r4
7 AND r1 r2	7 AND r0 r1	7 AND r0 r1	7 AND r3 r1	7 AND r4 r3	7 AND r4 r2
8 XOR r1 r4	8 OR r0 r2	8 OR r0 r2	8 OR r3 r2	8 XOR r4 r2	8 XOR r4 r0
9 OR r4 r3	9 XOR r2 r3	9 XOR r2 r3	9 XOR r2 r0	9 AND r2 r3	9 AND r0 r2
10 XOR r2 r4	10 AND r3 r0	10 AND r3 r0	10 AND r0 r3	10 XOR r1 r2	10 XOR r0 r1
r0 r1 r2 r3	11 XOR r1 r3	11 XOR r1 r3	11 XOR r0 r1	r0 r1 r3 r4	r0 r2 r3 r4
	r0 r1 r2 r4	r0 r1 r2 r4	r0 r2 r3 r4		

S-box 32	S-box 33	S-box 37	S-box 38	S-box 39	S-box 40
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 AND r0 r2	2 XOR r0 r2	2 XOR r0 r2
3 AND r2 r1	3 OR r2 r1	3 AND r2 r1	3 XOR r0 r3	3 AND r2 r0	3 AND r2 r0
4 XOR r2 r3	4 XOR r2 r3	4 XOR r2 r3	4 OR r3 r1	4 XOR r1 r0	4 XOR r2 r3
5 OR r3 r1	5 AND r3 r0	5 AND r3 r2	5 XOR r3 r2	5 XOR r2 r3	5 OR r3 r0
6 XOR r3 r4	6 XOR r3 r4	6 XOR r3 r0	6 AND r2 r0	6 OR r3 r0	6 XOR r3 r1
7 AND r4 r2	7 AND r4 r0	7 AND r0 r2	7 XOR r2 r4	7 XOR r3 r4	7 AND r1 r2
8 XOR r1 r4	8 XOR r1 r4	8 XOR r0 r4	8 AND r4 r3	8 OR r4 r3	8 XOR r0 r1
r0 r1 r2 r3	r0 r1 r2 r3	9 OR r4 r2	9 XOR r4 r1	9 XOR r2 r3	9 AND r1 r0
		10 XOR r1 r4	10 OR r1 r2	10 XOR r4 r1	10 XOR r1 r4
		r0 r1 r2 r3	11 XOR r0 r1	11 AND r1 r2	11 AND r4 r3
			r0 r2 r3 r4	12 XOR r0 r1	12 XOR r2 r4
				r0 r2 r3 r4	r0 r1 r2 r3
S-box 43	**S-box 44**	**S-box 45**	**S-box 46**	**S-box 51**	**S-box 52**
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 XOR r0 r2	2 XOR r0 r2	2 XOR r2 r0	2 XOR r0 r2	2 XOR r0 r2	2 XOR r2 r0
3 AND r2 r1	3 OR r2 r1	3 AND r0 r2	3 AND r2 r0	3 AND r2 r0	3 AND r0 r2
4 OR r2 r3	4 XOR r2 r3	4 XOR r0 r3	4 XOR r1 r0	4 XOR r2 r3	4 XOR r0 r3
5 XOR r2 r4	5 OR r3 r0	5 OR r3 r2	5 XOR r2 r3	5 OR r3 r0	5 OR r3 r2
6 XOR r4 r0	6 XOR r4 r2	6 XOR r1 r2	6 OR r3 r0	6 XOR r3 r1	6 XOR r3 r1
7 AND r0 r3	7 XOR r3 r4	7 XOR r3 r4	7 XOR r1 r2	7 AND r1 r2	7 AND r1 r0
8 XOR r0 r1	8 AND r4 r3	8 AND r4 r0	8 XOR r3 r4	8 XOR r1 r4	8 XOR r1 r4
9 MOV r1 r0	9 XOR r4 r2	9 XOR r4 r1	9 AND r4 r3	9 AND r4 r3	9 AND r4 r3
10 AND r0 r2	10 AND r2 r3	10 OR r1 r3	10 XOR r4 r1	10 XOR r2 r4	10 XOR r0 r4
11 XOR r0 r3	11 XOR r1 r2	11 XOR r1 r2	11 OR r1 r3	r0 r1 r2 r3	r0 r1 r2 r3
r0 r1 r2 r4	r0 r1 r3 r4	r0 r1 r3 r4	12 XOR r0 r1		
			r0 r2 r3 r4		
S-box 53	**S-box 54**	**S-box 55**	**S-box 56**	**S-box 57**	**S-box 58**
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2
3 AND r2 r0	3 AND r2 r0	3 AND r2 r0	3 AND r2 r0	3 AND r2 r0	3 AND r2 r1
4 XOR r0 r1	4 XOR r1 r0	4 XOR r1 r0	4 XOR r1 r0	4 XOR r2 r3	4 OR r2 r4
5 XOR r2 r3	5 XOR r2 r3	5 XOR r2 r3	5 XOR r2 r3	5 OR r3 r0	5 XOR r2 r3
6 AND r3 r0	6 OR r3 r0	6 OR r3 r0	6 OR r3 r0	6 XOR r0 r1	6 OR r3 r0
7 XOR r3 r4	7 XOR r3 r1	7 XOR r3 r1	7 XOR r3 r4	7 XOR r3 r4	7 XOR r1 r3
8 AND r4 r3	8 AND r1 r2	8 AND r1 r2	8 AND r4 r2	8 AND r4 r2	8 AND r3 r1
9 XOR r4 r2	9 XOR r1 r4	9 XOR r1 r4	9 XOR r4 r1	9 XOR r4 r1	9 XOR r3 r2
10 AND r2 r3	10 AND r4 r1	10 OR r4 r1	10 AND r1 r3	10 AND r1 r3	10 AND r2 r1
11 XOR r1 r2	11 XOR r0 r4	11 XOR r0 r4	11 XOR r0 r1	11 XOR r1 r2	11 XOR r2 r4
r0 r1 r3 r4	r0 r1 r2 r3	r0 r1 r2 r3	r0 r2 r3 r4	r0 r1 r3 r4	r0 r1 r2 r3
S-box 59	**S-box 60**	**S-box 61**	**S-box 62**	**S-box 63**	**S-box 64**
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 XOR r0 r2	2 XOR r0 r2	2 XOR r1 r2	2 XOR r1 r2	2 AND r0 r2	2 AND r0 r2
3 OR r2 r1	3 OR r2 r1	3 OR r1 r4	3 OR r1 r4	3 XOR r0 r3	3 XOR r0 r3
4 XOR r1 r0	4 XOR r1 r0	4 XOR r3 r0	4 XOR r4 r0	4 OR r3 r1	4 OR r3 r1
5 XOR r2 r3	5 XOR r3 r2	5 XOR r4 r3	5 XOR r0 r3	5 XOR r1 r2	5 XOR r2 r0
6 AND r3 r0	6 AND r2 r3	6 AND r3 r1	6 AND r3 r1	6 AND r2 r0	6 XOR r2 r3
7 XOR r3 r4	7 XOR r2 r4	7 XOR r1 r2	7 XOR r1 r2	7 XOR r2 r3	7 AND r3 r2
8 AND r4 r2	8 OR r4 r3	8 XOR r1 r3	8 XOR r1 r3	8 XOR r3 r4	8 XOR r3 r4
9 XOR r1 r4	9 XOR r4 r0	9 OR r3 r0	9 OR r3 r4	9 AND r4 r2	9 OR r4 r2
10 AND r4 r1	10 AND r0 r2	10 OR r2 r3	10 OR r2 r3	10 XOR r4 r1	10 XOR r4 r0
11 XOR r0 r4	11 XOR r0 r3	11 XOR r0 r2	11 XOR r2 r4	11 AND r1 r3	11 AND r0 r3
r0 r1 r2 r3	r0 r1 r2 r4	r0 r1 r3 r4	r0 r1 r2 r3	12 XOR r0 r1	12 XOR r0 r1
				r0 r2 r3 r4	r0 r2 r3 r4
S-box 65	**S-box 66**	**S-box 67**	**S-box 68**	**S-box 69**	**S-box 70**
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 AND r0 r2	2 AND r0 r2	2 OR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2
3 XOR r0 r3	3 XOR r0 r3	3 XOR r0 r3	3 AND r2 r0	3 AND r2 r0	3 AND r2 r0
4 OR r3 r1	4 OR r3 r1	4 AND r3 r2	4 XOR r2 r3	4 XOR r2 r3	4 XOR r2 r3
5 XOR r2 r0	5 XOR r3 r2	5 XOR r1 r0	5 OR r3 r0	5 OR r3 r0	5 OR r3 r0
6 XOR r2 r3	6 AND r2 r0	6 XOR r3 r1	6 XOR r0 r1	6 XOR r0 r2	6 XOR r0 r2
7 AND r3 r2	7 XOR r2 r1	7 AND r1 r3	7 XOR r3 r4	7 XOR r1 r0	7 XOR r3 r1
8 XOR r3 r4	8 XOR r2 r4	8 XOR r1 r4	8 AND r4 r2	8 XOR r3 r4	8 AND r1 r0
9 OR r4 r2	9 OR r4 r2	9 OR r4 r3	9 XOR r1 r4	9 AND r4 r2	9 XOR r1 r4
10 XOR r4 r0	10 XOR r0 r4	10 XOR r4 r0	10 XOR r2 r0	10 XOR r4 r1	10 AND r4 r3
11 OR r0 r3	11 OR r4 r3	11 AND r0 r1	11 AND r0 r3	11 AND r1 r3	11 OR r4 r2
12 XOR r0 r1	12 XOR r1 r4	12 XOR r0 r2	12 OR r0 r4	12 XOR r1 r2	12 XOR r0 r4
r0 r2 r3 r4	r0 r1 r2 r3	r0 r1 r3 r4	r0 r1 r2 r3	r0 r1 r3 r4	r0 r1 r2 r3

S-box 71	S-box 72	S-box 73	S-box 77	S-box 79	S-box 80
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 OR r0 r2	2 AND r0 r2	2 OR r0 r2
3 AND r2 r0	3 AND r2 r0	3 AND r2 r0	3 XOR r0 r3	3 XOR r0 r3	3 XOR r0 r3
4 XOR r2 r3	4 XOR r2 r3	4 XOR r2 r3	4 AND r3 r2	4 OR r3 r1	4 OR r3 r1
5 OR r3 r0	5 OR r3 r0	5 OR r3 r0	5 XOR r1 r0	5 XOR r1 r2	5 AND r2 r3
6 XOR r3 r1	6 XOR r3 r1	6 XOR r3 r1	6 XOR r3 r1	6 AND r2 r0	6 XOR r3 r4
7 AND r1 r0	7 AND r1 r2	7 AND r1 r2	7 AND r1 r3	7 XOR r2 r3	7 AND r4 r0
8 XOR r0 r4	8 XOR r1 r4	8 XOR r1 r4	8 XOR r1 r4	8 XOR r3 r4	8 XOR r4 r2
9 AND r4 r3	9 AND r4 r3	9 AND r4 r3	9 OR r4 r3	9 AND r4 r2	9 OR r2 r0
10 OR r1 r4	10 XOR r2 r4	10 XOR r4 r2	10 XOR r4 r0	10 XOR r1 r4	10 XOR r1 r2
11 OR r4 r2	11 AND r4 r2	11 AND r2 r1	11 OR r0 r1	r0 r1 r2 r3	r0 r1 r3 r4
12 XOR r0 r4	12 XOR r0 r4	12 XOR r0 r2	12 XOR r0 r2		
r0 r1 r2 r3	r0 r1 r2 r3	r0 r1 r3 r4	r0 r1 r3 r4		
S-box 81	**S-box 82**	**S-box 83**	**S-box 84**	**S-box 85**	**S-box 86**
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 XOR r0 r2	2 XOR r0 r2	2 AND r0 r2	2 AND r0 r2	2 XOR r0 r2	2 OR r0 r2
3 OR r2 r1	3 OR r2 r1	3 XOR r0 r3	3 XOR r0 r3	3 OR r2 r1	3 AND r2 r1
4 XOR r2 r3	4 XOR r2 r3	4 OR r3 r1	4 OR r3 r1	4 XOR r1 r0	4 XOR r0 r3
5 AND r3 r0	5 XOR r2 r4	5 AND r1 r0	5 XOR r3 r2	5 XOR r2 r3	5 OR r3 r1
6 XOR r0 r4	6 AND r4 r2	6 XOR r1 r2	6 AND r2 r0	6 AND r3 r0	6 XOR r0 r4
7 OR r4 r3	7 OR r4 r0	7 AND r2 r0	7 XOR r2 r4	7 XOR r3 r4	7 XOR r1 r0
8 XOR r4 r1	8 XOR r4 r3	8 XOR r2 r3	8 AND r4 r0	8 AND r4 r3	8 AND r0 r3
9 AND r1 r2	9 AND r3 r2	9 XOR r3 r4	9 XOR r1 r4	9 XOR r4 r1	9 OR r0 r2
10 OR r1 r3	10 XOR r1 r3	10 AND r4 r2	r0 r1 r2 r3	10 OR r1 r3	10 XOR r3 r4
r0 r1 r2 r4	r0 r1 r2 r4	11 XOR r0 r4		11 XOR r0 r1	11 OR r4 r0
		r0 r1 r2 r3		r0 r2 r3 r4	12 XOR r2 r4
					r0 r1 r2 r3
S-box 87	**S-box 88**	**S-box 89**	**S-box 90**	**S-box 91**	**S-box 93**
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 OR r0 r2	2 OR r0 r2	2 OR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2
3 AND r2 r1	3 XOR r0 r3	3 XOR r0 r3	3 AND r2 r0	3 AND r2 r0	3 OR r2 r1
4 XOR r4 r0	4 AND r3 r2	4 OR r3 r1	4 XOR r1 r0	4 XOR r1 r0	4 AND r2 r3
5 OR r1 r4	5 XOR r1 r0	5 XOR r4 r0	5 XOR r2 r3	5 XOR r2 r3	5 OR r3 r0
6 XOR r4 r3	6 XOR r1 r3	6 XOR r3 r4	6 AND r3 r1	6 OR r3 r0	6 XOR r1 r3
7 AND r3 r1	7 AND r3 r1	7 AND r4 r3	7 XOR r0 r2	7 XOR r3 r4	7 XOR r2 r4
8 XOR r0 r3	8 XOR r3 r4	8 XOR r4 r2	8 XOR r3 r4	8 XOR r1 r3	8 XOR r3 r2
9 OR r3 r2	9 OR r4 r1	9 AND r2 r3	9 AND r4 r3	9 AND r3 r1	9 AND r3 r1
10 XOR r2 r0	10 XOR r0 r4	10 XOR r0 r2	10 XOR r4 r2	10 XOR r0 r3	10 XOR r3 r4
11 AND r0 r3	11 OR r4 r0	11 AND r2 r0	11 OR r2 r3	11 OR r3 r2	r0 r1 r2 r3
12 XOR r0 r1	12 XOR r2 r4	12 XOR r1 r2	12 XOR r1 r2	12 XOR r3 r4	
r0 r2 r3 r4	r0 r1 r2 r3	r0 r1 r3 r4	r0 r1 r3 r4	r0 r1 r2 r3	
S-box 94	**S-box 95**	**S-box 96**	**S-box 97**	**S-box 98**	**S-box 99**
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 OR r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 XOR r0 r2	2 AND r0 r2	2 AND r0 r2	2 OR r0 r2	2 XOR r0 r2	2 XOR r0 r2
3 AND r2 r1	3 XOR r0 r3	3 XOR r0 r3	3 AND r2 r1	3 AND r2 r0	3 AND r2 r0
4 OR r2 r3	4 OR r3 r1	4 OR r3 r1	4 XOR r4 r0	4 XOR r1 r0	4 XOR r1 r0
5 XOR r4 r2	5 XOR r3 r2	5 XOR r2 r0	5 OR r1 r4	5 XOR r2 r3	5 XOR r2 r3
6 AND r2 r0	6 OR r2 r1	6 XOR r2 r3	6 XOR r0 r3	6 AND r3 r1	6 AND r3 r1
7 XOR r1 r2	7 XOR r2 r4	7 AND r3 r2	7 OR r3 r2	7 XOR r3 r4	7 XOR r3 r4
8 MOV r2 r1	8 AND r4 r3	8 XOR r3 r4	8 XOR r2 r0	8 MOV r4 r2	8 MOV r4 r2
9 AND r1 r4	9 XOR r1 r4	9 OR r4 r2	9 XOR r4 r3	9 AND r2 r3	9 OR r2 r3
10 XOR r1 r3	10 AND r4 r0	10 XOR r0 r4	10 AND r3 r1	10 XOR r0 r2	10 XOR r0 r2
r0 r1 r2 r4	11 XOR r2 r4	11 OR r4 r0	11 AND r0 r3	11 AND r2 r0	11 AND r2 r0
	r0 r1 r2 r3	12 XOR r1 r4	12 XOR r0 r1	12 XOR r1 r2	12 XOR r1 r2
		r0 r1 r2 r3	r0 r2 r3 r4	r0 r1 r3 r4	r0 r1 r3 r4

S-box 100	S-box 101	S-box 102	S-box 103	S-box 104	S-box 106
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2
3 AND r2 r0	3 AND r2 r0	3 AND r2 r0	3 AND r2 r0	3 AND r2 r0	3 AND r2 r0
4 XOR r1 r0	4 XOR r1 r3	4 XOR r1 r3	4 XOR r2 r3	4 XOR r2 r3	4 XOR r0 r1
5 XOR r2 r3	5 XOR r2 r1	5 XOR r2 r1	5 AND r3 r1	5 OR r3 r0	5 XOR r2 r3
6 OR r3 r0	6 AND r1 r0	6 AND r1 r0	6 OR r3 r0	6 XOR r0 r1	6 AND r3 r0
7 XOR r1 r3	7 XOR r1 r3	7 XOR r1 r4	7 XOR r3 r4	7 AND r0 r3	7 OR r1 r3
8 XOR r3 r4	8 XOR r1 r4	8 XOR r3 r1	8 AND r4 r0	8 XOR r0 r4	8 XOR r3 r4
9 OR r3 r1	9 AND r3 r1	9 AND r1 r3	9 OR r4 r2	9 AND r4 r2	9 OR r4 r2
10 XOR r0 r3	10 XOR r0 r3	10 XOR r0 r1	10 XOR r1 r4	10 XOR r3 r4	10 XOR r1 r4
11 OR r3 r2	11 OR r3 r2	11 OR r1 r2	11 AND r4 r3	11 AND r4 r0	r0 r1 r2 r3
12 XOR r3 r4	12 XOR r3 r4	12 XOR r1 r4	12 XOR r0 r4	12 XOR r1 r4	
r0 r1 r2 r3	r0 r1 r2 r3	r0 r1 r2 r3	r0 r1 r2 r3	r0 r1 r2 r3	
S-box 107	**S-box 108**	**S-box 109**	**S-box 110**	**S-box 111**	**S-box 112**
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 AND r0 r2	2 OR r0 r2
3 AND r2 r0	3 AND r2 r1	3 OR r2 r1	3 OR r2 r1	3 XOR r0 r3	3 XOR r0 r3
4 XOR r1 r0	4 XOR r2 r3	4 XOR r2 r3	4 XOR r2 r3	4 OR r3 r1	4 OR r3 r1
5 XOR r2 r3	5 AND r3 r2	5 AND r3 r0	5 AND r3 r0	5 XOR r3 r2	5 XOR r1 r0
6 AND r3 r1	6 XOR r3 r0	6 XOR r3 r4	6 XOR r3 r4	6 AND r2 r0	6 XOR r3 r4
7 XOR r3 r4	7 AND r0 r2	7 AND r4 r2	7 AND r4 r2	7 XOR r2 r4	7 OR r4 r2
8 MOV r4 r2	8 XOR r0 r4	8 XOR r0 r4	8 XOR r4 r1	8 AND r4 r2	8 AND r4 r0
9 AND r2 r3	9 OR r4 r0	9 AND r4 r0	9 OR r1 r3	9 XOR r4 r1	9 XOR r4 r1
10 XOR r1 r2	10 XOR r1 r4	10 XOR r1 r4	10 XOR r1 r2	10 OR r1 r2	10 OR r1 r3
r0 r1 r3 r4	r0 r1 r2 r3	r0 r1 r2 r3	r0 r1 r3 r4	11 XOR r0 r1	11 XOR r1 r2
				r0 r2 r3 r4	r0 r1 r3 r4
S-box 113	**S-box 114**	**S-box 115**	**S-box 116**	**S-box 119**	**S-box 120**
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 OR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 AND r0 r2	2 AND r0 r2
3 XOR r0 r3	3 AND r2 r0	3 OR r2 r1	3 AND r2 r0	3 XOR r0 r3	3 XOR r0 r3
4 OR r3 r1	4 XOR r2 r3	4 XOR r1 r3	4 XOR r1 r0	4 OR r3 r1	4 OR r3 r1
5 XOR r3 r4	5 OR r3 r0	5 AND r1 r0	5 XOR r2 r3	5 AND r3 r2	5 XOR r2 r3
6 OR r4 r1	6 XOR r1 r3	6 XOR r3 r4	6 OR r3 r0	6 XOR r3 r4	6 OR r3 r2
7 AND r4 r0	7 XOR r3 r2	7 XOR r0 r3	7 XOR r3 r4	7 AND r4 r0	7 XOR r3 r4
8 XOR r2 r3	8 OR r3 r1	8 AND r3 r2	8 OR r4 r3	8 OR r2 r4	8 AND r4 r0
9 XOR r4 r3	9 XOR r3 r4	9 XOR r2 r1	9 XOR r4 r1	9 XOR r4 r1	9 XOR r4 r2
10 OR r3 r0	10 AND r4 r1	10 OR r1 r3	10 AND r1 r3	10 AND r1 r3	10 AND r2 r3
11 XOR r1 r3	11 XOR r2 r4	11 XOR r3 r4	11 OR r1 r2	11 XOR r1 r2	11 XOR r1 r2
r0 r1 r2 r4	r0 r1 r2 r3	r0 r1 r2 r3	12 XOR r0 r1	r0 r1 r3 r4	r0 r1 r3 r4
			r0 r2 r3 r4		
S-box 121	**S-box 122**	**S-box 123**	**S-box 124**	**S-box 125**	**S-box 126**
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2
3 AND r2 r0	3 AND r2 r0	3 AND r2 r0	3 AND r2 r0	3 AND r2 r0	3 OR r2 r1
4 XOR r2 r3	4 XOR r2 r3	4 XOR r2 r3	4 XOR r2 r3	4 XOR r3 r2	4 XOR r4 r2
5 OR r3 r0	5 OR r3 r0	5 OR r3 r0	5 OR r3 r1	5 OR r2 r1	5 AND r2 r4
6 XOR r0 r3	6 XOR r3 r1	6 XOR r3 r1	6 AND r3 r2	6 AND r2 r3	6 XOR r2 r3
7 XOR r3 r1	7 AND r1 r0	7 AND r1 r0	7 XOR r3 r4	7 XOR r2 r4	7 AND r3 r0
8 OR r1 r0	8 XOR r1 r4	8 XOR r1 r4	8 OR r4 r2	8 OR r4 r3	8 XOR r1 r3
9 XOR r1 r4	9 AND r4 r3	9 OR r4 r2	9 XOR r4 r0	9 XOR r4 r0	9 MOV r3 r1
10 OR r4 r2	10 OR r4 r2	10 AND r4 r3	10 OR r0 r3	10 OR r0 r2	10 AND r1 r2
11 XOR r0 r4	11 XOR r0 r4	11 XOR r0 r4	11 XOR r0 r1	11 XOR r0 r1	11 XOR r0 r1
r0 r1 r2 r3	r0 r1 r2 r3	r0 r1 r2 r3	r0 r2 r3 r4	r0 r2 r3 r4	r0 r2 r3 r4
S-box 127	**S-box 128**	**S-box 129**	**S-box 130**	**S-box 131**	**S-box 132**
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 XOR r2 r0	2 AND r0 r2	2 AND r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2
3 AND r0 r2	3 XOR r0 r3	3 XOR r0 r3	3 AND r2 r0	3 AND r2 r0	3 AND r2 r0
4 XOR r0 r3	4 OR r3 r1	4 OR r3 r1	4 XOR r1 r0	4 XOR r1 r0	4 XOR r2 r3
5 OR r3 r2	5 XOR r1 r0	5 XOR r1 r2	5 XOR r2 r3	5 XOR r2 r3	5 OR r3 r0
6 AND r2 r0	6 XOR r3 r2	6 AND r2 r0	6 OR r3 r0	6 OR r3 r0	6 XOR r1 r2
7 XOR r2 r1	7 AND r2 r0	7 XOR r0 r4	7 XOR r1 r3	7 XOR r3 r4	7 XOR r3 r4
8 AND r1 r3	8 XOR r2 r4	8 AND r0 r3	8 OR r3 r1	8 AND r4 r3	8 AND r4 r0
9 XOR r1 r4	9 AND r4 r2	9 XOR r1 r0	9 XOR r3 r4	9 XOR r4 r1	9 XOR r4 r1
10 AND r4 r0	10 XOR r4 r1	10 OR r0 r2	10 XOR r4 r2	10 OR r1 r3	10 AND r1 r3
11 XOR r3 r4	11 OR r1 r2	11 XOR r2 r3	11 AND r4 r1	11 XOR r0 r1	11 XOR r0 r1
r0 r1 r2 r3	12 XOR r0 r1	12 XOR r3 r4	12 XOR r0 r4	r0 r2 r3 r4	r0 r2 r3 r4
	r0 r2 r3 r4	r0 r1 r2 r3	r0 r1 r2 r3		

S-box 133	S-box 134	S-box 135	S-box 136	S-box 137	S-box 138
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 XOR r1 r2	2 XOR r0 r2	2 XOR r0 r2	2 OR r0 r2	2 OR r0 r2	2 XOR r0 r2
3 OR r1 r0	3 OR r2 r1	3 OR r2 r1	3 XOR r0 r3	3 XOR r0 r3	3 AND r2 r0
4 XOR r1 r3	4 AND r2 r3	4 XOR r2 r3	4 OR r3 r1	4 OR r3 r1	4 XOR r1 r0
5 OR r3 r4	5 OR r3 r0	5 OR r2 r0	5 XOR r1 r0	5 XOR r3 r0	5 XOR r2 r3
6 XOR r4 r1	6 XOR r1 r3	6 XOR r2 r4	6 XOR r3 r4	6 XOR r4 r3	6 OR r3 r0
7 AND r1 r3	7 AND r3 r1	7 AND r4 r2	7 AND r4 r2	7 AND r3 r1	7 XOR r3 r4
8 OR r1 r0	8 XOR r3 r4	8 XOR r4 r3	8 OR r4 r0	8 AND r3 r4	8 AND r4 r2
9 XOR r3 r2	9 AND r4 r1	9 AND r3 r2	9 XOR r4 r1	9 XOR r3 r2	9 XOR r4 r1
10 OR r2 r1	10 XOR r2 r4	10 XOR r1 r3	10 AND r1 r3	10 OR r2 r4	10 OR r1 r3
11 XOR r0 r2	r0 r1 r2 r3	r0 r1 r2 r4	11 XOR r1 r2	11 XOR r1 r2	11 XOR r0 r1
r0 r1 r3 r4			r0 r1 r3 r4	r0 r1 r3 r4	r0 r2 r3 r4
S-box 139	**S-box 140**	**S-box 141**	**S-box 142**	**S-box 143**	**S-box 144**
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 AND r0 r2
3 AND r2 r3	3 OR r2 r1	3 OR r2 r1	3 AND r2 r0	3 OR r2 r1	3 XOR r0 r3
4 OR r2 r1	4 AND r2 r3	4 XOR r1 r3	4 XOR r2 r3	4 XOR r2 r3	4 OR r3 r1
5 XOR r1 r0	5 OR r3 r0	5 AND r1 r0	5 OR r3 r0	5 AND r3 r0	5 XOR r3 r2
6 XOR r3 r2	6 XOR r1 r3	6 XOR r1 r4	6 XOR r3 r1	6 XOR r3 r4	6 AND r2 r0
7 OR r0 r3	7 XOR r3 r4	7 XOR r2 r4	7 AND r1 r2	7 AND r4 r3	7 XOR r2 r4
8 AND r2 r0	8 AND r3 r1	8 AND r4 r1	8 XOR r1 r4	8 XOR r1 r4	8 AND r4 r3
9 XOR r3 r4	9 XOR r2 r3	9 XOR r4 r3	r0 r1 r2 r3	r0 r1 r2 r3	9 XOR r1 r4
10 AND r4 r2	10 OR r3 r0	10 OR r3 r1			r0 r1 r2 r3
11 XOR r0 r4	11 XOR r3 r4	11 XOR r0 r3			
r0 r1 r2 r3	r0 r1 r2 r3	r0 r1 r2 r4			
S-box 145	**S-box 146**	**S-box 147**	**S-box 148**	**S-box 149**	**S-box 150**
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2
3 AND r2 r0	3 AND r2 r1	3 OR r2 r1	3 OR r2 r1	3 OR r2 r1	3 OR r2 r1
4 XOR r1 r0	4 XOR r2 r3	4 AND r2 r3	4 XOR r1 r3	4 XOR r2 r3	4 XOR r2 r3
5 XOR r2 r3	5 AND r3 r2	5 OR r3 r0	5 AND r1 r0	5 XOR r2 r4	5 XOR r2 r4
6 OR r3 r0	6 XOR r3 r0	6 XOR r1 r3	6 XOR r1 r2	6 AND r3 r2	6 AND r4 r2
7 XOR r3 r4	7 XOR r4 r0	7 XOR r2 r4	7 XOR r2 r4	7 XOR r1 r3	7 XOR r1 r4
8 AND r4 r2	8 OR r0 r2	8 AND r4 r0	8 AND r4 r1	8 OR r3 r0	8 OR r4 r0
9 XOR r0 r4	9 XOR r0 r1	9 XOR r3 r4	9 XOR r3 r4	9 XOR r3 r4	9 XOR r3 r4
r0 r1 r2 r3	r0 r2 r3 r4	r0 r1 r2 r3	r0 r1 r2 r3	r0 r1 r2 r3	r0 r1 r2 r3
S-box 151	**S-box 152**	**S-box 153**	**S-box 154**	**S-box 155**	**S-box 156**
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 AND r0 r2	2 OR r0 r2	2 OR r1 r4	2 XOR r0 r2	2 XOR r0 r2	2 XOR r2 r0
3 XOR r0 r3	3 XOR r0 r3	3 XOR r0 r2	3 AND r2 r1	3 AND r2 r0	3 AND r0 r2
4 OR r3 r1	4 OR r3 r1	4 AND r2 r4	4 XOR r2 r0	4 XOR r2 r3	4 XOR r0 r3
5 XOR r1 r0	5 AND r3 r0	5 XOR r0 r3	5 XOR r2 r3	5 OR r3 r1	5 OR r3 r2
6 XOR r3 r2	6 XOR r3 r4	6 XOR r1 r2	6 XOR r4 r0	6 XOR r3 r4	6 XOR r1 r3
7 AND r2 r0	7 OR r4 r0	7 AND r2 r0	7 AND r0 r2	7 MOV r4 r2	7 AND r3 r1
8 XOR r2 r4	8 XOR r4 r1	8 XOR r2 r3	8 XOR r1 r0	8 AND r2 r3	8 XOR r3 r4
9 AND r4 r3	9 AND r1 r3	9 OR r3 r0	9 AND r0 r4	9 XOR r1 r2	9 OR r4 r0
10 XOR r0 r4	10 XOR r1 r2	10 XOR r3 r4	10 XOR r0 r3	r0 r1 r3 r4	10 OR r4 r1
r0 r1 r2 r3	r0 r1 r3 r4	r0 r1 r2 r3	r0 r1 r2 r4		11 XOR r2 r4
					r0 r1 r2 r3
S-box 157	**S-box 158**	**S-box 159**	**S-box 160**	**S-box 161**	**S-box 162**
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 XOR r0 r2	2 AND r0 r2	2 AND r0 r2	2 OR r0 r2	2 XOR r0 r2	2 XOR r0 r2
3 AND r2 r0	3 XOR r0 r3	3 XOR r0 r3	3 XOR r0 r3	3 AND r2 r0	3 AND r2 r0
4 XOR r1 r0	4 OR r3 r1	4 OR r3 r1	4 AND r3 r2	4 XOR r1 r0	4 XOR r1 r0
5 XOR r2 r3	5 XOR r2 r0	5 XOR r2 r0	5 XOR r1 r0	5 XOR r2 r3	5 XOR r2 r3
6 AND r3 r1	6 XOR r2 r3	6 XOR r2 r3	6 XOR r1 r3	6 OR r3 r0	6 OR r3 r0
7 XOR r3 r4	7 AND r3 r2	7 OR r3 r2	7 OR r3 r1	7 OR r3 r4	7 XOR r1 r3
8 OR r4 r1	8 XOR r3 r4	8 XOR r3 r4	8 XOR r3 r4	8 XOR r3 r1	8 AND r3 r1
9 AND r4 r2	9 OR r4 r2	9 AND r4 r2	9 AND r4 r1	9 AND r1 r2	9 XOR r3 r4
10 XOR r0 r4	10 XOR r0 r4	10 XOR r0 r4	10 XOR r0 r4	10 XOR r0 r1	10 AND r4 r2
11 AND r4 r3	11 AND r4 r0	11 OR r4 r0	11 AND r4 r0	11 AND r1 r0	11 AND r4 r3
12 XOR r1 r4	12 XOR r1 r4	12 XOR r1 r4	12 XOR r2 r4	12 XOR r1 r4	12 XOR r0 r4
r0 r1 r2 r3	r0 r1 r2 r3	r0 r1 r2 r3	r0 r1 r2 r3	r0 r1 r2 r3	r0 r1 r2 r3

S-box 163	S-box 164	S-box 165	S-box 166	S-box 167	S-box 168
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 AND r0 r2	2 AND r0 r2	2 OR r0 r2	2 OR r0 r2	2 XOR r0 r2	2 AND r0 r2
3 XOR r0 r3	3 XOR r0 r3	3 XOR r0 r3	3 XOR r0 r3	3 XOR r0 r3	3 XOR r0 r3
4 OR r3 r1	4 OR r3 r1	4 OR r3 r1	4 OR r3 r1	4 AND r2 r0	4 AND r3 r1
5 XOR r2 r0	5 XOR r2 r0	5 XOR r3 r4	5 XOR r3 r4	5 XOR r1 r2	5 OR r3 r2
6 XOR r2 r3	6 XOR r2 r3	6 OR r4 r2	6 MOV r4 r0	6 AND r2 r4	6 XOR r3 r4
7 AND r3 r2	7 OR r3 r2	7 AND r4 r0	7 AND r0 r3	7 XOR r2 r3	7 OR r4 r0
8 XOR r3 r4	8 XOR r3 r4	8 XOR r1 r4	8 XOR r0 r2	8 AND r3 r1	8 AND r2 r4
9 OR r4 r2	9 AND r4 r2	9 OR r4 r3	9 OR r2 r0	9 OR r3 r0	9 XOR r2 r1
10 XOR r1 r4	10 XOR r1 r4	10 XOR r2 r4	10 XOR r1 r2	10 XOR r3 r4	10 AND r1 r0
r0 r1 r2 r3	r0 r1 r2 r3	r0 r1 r2 r3	r0 r1 r3 r4	r0 r1 r2 r3	11 XOR r1 r4
					r0 r1 r2 r3
S-box 169	**S-box 170**	**S-box 171**	**S-box 172**	**S-box 173**	**S-box 174**
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 OR r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 AND r0 r2	2 AND r0 r2	2 XOR r0 r2	2 AND r0 r2	2 XOR r0 r2	2 XOR r0 r2
3 XOR r0 r3	3 XOR r0 r3	3 AND r2 r0	3 XOR r0 r3	3 AND r2 r0	3 AND r2 r0
4 OR r3 r1	4 OR r3 r1	4 XOR r2 r3	4 OR r3 r1	4 XOR r1 r0	4 XOR r2 r3
5 XOR r1 r2	5 XOR r3 r2	5 OR r3 r0	5 XOR r3 r2	5 XOR r2 r3	5 OR r3 r0
6 AND r1 r3	6 AND r2 r0	6 XOR r1 r3	6 AND r2 r3	6 OR r3 r0	6 AND r0 r2
7 XOR r1 r4	7 XOR r1 r2	7 OR r3 r1	7 XOR r2 r4	7 XOR r1 r3	7 XOR r0 r1
8 AND r4 r0	8 AND r2 r1	8 XOR r3 r4	8 AND r4 r0	8 AND r3 r1	8 AND r1 r3
9 XOR r3 r4	9 XOR r2 r4	9 OR r4 r2	9 XOR r4 r3	9 XOR r3 r4	9 XOR r1 r4
10 OR r4 r1	10 OR r4 r1	10 AND r4 r1	10 OR r3 r2	10 AND r4 r2	10 AND r4 r2
11 XOR r2 r4	11 XOR r0 r4	11 XOR r0 r4	11 XOR r1 r3	11 XOR r0 r4	11 XOR r3 r4
r0 r1 r2 r3	r0 r1 r2 r3	r0 r1 r2 r3	r0 r1 r2 r4	r0 r1 r2 r3	r0 r1 r2 r3
S-box 175	**S-box 176**	**S-box 177**	**S-box 178**	**S-box 179**	**S-box 180**
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 XOR r2 r0	2 OR r0 r2
3 AND r2 r0	3 AND r2 r0	3 AND r2 r1	3 AND r2 r0	3 AND r0 r2	3 AND r2 r1
4 XOR r2 r3	4 XOR r2 r3	4 XOR r3 r0	4 XOR r2 r3	4 XOR r0 r3	4 XOR r0 r4
5 OR r3 r0	5 OR r3 r0	5 XOR r2 r3	5 OR r3 r1	5 OR r3 r2	5 OR r1 r0
6 AND r0 r2	6 XOR r0 r2	6 AND r3 r2	6 XOR r3 r4	6 XOR r1 r2	6 XOR r0 r3
7 XOR r3 r1	7 XOR r3 r1	7 XOR r3 r0	7 OR r4 r0	7 XOR r3 r4	7 OR r3 r2
8 OR r1 r0	8 AND r1 r2	8 AND r0 r2	8 OR r4 r2	8 AND r4 r3	8 XOR r2 r3
9 XOR r1 r4	9 XOR r1 r4	9 XOR r0 r4	9 XOR r4 r3	9 XOR r4 r1	9 AND r3 r1
10 OR r4 r2	10 AND r4 r3	10 OR r4 r0	10 OR r3 r0	10 OR r1 r3	10 XOR r2 r4
11 XOR r0 r4	11 XOR r2 r4	11 XOR r1 r4	11 XOR r1 r3	11 XOR r1 r2	11 AND r4 r3
r0 r1 r2 r3	r0 r1 r2 r3	r0 r1 r2 r3	r0 r1 r2 r4	r0 r1 r3 r4	12 XOR r1 r4
					r0 r1 r2 r3
S-box 181	**S-box 182**	**S-box 183**	**S-box 184**	**S-box 185**	**S-box 186**
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 OR r0 r2	2 XOR r0 r2	2 AND r0 r2	2 OR r0 r2	2 XOR r0 r2	2 XOR r0 r2
3 AND r2 r1	3 AND r2 r0	3 XOR r0 r3	3 XOR r0 r3	3 AND r2 r0	3 AND r2 r0
4 XOR r0 r4	4 XOR r1 r0	4 OR r3 r1	4 OR r3 r1	4 XOR r1 r0	4 XOR r1 r0
5 XOR r3 r0	5 XOR r2 r3	5 XOR r2 r0	5 XOR r3 r4	5 XOR r2 r3	5 XOR r2 r3
6 XOR r1 r3	6 OR r3 r0	6 XOR r3 r4	6 MOV r4 r0	6 AND r3 r1	6 OR r3 r0
7 OR r1 r2	7 XOR r1 r3	7 AND r4 r3	7 AND r0 r3	7 XOR r3 r4	7 XOR r3 r4
8 XOR r4 r1	8 AND r3 r1	8 XOR r4 r2	8 XOR r0 r1	8 MOV r4 r0	8 MOV r4 r1
9 OR r1 r0	9 XOR r3 r4	9 OR r2 r3	9 AND r1 r0	9 AND r0 r3	9 AND r1 r3
10 XOR r2 r1	10 XOR r4 r2	10 XOR r1 r2	10 XOR r1 r2	10 XOR r0 r1	10 XOR r0 r1
11 OR r1 r4	11 OR r4 r1	r0 r1 r3 r4	r0 r1 r3 r4	r0 r2 r3 r4	r0 r2 r3 r4
12 XOR r0 r1	12 XOR r0 r4				
r0 r2 r3 r4	r0 r1 r2 r3				
S-box 187	**S-box 188**	**S-box 189**	**S-box 190**	**S-box 191**	**S-box 192**
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 OR r0 r2	2 XOR r0 r2	2 XOR r2 r0	2 XOR r2 r0	2 XOR r2 r0	2 AND r0 r2
3 XOR r0 r3	3 AND r2 r1	3 AND r0 r2	3 AND r0 r2	3 AND r0 r2	3 XOR r0 r3
4 OR r3 r1	4 XOR r2 r3	4 XOR r0 r3	4 XOR r0 r3	4 XOR r0 r3	4 OR r3 r1
5 XOR r3 r4	5 XOR r4 r0	5 OR r3 r2	5 OR r3 r2	5 OR r3 r2	5 XOR r2 r3
6 AND r4 r1	6 OR r3 r4	6 XOR r1 r0	6 XOR r3 r1	6 XOR r3 r1	6 AND r3 r2
7 XOR r4 r2	7 XOR r1 r3	7 XOR r3 r4	7 AND r1 r0	7 AND r1 r0	7 XOR r3 r4
8 OR r4 r0	8 AND r3 r1	8 AND r4 r3	8 XOR r1 r4	8 XOR r1 r4	8 AND r4 r0
9 XOR r1 r4	9 XOR r3 r0	9 XOR r4 r1	9 AND r4 r3	9 OR r4 r3	9 XOR r1 r4
10 OR r4 r3	10 OR r0 r1	10 OR r1 r3	10 XOR r2 r4	10 XOR r2 r4	r0 r1 r2 r3
11 XOR r2 r4	11 XOR r0 r4	11 XOR r1 r2	r0 r1 r2 r3	r0 r1 r2 r3	
r0 r1 r2 r3	r0 r1 r2 r3	r0 r1 r3 r4			

S-box 193	S-box 194	S-box 195	S-box 196	S-box 197	S-box 198
0 MOV r4 r0 1 AND r0 r1 2 AND r0 r2 3 XOR r0 r3 4 OR r3 r1 5 XOR r2 r3 6 OR r3 r2 7 XOR r3 r4 8 AND r4 r0 9 XOR r1 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 XOR r2 r0 3 AND r0 r2 4 XOR r0 r3 5 OR r3 r2 6 XOR r1 r3 7 OR r3 r1 8 XOR r3 r4 9 AND r4 r1 10 OR r4 r0 11 XOR r2 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 AND r0 r2 3 XOR r0 r3 4 AND r3 r1 5 OR r3 r2 6 XOR r3 r4 7 AND r4 r3 8 OR r4 r0 9 XOR r1 r4 10 OR r4 r1 11 OR r4 r3 12 XOR r2 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 OR r0 r2 3 XOR r0 r3 4 AND r3 r2 5 XOR r1 r0 6 XOR r1 r3 7 AND r3 r1 8 XOR r3 r4 9 OR r4 r1 10 XOR r0 r4 11 AND r4 r0 12 XOR r2 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 AND r2 r0 4 XOR r2 r3 5 OR r3 r0 6 XOR r1 r3 7 AND r3 r1 8 XOR r3 r4 9 OR r4 r1 10 OR r4 r2 11 XOR r0 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 AND r2 r0 4 XOR r2 r3 5 OR r3 r0 6 XOR r1 r3 7 OR r3 r1 8 XOR r3 r4 9 AND r4 r1 10 OR r4 r2 11 XOR r0 r4 r0 r1 r2 r3
S-box 199	S-box 200	S-box 201	S-box 202	S-box 203	S-box 204
0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 AND r2 r0 4 XOR r2 r3 5 OR r3 r0 6 XOR r1 r3 7 OR r3 r1 8 XOR r3 r4 9 OR r4 r2 10 AND r4 r1 11 XOR r0 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 XOR r2 r0 3 AND r0 r2 4 XOR r0 r3 5 OR r3 r2 6 XOR r1 r2 7 XOR r1 r3 8 AND r3 r1 9 XOR r3 r4 10 OR r4 r1 11 XOR r2 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 AND r0 r2 3 XOR r0 r3 4 OR r3 r1 5 XOR r2 r3 6 AND r3 r2 7 XOR r3 r4 8 AND r4 r0 9 AND r4 r2 10 XOR r1 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 AND r0 r2 3 XOR r0 r3 4 OR r3 r1 5 XOR r2 r3 6 XOR r3 r4 7 AND r3 r2 8 XOR r1 r3 9 OR r3 r0 10 XOR r3 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 AND r0 r2 3 XOR r0 r3 4 OR r3 r1 5 XOR r2 r3 6 XOR r3 r4 7 OR r4 r2 8 XOR r1 r4 9 OR r4 r0 10 XOR r3 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 AND r0 r2 3 XOR r0 r3 4 OR r3 r1 5 XOR r3 r2 6 XOR r2 r4 7 AND r2 r3 8 XOR r1 r2 9 OR r2 r0 10 XOR r2 r4 r0 r1 r2 r3
S-box 205	S-box 206	S-box 207	S-box 208	S-box 209	S-box 210
0 MOV r4 r0 1 AND r0 r1 2 AND r0 r2 3 XOR r0 r3 4 OR r3 r1 5 XOR r3 r0 6 XOR r2 r3 7 AND r3 r2 8 XOR r3 r4 9 OR r4 r2 10 XOR r1 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 OR r0 r2 3 XOR r0 r3 4 OR r3 r1 5 XOR r3 r4 6 AND r4 r2 7 OR r4 r0 8 XOR r1 r4 9 AND r4 r3 10 XOR r2 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 OR r0 r2 3 XOR r0 r3 4 OR r3 r1 5 XOR r3 r4 6 MOV r4 r0 7 OR r0 r3 8 XOR r0 r2 9 OR r2 r0 10 XOR r1 r2 r0 r1 r3 r4	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 AND r2 r0 4 XOR r1 r0 5 XOR r2 r3 6 AND r3 r1 7 XOR r3 r4 8 AND r4 r0 9 AND r4 r2 10 XOR r1 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 AND r0 r2 3 XOR r0 r3 4 OR r3 r1 5 XOR r1 r2 6 AND r1 r3 7 XOR r1 r4 8 AND r4 r0 9 XOR r3 r4 10 AND r4 r1 11 XOR r2 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 AND r0 r2 3 XOR r0 r3 4 OR r3 r1 5 XOR r1 r2 6 AND r2 r3 7 XOR r2 r4 8 AND r4 r0 9 XOR r3 r4 10 OR r4 r2 11 XOR r1 r4 r0 r1 r2 r3
S-box 211	S-box 212	S-box 213	S-box 214	S-box 215	S-box 216
0 MOV r4 r0 1 AND r0 r1 2 AND r0 r2 3 XOR r0 r3 4 OR r3 r1 5 XOR r2 r0 6 XOR r3 r4 7 XOR r4 r2 8 OR r2 r3 9 XOR r1 r2 10 OR r2 r0 11 XOR r2 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 AND r2 r0 4 XOR r2 r3 5 OR r3 r0 6 XOR r1 r2 7 XOR r3 r4 8 AND r4 r3 9 XOR r4 r1 10 OR r1 r3 11 XOR r0 r1 r0 r2 r3 r4	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 AND r2 r0 4 XOR r2 r3 5 OR r3 r0 6 XOR r3 r1 7 AND r1 r2 8 XOR r1 r4 9 AND r4 r3 10 XOR r0 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 AND r2 r0 4 XOR r2 r3 5 OR r3 r0 6 XOR r3 r1 7 AND r1 r2 8 XOR r1 r4 9 OR r4 r3 10 XOR r0 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 AND r2 r0 4 XOR r2 r3 5 OR r3 r0 6 XOR r3 r1 7 AND r1 r0 8 XOR r1 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 OR r2 r1 4 XOR r2 r3 5 AND r3 r0 6 XOR r1 r3 7 AND r3 r1 8 XOR r3 r4 r0 r1 r2 r3
S-box 217	S-box 218	S-box 219	S-box 220	S-box 221	S-box 222
0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 AND r2 r0 4 XOR r2 r3 5 OR r3 r0 6 XOR r3 r1 7 AND r1 r0 8 AND r1 r2 9 XOR r1 r4 10 AND r4 r3 11 XOR r0 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 AND r2 r0 4 XOR r2 r3 5 OR r3 r0 6 XOR r1 r3 7 AND r3 r1 8 XOR r3 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 XOR r0 r3 4 AND r2 r0 5 XOR r1 r2 6 AND r2 r4 7 XOR r2 r3 8 OR r3 r0 9 XOR r3 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 AND r2 r0 4 XOR r1 r0 5 XOR r2 r3 6 OR r3 r0 7 XOR r1 r3 8 OR r3 r1 9 XOR r3 r4 10 AND r4 r1 11 XOR r0 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 AND r0 r2 3 XOR r0 r3 4 OR r3 r1 5 OR r3 r2 6 XOR r3 r4 7 AND r4 r0 8 XOR r1 r4 9 AND r4 r1 10 XOR r2 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 AND r0 r2 3 XOR r0 r3 4 OR r3 r1 5 OR r3 r2 6 XOR r3 r4 7 AND r4 r0 8 XOR r1 r4 9 OR r4 r1 10 XOR r2 r4 r0 r1 r2 r3

S-box 223	S-box 224	S-box 225	S-box 226	S-box 227	S-box 228
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 XOR r0 r2	2 AND r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 AND r0 r2
3 OR r2 r0	3 XOR r0 r3	3 AND r2 r0	3 AND r2 r0	3 MOV r2 r0	3 XOR r0 r3
4 XOR r2 r3	4 OR r3 r1	4 XOR r1 r3	4 XOR r2 r3	4 AND r0 r3	4 AND r3 r1
5 AND r3 r0	5 OR r3 r2	5 XOR r2 r1	5 OR r3 r0	5 XOR r0 r1	5 OR r3 r2
6 XOR r1 r3	6 XOR r3 r4	6 AND r1 r0	6 XOR r1 r2	6 AND r1 r0	6 XOR r3 r4
7 OR r3 r1	7 AND r4 r0	7 XOR r1 r3	7 XOR r1 r3	7 XOR r1 r4	7 AND r4 r0
8 XOR r3 r4	8 AND r4 r1	8 OR r3 r1	8 OR r3 r1	8 OR r4 r0	8 AND r4 r2
9 AND r4 r1	9 XOR r4 r2	9 XOR r3 r4	9 XOR r3 r4	9 XOR r3 r4	9 XOR r4 r1
10 OR r4 r2	10 OR r2 r0	10 AND r4 r1	10 AND r4 r1	r0 r1 r2 r3	10 AND r1 r0
11 XOR r0 r4	11 AND r2 r3	11 XOR r0 r4	11 XOR r0 r4		11 AND r1 r3
r0 r1 r2 r3	12 XOR r1 r2	r0 r1 r2 r3	r0 r1 r2 r3		12 XOR r1 r2
	r0 r1 r3 r4				r0 r1 r3 r4
S-box 229	**S-box 230**	**S-box 231**	**S-box 232**	**S-box 233**	**S-box 234**
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 XOR r0 r2	2 XOR r2 r0	2 XOR r0 r2	2 OR r0 r2	2 XOR r0 r2	2 XOR r0 r2
3 AND r2 r0	3 AND r0 r2	3 AND r2 r0	3 XOR r3 r0	3 OR r2 r1	3 OR r2 r1
4 XOR r2 r3	4 XOR r0 r3	4 XOR r2 r3	4 OR r0 r1	4 XOR r2 r3	4 XOR r2 r3
5 AND r3 r1	5 OR r3 r2	5 OR r3 r0	5 AND r0 r3	5 AND r3 r0	5 AND r3 r0
6 OR r3 r0	6 XOR r1 r3	6 XOR r1 r3	6 XOR r0 r4	6 XOR r3 r1	6 XOR r1 r3
7 XOR r3 r4	7 AND r3 r1	7 AND r3 r1	7 OR r4 r3	7 AND r1 r2	7 AND r3 r1
8 AND r4 r0	8 XOR r3 r4	8 XOR r3 r4	8 XOR r4 r2	8 XOR r1 r4	8 XOR r3 r4
9 AND r4 r2	9 OR r4 r1	9 OR r4 r1	9 AND r2 r0	9 OR r4 r3	9 XOR r4 r2
10 XOR r1 r4	10 XOR r2 r4	10 XOR r0 r4	10 XOR r1 r2	10 XOR r0 r4	10 OR r4 r1
r0 r1 r2 r3	r0 r1 r2 r3	r0 r1 r2 r3	r0 r1 r3 r4	r0 r1 r2 r3	11 XOR r0 r4
					r0 r1 r2 r3
S-box 235	**S-box 236**	**S-box 237**	**S-box 238**	**S-box 239**	**S-box 240**
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 AND r0 r2	2 AND r0 r2	2 AND r0 r2	2 OR r0 r2	2 AND r0 r2	2 AND r0 r2
3 XOR r0 r3	3 XOR r0 r3	3 XOR r0 r3	3 XOR r2 r3	3 XOR r0 r3	3 XOR r0 r3
4 OR r3 r1	4 OR r3 r1	4 OR r3 r1	4 AND r2 r1	4 OR r3 r1	4 OR r3 r1
5 XOR r3 r2	5 XOR r3 r2	5 XOR r2 r3	5 XOR r2 r3	5 XOR r2 r3	5 XOR r2 r3
6 AND r2 r0	6 AND r2 r0	6 AND r3 r2	6 XOR r3 r0	6 AND r3 r2	6 AND r3 r2
7 XOR r2 r4	7 XOR r2 r4	7 XOR r3 r4	7 OR r0 r2	7 XOR r3 r4	7 XOR r3 r4
8 AND r4 r2	8 OR r4 r2	8 OR r4 r2	8 XOR r1 r0	8 AND r4 r0	8 OR r4 r0
9 XOR r1 r4	9 XOR r1 r4	9 AND r4 r0	9 XOR r3 r4	9 AND r4 r3	9 OR r4 r2
r0 r1 r2 r3	r0 r1 r2 r3	10 XOR r1 r4	10 AND r0 r3	10 XOR r1 r4	10 XOR r1 r4
		r0 r1 r2 r3	11 XOR r0 r4	r0 r1 r2 r3	r0 r1 r2 r3
			r0 r1 r2 r3		
S-box 241	**S-box 242**	**S-box 243**	**S-box 244**	**S-box 245**	**S-box 246**
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 OR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 AND r0 r2	2 XOR r0 r2	2 XOR r0 r2
3 XOR r0 r3	3 AND r2 r0	3 OR r2 r1	3 XOR r0 r3	3 AND r2 r0	3 OR r2 r1
4 AND r3 r2	4 XOR r2 r3	4 XOR r2 r3	4 OR r3 r1	4 XOR r2 r3	4 XOR r2 r4
5 XOR r1 r3	5 AND r3 r1	5 AND r3 r2	5 XOR r2 r3	5 OR r3 r0	5 AND r4 r3
6 OR r3 r1	6 OR r3 r0	6 XOR r3 r4	6 AND r3 r2	6 XOR r3 r1	6 OR r4 r0
7 XOR r3 r4	7 XOR r3 r4	7 AND r4 r0	7 XOR r4 r3	7 AND r1 r2	7 XOR r4 r2
8 AND r4 r1	8 AND r4 r3	8 XOR r1 r4	8 AND r3 r0	8 XOR r1 r0	8 AND r2 r3
9 OR r4 r0	9 OR r4 r2	r0 r1 r2 r3	9 XOR r1 r3	9 OR r0 r4	9 XOR r1 r2
10 XOR r2 r4	10 XOR r1 r4		r0 r1 r2 r4	10 AND r0 r3	r0 r1 r3 r4
r0 r1 r2 r3	r0 r1 r2 r3			11 XOR r0 r2	
				r0 r1 r3 r4	
S-box 247	**S-box 248**	**S-box 249**	**S-box 250**	**S-box 251**	**S-box 252**
0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0	0 MOV r4 r0
1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1	1 AND r0 r1
2 XOR r0 r2	2 XOR r0 r2	2 XOR r0 r2	2 OR r0 r2	2 XOR r0 r2	2 XOR r0 r2
3 XOR r1 r0	3 AND r2 r1	3 AND r2 r1	3 AND r2 r1	3 AND r2 r3	3 OR r2 r1
4 MOV r2 r0	4 XOR r2 r3	4 XOR r2 r3	4 XOR r2 r3	4 OR r2 r1	4 XOR r1 r4
5 AND r0 r3	5 AND r3 r0	5 AND r3 r2	5 AND r3 r4	5 XOR r1 r3	5 XOR r2 r3
6 XOR r0 r4	6 XOR r1 r3	6 XOR r3 r0	6 XOR r0 r3	6 AND r1 r0	6 AND r3 r0
7 AND r4 r0	7 OR r3 r4	7 AND r0 r2	7 OR r3 r2	7 XOR r1 r4	7 XOR r3 r1
8 XOR r4 r1	8 XOR r2 r3	8 XOR r0 r4	8 AND r3 r0	8 AND r4 r3	8 AND r1 r2
9 OR r1 r0	r0 r1 r2 r4	r0 r1 r2 r3	9 XOR r1 r3	9 XOR r2 r4	9 XOR r0 r1
10 XOR r1 r2			r0 r1 r2 r4	r0 r1 r2 r3	r0 r2 r3 r4
r0 r1 r3 r4					

S-box 253	S-box 254	S-box 255	S-box 256	S-box 257	S-box 258
0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 OR r2 r1 4 XOR r2 r3 5 AND r3 r0 6 XOR r1 r3 r0 r1 r2 r4	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 OR r2 r1 4 XOR r2 r3 5 AND r3 r0 6 XOR r3 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 AND r2 r1 4 XOR r2 r3 5 MOV r3 r2 6 AND r2 r4 7 XOR r1 r2 r0 r1 r3 r4	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 OR r2 r1 4 XOR r1 r3 5 AND r1 r0 6 XOR r1 r2 7 XOR r2 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 OR r2 r1 4 XOR r2 r3 5 XOR r4 r3 6 AND r3 r0 7 XOR r1 r3 r0 r1 r2 r4	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 AND r2 r1 4 XOR r2 r3 5 OR r3 r1 6 XOR r3 r4 r0 r1 r2 r3
S-box 259	S-box 260	S-box 261	S-box 262	S-box 263	S-box 264
0 MOV r4 r0 1 AND r0 r1 2 AND r0 r2 3 XOR r0 r3 4 OR r3 r1 5 XOR r2 r0 6 XOR r2 r3 7 AND r3 r2 8 XOR r3 r4 9 OR r4 r2 10 XOR r0 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 OR r0 r2 3 XOR r0 r3 4 OR r3 r1 5 XOR r3 r4 6 AND r4 r0 7 AND r4 r2 8 XOR r4 r3 9 AND r3 r0 10 XOR r1 r3 r0 r1 r2 r4	0 MOV r4 r0 1 AND r0 r1 2 OR r0 r2 3 XOR r0 r3 4 OR r3 r1 5 AND r2 r3 6 XOR r3 r4 7 AND r4 r0 8 XOR r2 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 OR r0 r2 3 XOR r0 r3 4 OR r3 r1 5 XOR r3 r4 6 MOV r4 r0 7 AND r0 r3 8 XOR r0 r1 r0 r2 r3 r4	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 AND r2 r1 4 AND r2 r3 5 XOR r2 r0 6 XOR r4 r0 7 OR r0 r3 8 XOR r0 r1 r0 r2 r3 r4	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 AND r2 r0 4 XOR r0 r1 5 XOR r2 r3 6 AND r3 r0 7 XOR r3 r4 8 MOV r4 r2 9 AND r2 r3 10 XOR r0 r2 r0 r1 r3 r4
S-box 265	S-box 266	S-box 267	S-box 268	S-box 269	S-box 270
0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 AND r2 r1 4 AND r2 r3 5 XOR r2 r0 6 AND r0 r3 7 XOR r0 r4 8 OR r4 r0 9 XOR r1 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 AND r2 r0 4 XOR r2 r3 5 OR r3 r0 6 XOR r1 r3 r0 r1 r2 r4	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 OR r2 r1 4 AND r2 r3 5 XOR r4 r2 6 AND r2 r0 7 XOR r1 r2 r0 r1 r3 r4	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 XOR r0 r3 4 AND r2 r0 5 XOR r1 r2 6 AND r2 r4 7 XOR r2 r3 r0 r1 r2 r4	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 AND r2 r0 4 XOR r3 r2 5 OR r2 r1 6 XOR r2 r4 7 AND r4 r0 8 XOR r1 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 OR r0 r2 3 XOR r3 r0 4 OR r0 r1 5 AND r0 r3 6 XOR r0 r4 7 OR r4 r3 8 XOR r4 r2 9 AND r2 r0 10 XOR r2 r3 r0 r1 r2 r4
S-box 271	S-box 272	S-box 273	S-box 274	S-box 275	S-box 276
0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 AND r2 r1 4 XOR r2 r3 5 OR r3 r1 6 XOR r3 r0 7 AND r3 r4 8 XOR r1 r3 9 OR r3 r2 10 XOR r0 r3 r0 r1 r2 r4	0 MOV r4 r0 1 AND r0 r1 2 OR r0 r2 3 XOR r0 r3 4 OR r3 r1 5 AND r2 r3 6 XOR r3 r4 7 OR r4 r1 8 XOR r2 r4 9 OR r4 r0 10 XOR r3 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 AND r0 r2 3 XOR r0 r3 4 AND r3 r1 5 OR r3 r2 6 XOR r3 r4 7 OR r4 r0 8 XOR r1 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 AND r0 r2 3 XOR r0 r3 4 OR r3 r1 5 AND r3 r2 6 XOR r3 r4 7 AND r4 r0 8 XOR r1 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 AND r0 r2 3 XOR r0 r3 4 AND r3 r1 5 OR r3 r2 6 XOR r3 r4 7 AND r4 r3 8 OR r4 r0 9 XOR r1 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 AND r0 r2 3 XOR r0 r3 4 AND r3 r1 5 OR r3 r2 6 XOR r3 r4 7 AND r4 r0 8 AND r4 r2 9 XOR r1 r4 r0 r1 r2 r3
S-box 277	S-box 278	S-box 279	S-box 280	S-box 281	S-box 282
0 MOV r4 r0 1 AND r0 r1 2 OR r0 r2 3 XOR r0 r3 4 OR r3 r1 5 AND r3 r4 6 XOR r3 r0 7 AND r0 r4 8 XOR r0 r2 9 AND r2 r0 10 XOR r1 r2 r0 r1 r3 r4	0 MOV r4 r0 1 AND r0 r1 2 OR r0 r2 3 XOR r0 r3 4 OR r3 r1 5 XOR r3 r4 6 OR r4 r0 7 AND r4 r1 8 XOR r2 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 AND r0 r2 3 XOR r0 r3 4 OR r3 r1 5 XOR r3 r2 6 AND r2 r0 7 AND r2 r4 8 XOR r1 r2 r0 r1 r3 r4	0 MOV r4 r0 1 AND r0 r1 2 AND r0 r2 3 XOR r0 r3 4 OR r3 r1 5 AND r3 r2 6 XOR r4 r3 7 AND r3 r0 8 AND r3 r4 9 XOR r1 r3 r0 r1 r2 r4	0 MOV r4 r0 1 AND r0 r1 2 AND r0 r2 3 XOR r0 r3 4 OR r3 r1 5 AND r3 r2 6 XOR r3 r4 7 AND r4 r0 8 AND r4 r2 9 XOR r1 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 AND r0 r2 3 XOR r0 r3 4 AND r3 r1 5 OR r3 r2 6 XOR r3 r4 7 AND r4 r0 8 AND r4 r1 9 XOR r2 r4 r0 r1 r2 r3
S-box 283	S-box 284	S-box 285	S-box 286	S-box 287	S-box 288
0 MOV r4 r0 1 AND r0 r1 2 AND r0 r2 3 XOR r0 r3 4 OR r3 r1 5 XOR r3 r2 6 AND r2 r0 7 XOR r1 r2 r0 r1 r3 r4	0 MOV r4 r0 1 AND r0 r1 2 AND r0 r2 3 XOR r0 r3 4 OR r3 r1 5 XOR r3 r2 6 AND r2 r0 7 XOR r2 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 AND r2 r0 4 XOR r2 r3 5 OR r3 r1 6 XOR r3 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 AND r0 r2 3 XOR r0 r3 4 OR r3 r1 5 XOR r2 r3 6 AND r3 r2 7 XOR r3 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 AND r2 r0 4 XOR r2 r3 5 AND r3 r1 6 OR r3 r0 7 XOR r3 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 OR r2 r1 4 XOR r2 r4 5 AND r4 r0 6 XOR r1 r4 r0 r1 r2 r3

S-box 289	**S-box 290**	**S-box 291**	**S-box 292**	**S-box 293**	**S-box 294**
0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 OR r2 r1 4 XOR r2 r4 5 AND r4 r0 6 XOR r3 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 AND r2 r0 4 XOR r3 r2 5 OR r2 r1 6 XOR r2 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 AND r2 r1 4 AND r2 r3 5 XOR r2 r0 6 AND r0 r3 7 XOR r0 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 MOV r2 r0 4 AND r0 r3 5 XOR r0 r1 r0 r2 r3 r4	0 MOV r4 r0 1 AND r0 r1 2 OR r0 r2 3 AND r2 r1 4 XOR r2 r3 5 AND r3 r1 6 XOR r0 r3 7 OR r3 r4 8 XOR r2 r3 r0 r1 r2 r4	0 MOV r4 r0 1 AND r0 r1 2 AND r0 r2 3 XOR r0 r3 4 AND r3 r1 5 OR r3 r2 6 XOR r3 r4 r0 r1 r2 r3
S-box 295	**S-box 296**	**S-box 297**	**S-box 298**	**S-box 299**	**S-box 300**
0 MOV r4 r0 1 AND r0 r1 2 AND r0 r2 3 XOR r0 r3 4 OR r3 r1 5 XOR r2 r3 r0 r1 r2 r4	0 MOV r4 r0 1 AND r0 r1 2 OR r0 r2 3 XOR r0 r3 4 AND r3 r1 5 AND r3 r4 6 XOR r2 r3 r0 r1 r2 r4	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 AND r2 r1 4 XOR r2 r3 r0 r1 r2 r4	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 OR r2 r1 4 XOR r2 r4 r0 r1 r2 r3	0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 3 AND r2 r0 4 XOR r2 r3 r0 r1 r2 r4	0 MOV r4 r0 1 AND r0 r1 2 AND r0 r2 3 XOR r0 r3 r0 r1 r2 r4
S-box 301	**S-box 302**				
0 MOV r4 r0 1 AND r0 r1 2 XOR r0 r2 r0 r1 r3 r4	r0 r1 r2 r3				

Appendix B

Extended table of all classes

In the following tables the representatives are listed. Next to the linear (c), the non-linear (p) histogram and the implementation cost we present the size of the class, the branch number (bn) of the representative, the maximum algebraic degree of the output bits (deg) and the class of the inverses of the s-boxes (inv).

TABLE B.1: Most efficient implementations with additional details 1–30

	Representative	$\lvert c\rvert$ = 1/4	1/2	3/4	1	p = 1/8	1/4	3/8	1/2	5/8	3/4	7/8	1	cost	size of class	bn	deg	inv
1	?	120	30	0	1	90	15	0	0	0	0	0	1	?	104 044 953 600	?	3	?
2	?	120	30	0	1	90	15	0	0	0	0	0	1	?	26 011 238 400	?	3	?
3	?	120	30	0	1	90	15	0	0	0	0	0	1	?	1 734 082 560	?	3	?
4	?	120	30	0	1	90	15	0	0	0	0	0	1	?	26 011 238 400	?	3	?
5	?	120	30	0	1	90	15	0	0	0	0	0	1	?	20 808 990 720	?	3	?
6	?	120	30	0	1	90	15	0	0	0	0	0	1	?	8 670 412 800	?	3	?
7	?	120	30	0	1	90	15	0	0	0	0	0	1	?	20 808 990 720	?	3	?
8	?	120	30	0	1	90	15	0	0	0	0	0	1	?	20 808 990 720	?	3	?
9	0cabf9d4e8635172	112	32	0	1	84	18	0	0	0	0	0	1	11	104 044 953 600	2	3	10
10	01298bd7cfe654a3	112	32	0	1	84	18	0	0	0	0	0	1	12	104 044 953 600	2	3	9
11	0a43562edfb1c789	112	32	0	1	84	18	0	0	0	0	0	1	13	104 044 953 600	2	3	11
12	?	112	32	0	1	84	18	0	0	0	0	0	1	?	104 044 953 600	?	3	?
13	086d5f7c4e2391ba	96	36	0	1	72	24	0	0	0	0	0	1	9	26 011 238 400	2	3	13
14	086c7e5f4d21b39a	96	36	0	1	72	24	0	0	0	0	0	1	10	26 011 238 400	2	3	15
15	0845d7fec6a391b2	96	36	0	1	72	24	0	0	0	0	0	1	10	26 011 238 400	2	3	14
16	01a2987cdef4563b	96	36	0	1	72	24	0	0	0	0	0	1	11	26 011 238 400	2	3	16
17	?	120	30	0	1	93	12	1	0	0	0	0	1	?	104 044 953 600	?	3	?
18	02839b7eca65df14	112	32	0	1	87	15	1	0	0	0	0	1	12	104 044 953 600	2	3	20
19	04afb6372e81c95d	112	32	0	1	87	15	1	0	0	0	0	1	12	104 044 953 600	2	3	?
20	02415f3e8bc6a9d7	112	32	0	1	87	15	1	0	0	0	0	1	12	104 044 953 600	2	2	18
21	0251c6afd7984e3b	112	32	0	1	87	15	1	0	0	0	0	1	13	104 044 953 600	2	3	21
22	?	112	32	0	1	87	15	1	0	0	0	0	1	?	104 044 953 600	?	3	?
23	?	120	30	0	1	96	9	2	0	0	0	0	1	?	26 011 238 400	?	3	?
24	?	120	30	0	1	96	9	2	0	0	0	0	1	?	26 011 238 400	?	3	?
25	0c69735248af1dbe	96	36	0	1	78	18	2	0	0	0	0	1	11	52 022 476 800	2	2	26
26	06a953b842c7df1e	96	36	0	1	78	18	2	0	0	0	0	1	11	52 022 476 800	2	2	25
27	0a2387bfc5de4961	96	36	0	1	78	18	2	0	0	0	0	1	12	52 022 476 800	2	2	29
28	0a2387bf4d56c1e9	96	36	0	1	78	18	2	0	0	0	0	1	12	52 022 476 800	2	2	28
29	0913a4bf2e6587dc	96	36	0	1	78	18	2	0	0	0	0	1	12	52 022 476 800	2	3	27
30	06af7d5e48c391b2	96	36	0	1	81	15	3	0	0	0	0	1	11	104 044 953 600	2	3	30

Table B.2: Most efficient implementations with additional details 31–84

	Representative	$\lvert c\rvert = 1/4$	1/2	3/4	1	$p = 1/8$	1/4	3/8	1/2	5/8	3/4	7/8	1	cost	size of class	bn	deg	inv
31	04598ceb6a72f3d1	96	36	0	1	80	18	0	1	0	0	0	1	11	26 011 238 400	2	3	31
32	08a319f4c6e5d7b2	64	44	0	1	64	24	0	2	0	0	0	1	9	13 005 619 200	2	3	33
33	086d5f7e4c2193ba	64	44	0	1	64	24	0	2	0	0	0	1	9	13 005 619 200	2	2	32
34	?	119	28	1	1	78	21	0	0	0	0	0	1	?	14 863 564 800	?	3	?
35	?	119	28	1	1	78	21	0	0	0	0	0	1	?	104 044 953 600	?	3	?
36	?	119	28	1	1	78	21	0	0	0	0	0	1	?	104 044 953 600	?	3	?
37	03298bd5efc476a1	111	30	1	1	72	24	0	0	0	0	0	1	11	52 022 476 800	2	3	37
38	03d74f985ec621ab	111	30	1	1	72	24	0	0	0	0	0	1	12	52 022 476 800	2	3	38
39	0e8952d7ca4b61f3	119	28	1	1	81	18	1	0	0	0	0	1	13	104 044 953 600	2	3	39
40	0283db4eca769f15	119	28	1	1	81	18	1	0	0	0	0	1	13	104 044 953 600	2	3	40
41	?	119	28	1	1	81	18	1	0	0	0	0	1	?	104 044 953 600	?	3	?
42	?	119	28	1	1	81	18	1	0	0	0	0	1	?	104 044 953 600	?	3	?
43	0c2784fa5961e3db	111	30	1	1	75	21	1	0	0	0	0	1	12	104 044 953 600	2	2	44
44	0c4d9fba8e635172	111	30	1	1	75	21	1	0	0	0	0	1	12	104 044 953 600	2	3	43
45	06abc84e79f2d153	111	30	1	1	75	21	1	0	0	0	0	1	12	104 044 953 600	2	3	45
46	0d9163e5fb7ac842	119	28	1	1	84	15	2	0	0	0	0	1	13	104 044 953 600	2	3	?
47	?	119	28	1	1	84	15	2	0	0	0	0	1	?	104 044 953 600	?	3	?
48	?	119	28	1	1	84	15	2	0	0	0	0	1	?	104 044 953 600	?	3	?
49	?	119	28	1	1	84	15	2	0	0	0	0	1	?	104 044 953 600	?	3	?
50	?	119	28	1	1	84	15	2	0	0	0	0	1	?	104 044 953 600	?	3	?
51	0283db7eca659f14	111	30	1	1	78	18	2	0	0	0	0	1	11	104 044 953 600	2	3	63
52	0285cf4b9a36de71	111	30	1	1	78	18	2	0	0	0	0	1	11	104 044 953 600	2	3	59
53	0c3e97af86d4512b	111	30	1	1	78	18	2	0	0	0	0	1	12	104 044 953 600	2	3	70
54	038a75dbcf6e1294	111	30	1	1	78	18	2	0	0	0	0	1	12	104 044 953 600	2	3	54
55	038a64dbcf7e1295	111	30	1	1	78	18	2	0	0	0	0	1	12	104 044 953 600	2	3	55
56	0481e37d6afbc952	111	30	1	1	78	18	2	0	0	0	0	1	12	104 044 953 600	2	3	57
57	04987bcf6ad251e3	111	30	1	1	78	18	2	0	0	0	0	1	12	104 044 953 600	2	3	56
58	0c2db39a6e857f14	111	30	1	1	78	18	2	0	0	0	0	1	12	104 044 953 600	2	3	58
59	086e7d5c4f21b39a	111	30	1	1	78	18	2	0	0	0	0	1	12	104 044 953 600	2	3	52
60	0cf1634b9d25a78e	111	30	1	1	78	18	2	0	0	0	0	1	12	104 044 953 600	2	3	66
61	0a24193685def7bc	111	30	1	1	78	18	2	0	0	0	0	1	12	52 022 476 800	2	3	62
62	0a23486519dcfb7e	111	30	1	1	78	18	2	0	0	0	0	1	12	52 022 476 800	2	3	61
63	04f28d617be3c95a	111	30	1	1	78	18	2	0	0	0	0	1	13	104 044 953 600	2	3	51
64	0cfbae138594726d	111	30	1	1	78	18	2	0	0	0	0	1	13	104 044 953 600	2	3	?
65	0debaf129584736c	111	30	1	1	78	18	2	0	0	0	0	1	13	104 044 953 600	2	3	65
66	07d9af54bec86231	111	30	1	1	78	18	2	0	0	0	0	1	13	104 044 953 600	2	3	60
67	0ae36592748cdf1b	111	30	1	1	78	18	2	0	0	0	0	1	13	104 044 953 600	2	3	?
68	086293efc7b5d4a1	111	30	1	1	78	18	2	0	0	0	0	1	13	104 044 953 600	2	3	73
69	04816aced372f95b	111	30	1	1	78	18	2	0	0	0	0	1	13	104 044 953 600	2	3	69
70	0281df5bce679a34	111	30	1	1	78	18	2	0	0	0	0	1	13	104 044 953 600	2	3	53
71	0182cf6ade579b34	111	30	1	1	78	18	2	0	0	0	0	1	13	104 044 953 600	2	3	71
72	0283db7fca659e14	111	30	1	1	78	18	2	0	0	0	0	1	13	104 044 953 600	2	3	?
73	0243d6aec7b95f18	111	30	1	1	78	18	2	0	0	0	0	1	13	104 044 953 600	2	3	68
74	?	111	30	1	1	78	18	2	0	0	0	0	1	?	104 044 953 600	?	3	?
75	?	111	30	1	1	78	18	2	0	0	0	0	1	?	104 044 953 600	?	3	?
76	?	111	30	1	1	78	18	2	0	0	0	0	1	?	104 044 953 600	?	3	?
77	0bf36482759cde1a	111	30	1	1	81	15	3	0	0	0	0	1	13	104 044 953 600	2	3	77
78	?	111	30	1	1	81	15	3	0	0	0	0	1	?	104 044 953 600	?	3	?
79	08e42ac1d7f5b396	95	34	1	1	72	18	4	0	0	0	0	1	11	104 044 953 600	2	3	80
80	04693fd17b52eac8	95	34	1	1	72	18	4	0	0	0	0	1	11	104 044 953 600	2	3	79
81	09e65cf74d82ba31	95	34	1	1	72	18	4	0	0	0	0	1	11	104 044 953 600	2	2	82
82	0c6bd9f2e8a51734	95	34	1	1	72	18	4	0	0	0	0	1	11	104 044 953 600	2	3	81
83	08c52ae1d4f6b397	95	34	1	1	72	18	4	0	0	0	0	1	12	104 044 953 600	2	3	83
84	04ae8c219fbd5376	63	42	1	1	78	0	14	0	0	0	0	1	10	14 863 564 800	2	3	84

TABLE B.3: Most efficient implementations with additional details 85–138

	Representative	$\|c\|$ = 1/4	1/2	3/4	1	p = 1/8	1/4	3/8	1/2	5/8	3/4	7/8	1	cost	size of class	bn	deg	inv
85	0dbea6372f91c845	111	30	1	1	80	18	0	1	0	0	0	1	12	104 044 953 600	2	2	85
86	0ea62c93f4d587b1	111	30	1	1	80	18	0	1	0	0	0	1	13	104 044 953 600	2	3	86
87	0913b2c486ed5a7f	118	26	2	1	72	21	2	0	0	0	0	1	13	104 044 953 600	2	3	87
88	0d7c2a186e5fb349	118	26	2	1	72	21	2	0	0	0	0	1	13	104 044 953 600	2	3	89
89	0c6749be1532f8da	118	26	2	1	72	21	2	0	0	0	0	1	13	104 044 953 600	2	3	88
90	0e2f84acb7d65319	118	26	2	1	72	21	2	0	0	0	0	1	13	104 044 953 600	2	3	90
91	0329d7e8f4c51ba6	118	26	2	1	72	21	2	0	0	0	0	1	13	104 044 953 600	2	3	91
92	?	118	26	2	1	72	21	2	0	0	0	0	1	?	104 044 953 600	?	3	?
93	0c2dbf16ae497358	110	28	2	1	66	24	2	0	0	0	0	1	11	52 022 476 800	2	2	95
94	095f18e4a7b3d2c6	110	28	2	1	66	24	2	0	0	0	0	1	11	52 022 476 800	2	2	94
95	04e8ca639bfd5172	110	28	2	1	66	24	2	0	0	0	0	1	12	52 022 476 800	2	3	93
96	0bd57f243a18e6c9	118	26	2	1	75	18	3	0	0	0	0	1	13	104 044 953 600	2	3	96
97	0913b24ca6ed587f	118	26	2	1	75	18	3	0	0	0	0	1	13	104 044 953 600	2	3	97
98	0425be968fdc731a	118	26	2	1	75	18	3	0	0	0	0	1	13	104 044 953 600	2	3	98
99	0724ae85bfdc6319	118	26	2	1	75	18	3	0	0	0	0	1	13	52 022 476 800	2	3	99
100	01b2c5e3f7d689a4	118	26	2	1	75	18	3	0	0	0	0	1	13	104 044 953 600	2	3	102
101	0ac5d736f4b912e8	118	26	2	1	75	18	3	0	0	0	0	1	13	52 022 476 800	2	3	101
102	09657cade4831bf2	118	26	2	1	75	18	3	0	0	0	0	1	13	104 044 953 600	2	3	100
103	0821e7ca6fd593b4	118	26	2	1	75	18	3	0	0	0	0	1	13	104 044 953 600	2	3	104
104	012bd4e3c7f598a6	118	26	2	1	75	18	3	0	0	0	0	1	13	104 044 953 600	2	3	103
105	?	118	26	2	1	75	18	3	0	0	0	0	1	?	104 044 953 600	?	3	?
106	0a387f496edcb125	110	28	2	1	69	21	3	0	0	0	0	1	11	104 044 953 600	2	3	109
107	0425bd968ecf731a	110	28	2	1	69	21	3	0	0	0	0	1	11	104 044 953 600	2	3	115
108	03298bd5cfe476a1	110	28	2	1	69	21	3	0	0	0	0	1	11	104 044 953 600	2	3	111
109	086e5c7d4f2391ba	110	28	2	1	69	21	3	0	0	0	0	1	11	104 044 953 600	2	3	106
110	06853d942cab71fe	110	28	2	1	69	21	3	0	0	0	0	1	11	104 044 953 600	2	3	112
111	0bd74f985ec621a3	110	28	2	1	69	21	3	0	0	0	0	1	12	104 044 953 600	2	3	108
112	06e915dbf37a42c8	110	28	2	1	69	21	3	0	0	0	0	1	12	104 044 953 600	2	3	110
113	0ec9731dfb52a486	110	28	2	1	69	21	3	0	0	0	0	1	12	104 044 953 600	2	3	114
114	08a1f356e24db79c	110	28	2	1	69	21	3	0	0	0	0	1	12	104 044 953 600	2	3	113
115	0942563718fdabec	110	28	2	1	69	21	3	0	0	0	0	1	12	104 044 953 600	2	3	107
116	0c8962e5fb7a41d3	118	26	2	1	78	15	4	0	0	0	0	1	13	104 044 953 600	2	3	116
117	?	118	26	2	1	78	15	4	0	0	0	0	1	?	52 022 476 800	?	3	?
118	?	118	26	2	1	78	15	4	0	0	0	0	1	?	52 022 476 800	?	3	?
119	048e26c31f957bda	110	28	2	1	72	18	4	0	0	0	0	1	12	104 044 953 600	2	3	120
120	04cae86bf1d35972	110	28	2	1	72	18	4	0	0	0	0	1	12	104 044 953 600	2	3	119
121	03a1df79ec658b24	110	28	2	1	72	18	4	0	0	0	0	1	12	104 044 953 600	2	3	124
122	0281ce6bdf549a37	110	28	2	1	72	18	4	0	0	0	0	1	12	104 044 953 600	2	3	129
123	0281ce7bdf459a36	110	28	2	1	72	18	4	0	0	0	0	1	12	104 044 953 600	2	3	128
124	0d14376cfae2958b	110	28	2	1	72	18	4	0	0	0	0	1	12	104 044 953 600	2	3	121
125	0b12756acfe4938d	110	28	2	1	72	18	4	0	0	0	0	1	12	104 044 953 600	2	3	127
126	08f5b1e42ac3d697	110	28	2	1	72	18	4	0	0	0	0	1	12	34 681 651 200	2	3	126
127	02418ae693fcd7b5	110	28	2	1	72	18	4	0	0	0	0	1	12	104 044 953 600	2	3	125
128	0bd74f91c65ea823	110	28	2	1	72	18	4	0	0	0	0	1	13	104 044 953 600	2	3	123
129	08e52ac1f4d693b7	110	28	2	1	72	18	4	0	0	0	0	1	13	104 044 953 600	2	3	122
130	08a2d5e3f6c791b4	118	26	2	1	74	21	0	1	0	0	0	1	13	104 044 953 600	2	3	130
131	0d91ea6572f3c84b	118	26	2	1	77	18	1	1	0	0	0	1	12	104 044 953 600	2	3	131
132	0481e37dfa6b59c2	110	28	2	1	71	21	1	1	0	0	0	1	12	104 044 953 600	2	3	133
133	0c86d352f74e19ba	110	28	2	1	71	21	1	1	0	0	0	1	12	104 044 953 600	2	3	132
134	0829b71ea64df35c	110	28	2	1	74	18	2	1	0	0	0	1	11	26 011 238 400	2	2	135
135	0c635172e8abf9d4	110	28	2	1	74	18	2	1	0	0	0	1	11	26 011 238 400	2	3	134
136	04e935fb71d86ac2	110	28	2	1	74	18	2	1	0	0	0	1	12	104 044 953 600	2	3	137
137	0ac9f75d1b32e684	110	28	2	1	74	18	2	1	0	0	0	1	12	104 044 953 600	2	3	136
138	0591e26d7afbc843	110	28	2	1	74	18	2	1	0	0	0	1	12	104 044 953 600	2	3	138

Table B.4: Most efficient implementations with additional details 139–192

	Representative	$\lvert c\rvert$	= 1/4	1/2	3/4	1	p = 1/8	1/4	3/8	1/2	5/8	3/4	7/8	1	cost	size of class	bn	deg	inv
139	08f43bd6912c7e5a		110	28	2	1	74	18	2	1	0	0	0	1	12	26 011 238 400	2	2	139
140	0821f396ea45b7dc		110	28	2	1	74	18	2	1	0	0	0	1	12	104 044 953 600	2	3	141
141	0f415a6b97d2e8c3		110	28	2	1	74	18	2	1	0	0	0	1	12	104 044 953 600	2	3	140
142	0283df7ace659b14		94	32	2	1	62	24	2	1	0	0	0	1	9	52 022 476 800	2	3	143
143	0a6d5f7c4e2391b8		94	32	2	1	62	24	2	1	0	0	0	1	9	52 022 476 800	2	2	142
144	04ac8e239fbd5176		94	32	2	1	62	24	2	1	0	0	0	1	10	52 022 476 800	2	3	145
145	0821f6dac5e4b397		94	32	2	1	62	24	2	1	0	0	0	1	10	52 022 476 800	2	3	144
146	0814d5ea7f62b3c9		94	32	2	1	62	24	2	1	0	0	0	1	10	52 022 476 800	2	3	147
147	0425b796aec1f3d8		94	32	2	1	62	24	2	1	0	0	0	1	10	52 022 476 800	2	2	146
148	04617b5a8ce3d9f2		94	32	2	1	62	24	2	1	0	0	0	1	10	52 022 476 800	2	2	150
149	0c63d1fae82597b4		94	32	2	1	62	24	2	1	0	0	0	1	10	52 022 476 800	2	2	149
150	0e6bd9f2c8a51734		94	32	2	1	62	24	2	1	0	0	0	1	10	52 022 476 800	2	3	148
151	04af8d21be9c7356		94	32	2	1	62	24	2	1	0	0	0	1	11	52 022 476 800	2	3	153
152	0c81bf53d9762ea4		94	32	2	1	62	24	2	1	0	0	0	1	11	52 022 476 800	2	3	152
153	0a2395b8d7f6c4e1		94	32	2	1	62	24	2	1	0	0	0	1	11	52 022 476 800	2	3	151
154	0824f6ae5d7391cb		94	32	2	1	62	24	2	1	0	0	0	1	11	52 022 476 800	2	3	154
155	04639fd2e8cb517a		94	32	2	1	64	24	0	2	0	0	0	1	10	52 022 476 800	2	3	155
156	0c69a24ef758b31d		117	24	3	1	66	21	4	0	0	0	0	1	12	104 044 953 600	2	3	156
157	08296c5a4fdeb317		117	24	3	1	66	21	4	0	0	0	0	1	13	34 681 651 200	2	3	157
158	0bd57f26183ac4e9		117	24	3	1	69	18	5	0	0	0	0	1	13	104 044 953 600	2	3	158
159	08e5c7a4b391f6d2		117	24	3	1	69	18	5	0	0	0	0	1	13	104 044 953 600	2	3	160
160	08a9f65db217e3c4		117	24	3	1	69	18	5	0	0	0	0	1	13	104 044 953 600	2	3	159
161	0a8b46d2ce7f1395		117	24	3	1	69	18	5	0	0	0	0	1	13	104 044 953 600	2	3	161
162	08235cfae64719bd		117	24	3	1	69	18	5	0	0	0	0	1	13	104 044 953 600	2	3	162
163	0ac46e251b39f7d8		109	26	3	1	63	21	5	0	0	0	0	1	11	104 044 953 600	2	3	166
164	08e4c6a591b3d7f2		109	26	3	1	63	21	5	0	0	0	0	1	11	104 044 953 600	2	3	165
165	0ce53b91d7f284a6		109	26	3	1	63	21	5	0	0	0	0	1	11	104 044 953 600	2	3	164
166	046abec9f8d27351		109	26	3	1	63	21	5	0	0	0	0	1	11	104 044 953 600	2	3	163
167	0823b79ad5f64ce1		109	26	3	1	63	21	5	0	0	0	0	1	11	104 044 953 600	2	3	169
168	0a4e86c13bd5f792		109	26	3	1	63	21	5	0	0	0	0	1	12	104 044 953 600	2	3	168
169	06e842c397f5b1da		109	26	3	1	63	21	5	0	0	0	0	1	12	104 044 953 600	2	3	167
170	05bf8d349cae6217		109	26	3	1	63	21	5	0	0	0	0	1	12	104 044 953 600	2	3	170
171	0291e8a3f6d5b7c4		117	24	3	1	72	15	6	0	0	0	0	1	12	104 044 953 600	2	3	171
172	068cea2db7951f34		109	26	3	1	66	18	6	0	0	0	0	1	12	104 044 953 600	2	3	172
173	08235cfae74619bd		109	26	3	1	66	18	6	0	0	0	0	1	12	104 044 953 600	2	3	173
174	0219d7e3c6f48ab5		109	26	3	1	66	18	6	0	0	0	0	1	12	104 044 953 600	2	3	174
175	03a1ec69df748b25		109	26	3	1	66	18	6	0	0	0	0	1	12	104 044 953 600	2	3	177
176	0283ca6edb749f15		109	26	3	1	66	18	6	0	0	0	0	1	12	104 044 953 600	2	3	176
177	032476a5cfe98bd1		109	26	3	1	66	18	6	0	0	0	0	1	12	104 044 953 600	2	3	175
178	0289f75a6c4d3b1e		109	26	3	1	69	15	7	0	0	0	0	1	12	104 044 953 600	2	3	178
179	0ea3c84671f2d95b		117	24	3	1	65	24	1	1	0	0	0	1	12	104 044 953 600	2	3	179
180	072634bc58f9e1ad		117	24	3	1	65	24	1	1	0	0	0	1	13	104 044 953 600	2	3	181
181	06b3e9c1fa4d2785		117	24	3	1	65	24	1	1	0	0	0	1	13	104 044 953 600	2	3	180
182	09324debf75618ac		117	24	3	1	68	21	2	1	0	0	0	1	13	104 044 953 600	2	3	182
183	0e42a6c3fbd97158		109	26	3	1	62	24	2	1	0	0	0	1	11	104 044 953 600	2	3	184
184	047bafe9d8c36251		109	26	3	1	62	24	2	1	0	0	0	1	11	104 044 953 600	2	3	183
185	041dbea5267fc983		109	26	3	1	62	24	2	1	0	0	0	1	11	52 022 476 800	2	3	185
186	0481eb7d62f3c95a		109	26	3	1	62	24	2	1	0	0	0	1	11	52 022 476 800	2	3	186
187	0ce53b91f7d4a286		109	26	3	1	62	24	2	1	0	0	0	1	12	52 022 476 800	2	3	187
188	0a3b29c5fe4d6781		109	26	3	1	62	24	2	1	0	0	0	1	12	52 022 476 800	2	2	188
189	0ea342c6fb78d951		117	24	3	1	80	9	6	1	0	0	0	1	12	104 044 953 600	2	3	189
190	0285ca4e9f36db71		109	26	3	1	74	12	6	1	0	0	0	1	11	104 044 953 600	2	3	190
191	06c18a4edf329b75		109	26	3	1	74	12	6	1	0	0	0	1	11	104 044 953 600	2	3	191
192	08e64c29d7f51b3a		93	30	3	1	65	15	7	1	0	0	0	1	10	104 044 953 600	2	3	193

TABLE B.5: Most efficient implementations with additional details 193–246

	Representative	$\|c\|$	$= 1/4$	1/2	3/4	1	$p = 1/8$	1/4	3/8	1/2	5/8	3/4	7/8	1	cost	size of class	bn	deg	inv
193	08e6c4a1d7f593b2		93	30	3	1	65	15	7	1	0	0	0	1	10	104 044 953 600	2	3	192
194	08a1e2c6f7d4b395		116	22	4	1	60	21	6	0	0	0	0	1	12	104 044 953 600	2	3	194
195	0e6c84a17fd5b392		116	22	4	1	60	21	6	0	0	0	0	1	13	26 011 238 400	2	3	195
196	0d786e5c2a1fb349		116	22	4	1	60	21	6	0	0	0	0	1	13	26 011 238 400	2	3	196
197	0938e64bf75ca21d		116	22	4	1	63	18	7	0	0	0	0	1	12	104 044 953 600	2	3	197
198	08a1e6c3f7d4b295		116	22	4	1	66	15	8	0	0	0	0	1	12	104 044 953 600	2	3	198
199	08a1e6d3f7c5b294		116	22	4	1	66	15	8	0	0	0	0	1	12	104 044 953 600	2	3	199
200	0c6348aef71259bd		116	22	4	1	59	24	3	1	0	0	0	1	12	104 044 953 600	2	3	200
201	08e64c2bd7f5193a		108	24	4	1	56	24	4	1	0	0	0	1	11	104 044 953 600	2	3	201
202	08ce4623f5d791ba		108	24	4	1	56	24	4	1	0	0	0	1	11	52 022 476 800	2	3	203
203	024ce6a97f5d1b38		108	24	4	1	56	24	4	1	0	0	0	1	11	52 022 476 800	2	3	202
204	04a8ec23dbf95176		108	24	4	1	56	24	4	1	0	0	0	1	11	52 022 476 800	2	3	204
205	0ac46e2d1b397f58		108	24	4	1	62	18	6	1	0	0	0	1	11	104 044 953 600	2	3	206
206	08a17b95f3d2c6e4		108	24	4	1	62	18	6	1	0	0	0	1	11	104 044 953 600	2	3	205
207	0759aec8fbd26341		108	24	4	1	62	18	6	1	0	0	0	1	11	104 044 953 600	2	3	208
208	08297d5a4cefb316		108	24	4	1	62	18	6	1	0	0	0	1	11	104 044 953 600	2	3	207
209	02a846c397b1f5de		108	24	4	1	62	18	6	1	0	0	0	1	12	104 044 953 600	2	3	209
210	06ac24e397b5d1f8		108	24	4	1	62	18	6	1	0	0	0	1	12	104 044 953 600	2	3	211
211	0ac62e83b795f1d4		108	24	4	1	62	18	6	1	0	0	0	1	12	104 044 953 600	2	3	210
212	0d98ea65fb7341c2		116	22	4	1	73	12	5	2	0	0	0	1	12	104 044 953 600	2	3	212
213	0283de7bcf659a14		108	24	4	1	67	15	5	2	0	0	0	1	11	104 044 953 600	2	3	214
214	0392ce7bdf648a15		108	24	4	1	67	15	5	2	0	0	0	1	11	104 044 953 600	2	3	213
215	0281df7ace459b36		92	28	4	1	48	30	0	3	0	0	0	1	9	26 011 238 400	2	3	216
216	086f5d7e4c29b31a		92	28	4	1	48	30	0	3	0	0	0	1	9	26 011 238 400	2	2	215
217	0283de7bcf459a16		115	20	5	1	56	21	6	1	0	0	0	1	12	104 044 953 600	2	3	217
218	0829f75ae64db31c		107	22	5	1	52	24	4	2	0	0	0	1	9	104 044 953 600	2	3	218
219	0823b79ad5f6c4e1		107	22	5	1	52	24	4	2	0	0	0	1	10	104 044 953 600	2	3	219
220	08a2d5f3e7c691b4		115	20	5	1	64	15	6	2	0	0	0	1	12	104 044 953 600	2	3	220
221	08a2c4e597b1d3f6		107	22	5	1	58	18	6	2	0	0	0	1	11	104 044 953 600	2	3	221
222	08e6c4a197f5d3b2		107	22	5	1	58	18	6	2	0	0	0	1	11	104 044 953 600	2	3	222
223	08a64ce75df1b293		114	18	6	1	63	6	15	0	0	0	0	1	12	17 340 825 600	2	3	223
224	0462e8c3715bf9da		114	18	6	1	63	6	15	0	0	0	0	1	13	5 780 275 200	2	3	224
225	08465dbce7a291f3		114	18	6	1	54	21	4	3	0	0	0	1	12	104 044 953 600	2	3	225
226	08a1d5f3c4e6b297		114	18	6	1	54	21	4	3	0	0	0	1	12	34 681 651 200	2	3	226
227	0abd4ef9823657c1		106	20	6	1	54	18	6	3	0	0	0	1	10	26 011 238 400	2	2	227
228	048c62e315f97bda		114	18	6	1	69	6	9	3	0	0	0	1	13	34 681 651 200	2	3	228
229	0823d7fa4ce591b6		106	20	6	1	63	9	9	3	0	0	0	1	11	104 044 953 600	2	3	229
230	0c69a24ef718b35d		113	16	7	1	62	9	8	4	0	0	0	1	11	104 044 953 600	2	3	230
231	0938e65bf74ca21d		110	10	10	1	65	0	5	10	0	0	0	1	11	20 808 990 720	2	3	231
232	092e7456cdfb83a1		94	32	2	1	66	23	1	0	1	0	0	1	11	52 022 476 800	2	3	232
233	03fc56ed47928a1b		94	32	2	1	66	23	1	0	1	0	0	1	11	52 022 476 800	2	2	233
234	097e4d6f5c38a21b		109	26	3	1	66	23	1	0	1	0	0	1	12	52 022 476 800	2	2	234
235	06ac8e239fbd5174		92	28	4	1	60	17	7	0	1	0	0	1	10	104 044 953 600	2	3	235
236	06ac8e219fbd7354		92	28	4	1	60	17	7	0	1	0	0	1	10	104 044 953 600	2	3	236
237	08e64c29f7d51b3a		107	22	5	1	48	29	3	0	1	0	0	1	11	52 022 476 800	2	3	237
238	0921b3d5f7a86e4c		107	22	5	1	48	29	3	0	1	0	0	1	12	17 340 825 600	2	3	238
239	08e64c29d5f71b3a		107	22	5	1	54	23	5	0	1	0	0	1	11	34 681 651 200	2	3	239
240	0ac46e29f7d53b18		107	22	5	1	60	17	7	0	1	0	0	1	11	104 044 953 600	2	3	240
241	08a719b35df6e2c4		105	18	7	1	58	11	9	2	1	0	0	1	11	104 044 953 600	2	3	241
242	0a23f7d86ec591b4		105	18	7	1	58	11	9	2	1	0	0	1	11	52 022 476 800	2	3	242
243	086d5f7ec4291b3a		62	40	2	1	48	33	0	0	0	1	0	1	9	17 340 825 600	2	2	243
244	08e64c2bf7d5193a		90	24	6	1	42	27	6	0	0	1	0	1	10	34 681 651 200	2	3	244
245	084a6e1d5c397f2b		112	28	0	2	57	21	7	0	0	0	0	1	12	14 863 564 800	2	3	245
246	08ab193246cf7d5e		96	32	0	2	51	21	9	0	0	0	0	1	10	52 022 476 800	2	2	246

Table B.6: Most efficient implementations with additional details 247–302

	Representative	$\lvert c\rvert = 1/4$	1/2	3/4	1	$p = 1/8$	1/4	3/8	1/2	5/8	3/4	7/8	1	cost	size of class	bn	deg	inv
247	0ba981234fe6dc57	112	28	0	2	50	30	2	1	0	0	0	1	11	26 011 238 400	2	2	247
248	0c2f1d7a48693b5e	96	32	0	2	44	30	4	1	0	0	0	1	9	13 005 619 200	2	2	251
249	012b89f7cde654a3	96	32	0	2	44	30	4	1	0	0	0	1	9	6 502 809 600	2	3	249
250	082b195d4f6e7c3a	96	32	0	2	44	30	4	1	0	0	0	1	10	13 005 619 200	2	3	250
251	024513768ecbf9da	96	32	0	2	44	30	4	1	0	0	0	1	10	13 005 619 200	2	2	248
252	0c7b3e6a2f4d5918	96	32	0	2	44	30	4	1	0	0	0	1	10	13 005 619 200	2	3	252
253	086f5d7e4c293b1a	64	40	0	2	32	36	0	4	0	0	0	1	7	6 502 809 600	2	2	256
254	086f5d7e4c2391ba	64	40	0	2	32	36	0	4	0	0	0	1	7	3 251 404 800	2	2	255
255	082b197c4e6d5f3a	64	40	0	2	32	36	0	4	0	0	0	1	8	3 251 404 800	2	3	254
256	046173528cebd9fa	64	40	0	2	32	36	0	4	0	0	0	1	8	6 502 809 600	2	2	253
257	086f5d7ec4a1b392	64	40	0	2	32	36	0	4	0	0	0	1	8	6 502 809 600	2	2	257
258	08a319f6c4e7d5b2	0	56	0	2	64	0	0	14	0	0	0	1	7	232 243 200	2	2	258
259	09f75d26183ac4eb	110	24	2	2	45	21	11	0	0	0	0	1	11	52 022 476 800	2	3	259
260	08a357dfb192c4e6	110	24	2	2	50	18	10	1	0	0	0	1	11	26 011 238 400	2	3	260
261	08a71df395b2c4e6	94	28	2	2	40	24	8	2	0	0	0	1	9	26 011 238 400	2	3	263
262	0459afebd8c16273	94	28	2	2	40	24	8	2	0	0	0	1	9	52 022 476 800	2	3	262
263	0812b3a95d46f7ce	94	28	2	2	40	24	8	2	0	0	0	1	9	26 011 238 400	2	2	261
264	04369ca78def512b	110	24	2	2	48	24	4	3	0	0	0	1	11	34 681 651 200	2	3	264
265	032547618bacfe9d	108	20	4	2	44	12	16	1	0	0	0	1	10	13 005 619 200	2	2	265
266	08297f5a6e4d3b1c	92	24	4	2	28	30	4	5	0	0	0	1	7	13 005 619 200	2	3	267
267	082b193a4ce5f7d6	92	24	4	2	28	30	4	5	0	0	0	1	8	13 005 619 200	2	2	266
268	082b3f1a5d7e4c69	92	24	4	2	28	30	4	5	0	0	0	1	8	13 005 619 200	2	3	268
269	0461dbf28ce9537a	56	28	8	1	0	56	0	0	0	0	0	2	9	309 657 600	2	3	269
270	092e1436cdfb85a7	96	32	0	2	48	29	3	0	1	0	0	1	11	17 340 825 600	2	3	270
271	082b1a6d5f7e4c39	96	32	0	2	48	29	3	0	1	0	0	1	11	8 670 412 800	2	3	271
272	046f953b1d7ec82a	110	24	2	2	36	35	3	0	1	0	0	1	11	8 670 412 800	2	3	272
273	0a28c6e53b91f7d4	92	24	4	2	40	17	11	2	1	0	0	1	9	52 022 476 800	2	3	273
274	082a4ce51b39d7f6	92	24	4	2	40	17	11	2	1	0	0	1	9	26 011 238 400	2	3	274
275	0a28c4e53b91f7d6	106	16	6	2	32	23	7	4	1	0	0	1	10	26 011 238 400	2	3	275
276	082ac4e519b3d7f6	104	12	8	2	42	11	5	9	1	0	0	1	10	17 340 825 600	2	3	276
277	082b7c5a496e3f1d	108	20	4	2	58	16	0	5	2	0	0	1	11	3 251 404 800	2	3	277
278	08a75db391f6c4e2	92	24	4	2	46	22	0	5	2	0	0	1	9	6 502 809 600	2	3	278
279	086e4c295d7f1b3a	90	20	6	2	42	9	15	0	3	0	0	1	9	17 340 825 600	2	3	279
280	082a4ce7193bf5d6	104	12	8	2	24	32	0	3	4	0	0	1	10	2 167 603 200	2	3	280
281	082a4ce5193bd7f6	104	12	8	2	48	8	8	3	4	0	0	1	10	4 335 206 400	2	3	281
282	082ac4e319b7d5f6	100	4	12	2	54	0	0	9	6	0	0	1	10	3 251 404 800	2	3	282
283	086e4c295d7f3b1a	62	36	2	2	36	21	12	0	0	1	0	1	8	17 340 825 600	2	3	283
284	04ae8c239dbf5176	62	36	2	2	36	21	12	0	0	1	0	1	8	13 005 619 200	2	3	284
285	08a35df2c4e791b6	60	32	4	2	32	27	0	7	0	1	0	1	7	6 502 809 600	2	3	285
286	08e64c2bd5f7193a	90	20	6	2	24	27	8	3	0	1	0	1	8	13 005 619 200	2	3	286
287	0823d5fa4ce791b6	88	16	8	2	38	13	8	4	2	1	0	1	8	13 005 619 200	2	3	287
288	046153728ce9dbfa	0	56	0	2	0	56	0	0	0	0	0	2	7	309 657 600	2	2	288
289	046b59728ce3d1fa	0	56	0	2	0	56	0	0	0	0	0	2	7	270 950 400	2	2	289
290	0463d9f28ceb517a	56	24	8	2	0	44	0	6	0	0	0	2	7	541 900 800	2	3	290
291	0127456389aedcbf	96	24	0	4	24	6	24	3	0	0	0	1	8	1 083 801 600	2	2	291
292	081b2a394c5e7f6d	64	32	0	4	0	36	0	12	0	0	0	1	6	1 625 702 400	2	2	292
293	0c2f1d7b483a596e	96	24	0	4	12	38	0	3	4	0	0	1	9	541 900 800	2	3	293
294	082ac4e719b3d5f6	88	8	8	4	12	20	0	9	4	2	0	1	7	1 625 702 400	2	3	294
295	086e4c2b5d7f193a	60	24	4	4	16	9	16	5	0	3	0	1	6	3 251 404 800	2	3	295
296	082b5d7f193e4c6a	84	0	12	4	36	3	0	0	12	3	0	1	7	361 267 200	2	3	296
297	082b197e4c6f5d3a	0	48	0	4	0	32	0	12	0	0	0	2	5	203 212 800	2	2	297
298	046351728cebd9fa	0	48	0	4	0	32	0	12	0	0	0	2	5	270 950 400	2	2	298
299	082b5d7a4c6f193e	56	16	8	4	0	26	0	12	0	2	0	2	5	270 950 400	2	3	299
300	082a4c6f193b5d7e	56	0	8	8	0	14	0	0	0	14	0	2	4	38 707 200	2	3	300
301	082b193a4c6f5d7e	0	32	0	8	0	0	0	24	0	0	0	4	3	33 868 800	2	2	301
302	0123456789abcdef	0	0	0	16	0	0	0	0	0	0	0	16	0	322 560	2	1	302

Bibliography

[1] R. Anderson, E. Biham, and L. Knudsen. Serpent: A Proposal for the Advanced Encryption Standard. In *in First Advanced Encryption Standard (AES) Conference*, 1998.

[2] Atmel. Section 1 8051 Microcontroller Instruction Set. URL: `www.atmel.com/atmel/acrobat/doc0509.pdf`, 2006.

[3] G. V. Bard. *Algebraic Cryptanalysis*. Springer Publishing Company, Incorporated, 2009.

[4] R. Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Inf.*, 1:290–306, 1972.

[5] E. Biham and A. Shamir. Differential Cryptanalysis of DES-like Cryptosystems. In *CRYPTO '90: Proceedings of the 10th Annual International Cryptology Conference on Advances in Cryptology*, pages 2–21, London, UK, 1991. Springer-Verlag.

[6] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In *CHES*, pages 450–466, 2007.

[7] Canonical. Ubuntu – Details of libavl-dev in karmic. URL: `http://packages.ubuntu.com/karmic/libavl-dev`, last visited 2010-04-17.

[8] A. Canteaut, C. Lauradoux, and A. Seznec. Understanding cache attacks. URL: `hal.inria.fr/docs/00/07/13/87/PDF/RR-5881.pdf`, 2006.

[9] J. Daemen, M. Peeters, G. Van Assche, and V. Rijmen. Nessie proposal: NOEKEON. URL: `gro.noekeon.org/Noekeon-spec.pdf`, 2000.

[10] J. Daemen and V. Rijmen. *The Design of Rijndael*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[11] J. F. Dillon. APN polynomials: an update. presented at International Conference on Finite fields and applications - Fq9, 2009.

[12] Y. Hatano and D. Watanabe. Higher Order Differential Attack on Step-Reduced Variants of Luffa. URL: `http://www.sdl.hitachi.co.jp/crypto/luffa/HigherOrderDifferentialAttackOnLuffa_v1_20090915.pdf`, 2010.

[13] H. M. Heys. A tutorial on linear and differential cryptanalysis. *Cryptologia*, 26(3):189–221, 2002.

[14] G. D. Knott. A balanced tree storage and retrieval algorithm. In *SIGIR '71: Proceedings of the 1971 international ACM SIGIR conference on Information storage and retrieval*, pages 175–196, New York, NY, USA, 1971. ACM.

[15] G. Leander and A. Poschmann. On the Classification of 4 Bit S-Boxes. In *WAIFI '07: Proceedings of the 1st international workshop on Arithmetic of Finite Fields*, pages 159–176, Berlin, Heidelberg, 2007. Springer-Verlag.

[16] M. Matsui and A. Yamagishi. A New Method for Known Plaintext Attack of FEAL Cipher. In *EUROCRYPT*, pages 81–91, 1992.

[17] C. De Cannière. *Analysis and design of symmetric encryption algorithms*. PhD thesis, K.U. Leuven, 2007.

[18] C. De Cannière, A. Biryukov, and B. Preneel. An Introduction to Block Cipher Cryptanalysis. *Proceedings of the IEEE*, 94(2):346–356, 2006.

[19] C. De Cannière, O. Dunkelman, and M. Knežević. KATAN and KTANTAN – A Family of Small and Efficient Hardware-Oriented Block Ciphers. In *CHES '09: Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems*, pages 272–288, Berlin, Heidelberg, 2009. Springer-Verlag.

[20] C. De Cannière, H. Sato, and D. Watanabe. Hash Function Luffa: Specification. Submission to NIST (Round 2), 2009.

[21] D. De Schreye. Artificiele Intelligentie. lecture notes, URL: http://www.cs.kuleuven.ac.be/~dannyd/AI_Leuven, 2010.

[22] H. C. van Tilborg. *Encyclopedia of Cryptography and Security*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[23] Vlaams Supercomputer Centrum. VIC3 User Manual. URL: https://vscentrum.be/vsc-help-center/reference-manuals/vic3-user-manual/referencemanual-all-pages, last visited 2010-04-17.

[24] D. A. Osvik. Speeding up Serpent. In *AES Candidate Conference*, pages 317–329, 2000.

[25] B. Pfaff. GNU libavl. URL: http://savannah.gnu.org/projects/avl, last visited 2010-05-14.

[26] B. Pfaff. Performance analysis of BSTs in system software. *SIGMETRICS Perform. Eval. Rev.*, 32(1):410–411, 2004.

[27] B. Schneier and J. Kelsey. Unbalanced feistel networks and block-cipher design. In *Fast Software Encryption, 3rd International Workshop Proceedings*, pages 121–144. Springer-Verlag, 1996.

[28] C. Shannon. Communication Theory of Secrecy Systems. *Bell System Technical Journal, Vol 28, pp. 656-715*, Oktober 1949.

[29] D. Watanabe. How to generate the Sbox of Luffa. presented at ESC2010@Remich, January 2010.