Performance and Security of Block Ciphers using Operations in $GF(2^n)$

Shiho Moriai, Takeshi Shimoyama

Research Project of Info-Communication Security,
Telecommunications Advancement Organization of Japan
1-1-32 Shin'urashima, Kanagawa-ku, Yokohama, 221 Japan

{shiho,shimo}@yokohama.tao.or.jp

Abstract. We present a block cipher that has the best performance of all 64-bit block ciphers with the smallest proven differential/linear probability 2^{-62} using the structure proposed by Matsui. One of features of his methodology is to use some power functions in $GF(2^n)$ with different n that are resistant to differential and linear attacks. In this paper we discuss how to combine power functions x^k in $GF(2^n)$ over $GF(2)$ with different k and n with the goal of designing block ciphers with high performance and security against known attacks.

Keywords. block cipher, provable security, extension of finite field, power functions, differential attack, linear attack, higher order differential attack

1 Introduction

Many methods were proposed for constructing practical and provably secure block ciphers. One of them is Matsui's structure[12] for block ciphers with provable security against differential and linear attacks. It is a modified Feistel cipher based on three principles: change of the location of round functions, round functions with recursive structure, and substitution boxes of different sizes. The first realizes parallel computation of the round functions without losing provable security, and the second reduces the size of substitution boxes, and the last is expected to make algebraic attacks difficult.

The feature to which we direct our attention is to use some power functions x^k in $GF(2^n)$ over $GF(2)$ with different k and n that are resistant to differential and linear attacks. In this paper we discuss how to combine power functions x^k in $GF(2^n)$ over $GF(2)$ with different k and n with the goal of designing block ciphers using Matsui's structure with high performance and security against known attacks. The known attacks include the algebraic attacks such as higher order differential attack[9, 6] as well as differential and linear attacks.

The block cipher MISTY, also proposed by Matsui[13], adopts this structure and is guaranteed to be provably secure against differential and linear attacks. The substitution boxes of MISTY are composite functions of some affine functions and power functions over $GF(2^7)$ and $GF(2^9)$ which are bijective, almost bent, and almost perfect nonlinear. However, we can construct many other ciphers with the same or higher level of security in terms of probability of differential and linear hull by using other power functions x^k in $GF(2^n)$ over $GF(2)$. Here we

also consider the encryption and decryption speed to select the exponent of the power function k and extension degree n.

In this paper we describe the results of: **1.** implementing cubic function (i.e. $k = 3$) in $GF(2^n)$ over $GF(2)$ for $3 \leq n \leq 65$ by using various algorithms to attain the best performance, and **2.** computing the encryption speed of all 64-bit block ciphers using cubic functions with the smallest proven differential/linear probability 2^{-62} by combining the results above.

Finally, we present a block cipher that has the best performance of all 64-bit block ciphers with the smallest proven differential/linear probability 2^{-62} using Matsui's structure. The countermeasure against higher order differential attack[6] is also discussed.

2 Preliminaries

2.1 Definitions

Definition 1 *Let $p(x)$ be an irreducible polynomial over $GF(2)$ with degree n. The extension field $GF(2)[x]/Id(p(x))$ is denoted by $GF(2^n)$, where $Id(p(x))$ is an ideal of $p(x)$. n is the extension degree.*

It is known that for every positive integer n there exists an irreducible polynomial with degree n. For finding irreducible polynomials over $GF(2)$, see the references [20, 21, 23, 24]. In Appendix B we list the irreducible polynomials over $GF(2^n)$ $(2 \leq n \leq 68)$, primitive elements, and normal generators used in our experiments.

Definition 2 *Let $F : GF(2^n) \rightarrow GF(2^n)$ be a function with an n-bit input $x = (x_{n-1}, \ldots, x_0)$ and an n-bit output $y = (y_{n-1}, \ldots, y_0)$. Each output bit is represented by a polynomial of input bits; $y_i = f(x_{n-1}, \ldots, x_0)$. We call it the coordinate function. Let $ord(f)$ be the algebraic order (total degree) of f.*

Definition 3 *Let $F : GF(2^n) \rightarrow GF(2^n)$ be an n-bit function. We define the differential probability $\mathsf{DP}(\Delta X \rightarrow \Delta Y)$ and linear probability $\mathsf{LP}(\Gamma Y \rightarrow \Gamma X)$ as follows;*

$$\begin{aligned} \mathsf{DP}(\Delta X \rightarrow \Delta Y) &= Prob_X\{F(X + \Delta X) = F(X) + \Delta Y\} \\ \mathsf{LP}(\Gamma_Y \rightarrow \Gamma_X) &= |2Prob_X\{X \cdot \Gamma_X = F(X) \cdot \Gamma_Y\} - 1|^2, \end{aligned}$$

where $+$ denotes bitwise exclusive-or, and $a \cdot b$ denotes the even parity of the bitwise product of a and b.

We also define the maximal differential probability $\mathsf{DP}_{\max}$ and the maximal linear probability $\mathsf{LP}_{\max}$ as follows;

$$\begin{aligned} \mathsf{DP}_{\max} &= \max_{\Delta X \neq 0, \Delta Y} \mathsf{DP}(\Delta X \rightarrow \Delta Y) \\ \mathsf{LP}_{\max} &= \max_{\Gamma_X, \Gamma_Y \neq 0} \mathsf{LP}(\Gamma_Y \rightarrow \Gamma_X). \end{aligned}$$

2.2 Functions with provable security against differential and linear attacks

Some power functions in $GF(2^n)$ are known to have minimal $\mathsf{DP}_{\max}$ and $\mathsf{LP}_{\max}$, i.e. they have provable security against differential and linear attacks (see Table 1 [7]).

$F(x)$	$ord(f)$	$\mathsf{DP}_{\max}$	$\mathsf{LP}_{\max}$	conditions
x^{2^t+1}	2	2^{s-n}		$s = \gcd(t, n)$
x^{2^t+1}	2		2^{s-n}	$s = \gcd(t, n)$
				n/s odd
x^{-1}	$n-1$	2^{1-n}	2^{1-n}	n odd
x^{-1}	$n-1$	2^{2-n}	2^{1-n}	n even

Table 1: Examples of power functions in $GF(2^n)$ over $GF(2)$ which attain minimal values of differential/linear probability

3 Algorithms for Computing Arithmetic Operations in $GF(2^n)$

$GF(2^n)$ is an n-dimensional vector space over $GF(2)$. Once a basis for $GF(2^n)$ over $GF(2)$ has been given, any element a in $GF(2^n)$ is represented by a vector with n elements in $GF(2)$ as follows:

$$a = (a_{n-1}, ..., a_0), \quad a_i \in GF(2).$$

There are two typical bases of $GF(2^n)$: polynomial basis and normal basis. The coordinate functions depend on the basis.

3.1 Polynomial basis

Fix an irreducible polynomial $p(x)$ over $GF(2)$ with algebraic degree n. The elements of $\{x^{n-1}, \ldots, x, 1\}$ become linearly independent over $GF(2)$, and this set is called the polynomial basis. An element $a = (a_{n-1}, \ldots, a_0)$ in $GF(2^n)$ corresponds to the polynomial $a_{n-1}x^{n-1} + \cdots + a_1x + a_0$.

The addition in $GF(2^n)$ corresponds to addition in n-dimensional vector space over $GF(2)$, i.e. it is done by bitwise exclusive-or. For multiplication in $GF(2^n)$, let $A = (a_{n-1}, ..., a_0)$ and $B = (b_{n-1}, ..., b_0)$ be elements in $GF(2^n)$, A is then represented as $a_{n-1}x^{n-1} + \cdots + a_0$, and B as $b_{n-1}x^{n-1} + \cdots + b_0$. The product of A and B is calculated by $AB \bmod p(x)$. There is an algorithm for computing this product which requires n shift operations and n additions(exclusive-or operations).

The k-th power function in $GF(2^n)$ is calculated by $A^k \bmod p(x)$. There is an algorithm for computing the k-th power which requires $2\log_2(k)$ multiplications in $GF(2^n)$. For a fast implementation with less memory, it is important to use an irreducible polynomial with small number of terms such as an irreducible trinomial (see Appendix B).

3.2 Normal basis

For each extension degree n, it is known that there is an element s in $GF(2^n)$ such that the elements of $\{s^{2^{n-1}}, \ldots, s^{2^2}, s^2, s\}$ are linearly independent. This element s is called a normal generator, and the above set is called the normal basis. An element in $GF(2^n)$ $a = (a_{n-1}, \ldots, a_0)$ corresponds to the polynomial $a_{n-1}s^{2^{n-1}} + \cdots + a_1s^2 + a_0s$.

The addition in $GF(2^n)$ is done in the same way as the polynomial basis case. Normal basis allows for a very fast squaring: it can be done by one shift operation, but multiplication is more complex than in polynomial basis. The Massey-Omura algorithm[10] and some improved algorithms have been proposed for multiplication[20], but the normal basis representation seems more appropriate for hardware, as some references reported[3, 4, 5, 15, 25].

2^k-th power operation by using normal basis requires only k cyclic shift operations, and it can be computed very fast.

Example 1 *Let* $n = 3$, $p(x) = x^3 + x + 1$, $A = (a_2, a_1, a_0)$ *and* $B = (b_2, b_1, b_0)$. *Then* $A^2 = (a_1, a_0, a_2)$ *and* $A^4 = (a_0, a_2, a_1)$. *If* $AB = (c_2, c_1, c_0)$ *denotes the product of* A *and* B, *each output bit of* AB *is obtained as follows.*

$$\begin{aligned} c_2 &= (a_2 + a_0)(b_2 + b_0) + (a_0 + a_1)(b_0 + b_1) + a_2 b_2 \\ c_1 &= (a_1 + a_2)(b_1 + b_2) + (a_2 + a_0)(b_2 + b_0) + a_1 b_1 \\ c_0 &= (a_0 + a_1)(b_0 + b_1) + (a_1 + a_2)(b_1 + b_2) + a_0 b_0 \end{aligned}$$

3.3 Table lookup

Generally speaking, table lookup achieves rather high speed. The problem is that the size of the table grows as the extension degree n increases. The upper bound of the table size and the efficiency of table lookup depend on the computer environment such as the sizes of memory and cache.

For example, we have three ways of realizing table lookups for computing $F(x)$ from x.

(1) lookup the table of the map $x \mapsto F(x)$ directly

(2) use the table of the inversion map[11]

(3) use the exp-log lookup tables[25]*

When aiming at the highest speed by the table lookup method, method (1) is suitable for a fixed function. On the other hand, tables (2) or (3) are of wider use, that is, they are useful for various products, power functions, inversion, and so on. In this paper, we use method (3).

3.4 Use of maps of coordinate functions

A function F in $GF(2^n)$ is also represented by the list of n coordinate functions $(f_{n-1}, \ldots, f_0)$, where f_i is a polynomial with n input variables. In advance we construct the polynomial of each coordinate function, where some improvement in computation speed is possible by finding the expression requiring the smaller number of operations†. There is a problem that the construction and optimization of the polynomials become more difficult as the extension degree n or the algebraic degree of f_i increases. However, using maps of coordinate functions and table lookup together is effective for a large extension degree n, as we show later.

3.5 Successive extension

It is known that a field can be considered as a vector space of one of its subfields. In this section we discuss the computation of operations in $GF(2^n)$ using successive extension. All methods explained previously are concerned with the single extension field of $GF(2)$. Generally speaking, every method based on a single extension field requires a larger number of calculations as the extension degree n increases, however, we can reduce the number of calculations by using successive extension.

*In [25], they call them `log`-table and `alog`-table.

†Actually the DES implementation in software using the similar method ("Bit-slice DES") gains a significant speedup[2].

When n can be factorized to some integers n_1 and n_2, an element of $GF(2^n)$ can be represented as a polynomial $\alpha_{n_1-1}x^{n_1-1} + \cdots + \alpha_1 x + \alpha_0$, where α_i are elements of $GF(2^{n_2})$. Here $GF(2^n)$ is the extension field of $GF(2^{n_2})$ with extension degree n_1.

When n_1 and n_2 are relatively prime, an irreducible polynomial with degree n_1 over $GF(2)$ is also irreducible over $GF(2^{n_2})$. In this case the implementation of functions in $GF(2^n)$ is rather easy. De Win et al. succeeded in slightly improving the operations in $GF(2^n)$ for $(n_1, n_2) = 1$ by using exp-log operation tables[25]. When $(n_1, n_2) \neq 1$, Aoki et al.[1] improved the operations in $GF(2^{32})$ which is equivalent to $GF((2^{16})^2)$. They constructed an irreducible polynomial with degree 2 and a normal basis over $GF(2^{16})$ for faster multiplications in $GF(2^{32})$. In general, it is not easy to find an irreducible polynomial with degree n_1 over $GF(2^{n_2})$ when n_1 and n_2 are not relatively prime. In this paper, we will consider only the case of $(n_1, n_2) = 1$.

Example 2 *We show an example of the polynomial representation of the element in* $GF(2^6) = GF((2^3)^2)$. *Since* 2 *and* 3 *are relatively prime, an irreducible polynomial* x^2+x+1 *over* $GF(2)$ *is also irreducible over* $GF(2^3)$. *So we can represent an element in* $GF(2^6)$ *as follows, considering* $GF(2^6)$ *as the successive extension field* $GF((2^3)^2)$;

$$(a_5y^2 + a_4y + a_3)x + (a_2y^2 + a_1y + a_0), \quad a_i \in GF(2).$$

Example 3 *If* $(p, q) \neq 1$, *an irreducible polynomial with degree* p *over* $GF(2)$ *is not irreducible over* $GF(2^q)$ *in general. For example, an irreducible polynomial* $x^3 + x + 1$ *over* $GF(2)$ *can be factorized over* $GF(2^6) = GF(2)[y]/Id(y^6 + y + 1)$ *into two polynomials as follows.*

$$(x + y^3 + y^2 + y)(x^2 + (y^3 + y^2 + y)x + y^4 + y^2 + y)$$

4 Performance of cubic function in $GF(2^n)$

In this section, we examine the computation times of the cubic functions x^3 in $GF(2^n)$ for $3 \leq n \leq 65$ by various methods. If n is odd, the cubic function x^3 in $GF(2^n)$ has been proved to be an almost bent and almost perfect nonlinear bijective function, namely, its maximal differential/linear probability is 2^{1-n} (see Table 1). Strictly speaking, the performances of other power functions x^k in $GF(2^n)$ with $k \neq 3$ are not the same as that of x^3. However, those of x^{2^t+1} in $GF(2^n)$, in particular, which are also almost bent and almost perfect nonlinear when $\frac{n}{gcd(t,n)}$ are odd, can achieve almost the same performance as x^3 by tuning the algorithm because of the same Hamming weight of their extension degree.

We executed the tests on a Sun Ultra 1 (UltraSPARC 170MHz) with 448MB memory and 512KB second cache. We used gcc compiler version 2.6.3.

4.1 Single extension

Figures 1 and 2 show the execution times of function x^3 in $GF(2^n)$ over $GF(2)$ by using single extension field adjoining the single root of $p(x)$ with respect to a polynomial basis and a normal basis, respectively. If n is a prime number, $GF(2^n)$ must be dealt with as the single extension field of $GF(2)$.

4.2 Table lookup

Some people assume that the difference in execution times of different sizes of lookup tables is negligible, however, such assumption is not true according to our experiments. Figure 3 shows the execution times for calculation of x^3 in $GF(2^n)$ by using exp-log lookup tables.

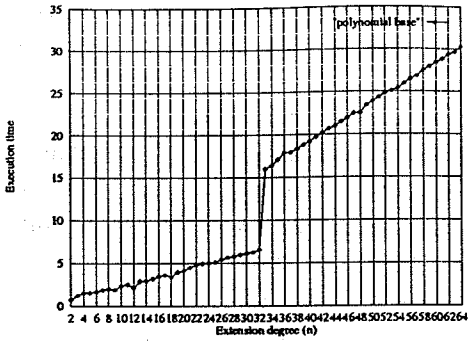


Figure 1: Execution time by single extension (polynomial basis) (μs.)

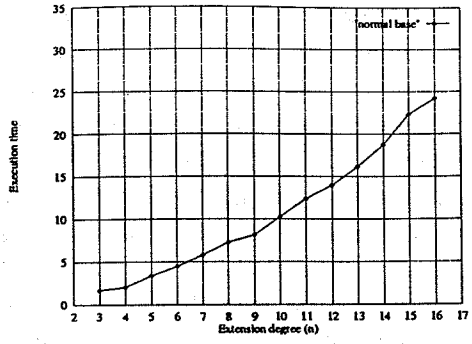


Figure 2: Execution time by single extension (normal basis) (μs.)

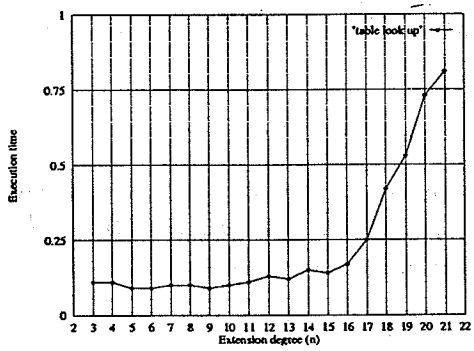


Figure 3: Execution time by table lookup (μs.)

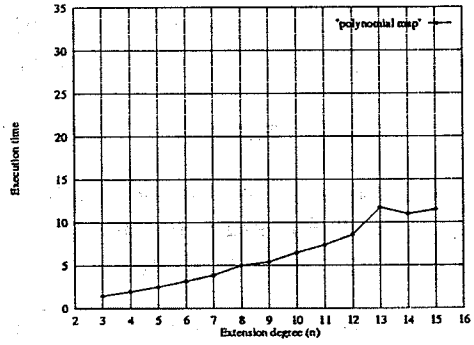


Figure 4: Execution time by using coordinate functions (μs.)

We find a sharp increase in the execution time at the "threshold" value which is about $n = 16$ in Figure 3. This phenomenon may come from cache mis-hit. This "threshold" value depends on the computer architecture. The lookup table of a power function in $GF(2^n)$ requires $2 \cdot w \cdot 2^n$ bits, where w is the word size in bits of the processor. Since we used a computer with 512KB second cache, the size of the lookup table exceeds the cache size if n is bigger than 16. When we also conducted the same experiment on another computer with 256KB second cache, there was a steep rise at about $n = 15$.

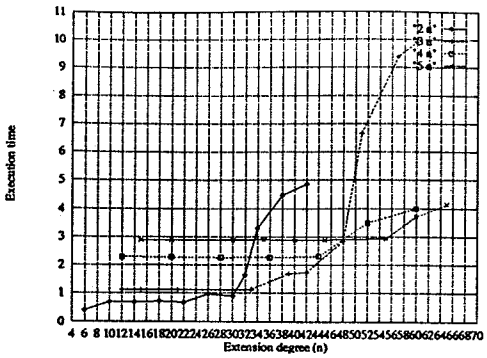
4.3 Use of maps of coordinate functions

Figure 4 shows the execution times of x^3 in $GF(2^n)$ by using coordinate functions[‡]. This method has an advantage in that coordinate functions can be calculated in parallel. However, parallel computation of coordinate polynomials are hard to realize with software implementation, and moreover, only a part of the resource can be used (e.g. one bit operation in 32-bit CPU). Therefore, direct use of this method would be inefficient.

4.4 Successive extension

Figure 5 shows the execution times by using successive extension. Assume that n can be factorized into n_1 and n_2, where $n_1 < n_2$. In this paper, we consider only the cases of extension of degree $n_1 \in \{2, 3, 4, 5\}$. Calculations of coefficients in $GF(2^{n_2})$ are done by table lookup, and

[‡]For construction of coordinate functions, we used the computer algebra system Risa/Asir[16].



$2n$: extension of degree 2 over $GF(2^{n_2})$ $(3 \leq n_2 \leq 22)$
$3n$: extension of degree 3 over $GF(2^{n_2})$ $(3 \leq n_2 \leq 20)$
$4n$: extension of degree 4 over $GF(2^{n_2})$ $(3 \leq n_2 \leq 15)$
$5n$: extension of degree 5 over $GF(2^{n_2})$ $(3 \leq n_2 \leq 13)$

Figure 5: Execution time by using successive extension (μs.)

extension of degree n_1 is calculated by using coordinate functions. Since n_1 is rather small, the coordinate function is simple enough to optimize. For each extension degree n_1, the required times of operations in the coefficient field $GF(2^{n_2})$ are as follows.

n_1	# of cubings	# of squarings	# of multiplications
2	1	0	3
3	5	2	3
4	7	4	8
5	10	5	16

Only for the case of $n_1 = 2$ and $n_2 = 16$, are they $\{2, 2, 4\}$. Note that we do not know whether the numbers of the operations above are minimal or not.

4.5 Our best execution times of x^3 in $GF(2^n)$ for $3 \leq n \leq 65$

Of the various algorithms we used for computing x^3 in $GF(2^n)$ for $3 \leq n \leq 65$, we plot the best execution times in Figure 6. The list of execution times and the algorithms we adopted appears in Appendix A. Generally speaking, x^3 in $GF(2^n)$ can be computed fast **1.** when n is small (< 20) by table lookup, and **2.** when n is factorized and the technique for successive extension is available.

5 Construction of block ciphers with high security and performance

We considered speed and security separately in the previous chapters, however, in this section we discuss which combination of x^k in $GF(2^n)$ yields 64-bit block ciphers with high security and high speed using the execution times of cubic functions derived in Section 4. Strictly speaking, the performances of power functions in $GF(2^n)$ with $k \neq 3$ are not the same as that of x^3. However, those of x^{2^t+1} in $GF(2^n)$, in particular, which are also almost bent and almost perfect nonlinear when $\frac{n}{gcd(t,n)}$ are odd, can achieve almost the same performance as x^3 by tuning the algorithm because of the same Hamming weight of their extension degree. Therefore, we discuss the construction of block ciphers using Matsui's structure with high performance and security using the execution times of x^k in $GF(2^n)$ for $3 \leq n \leq 65$ only for $k = 3$.

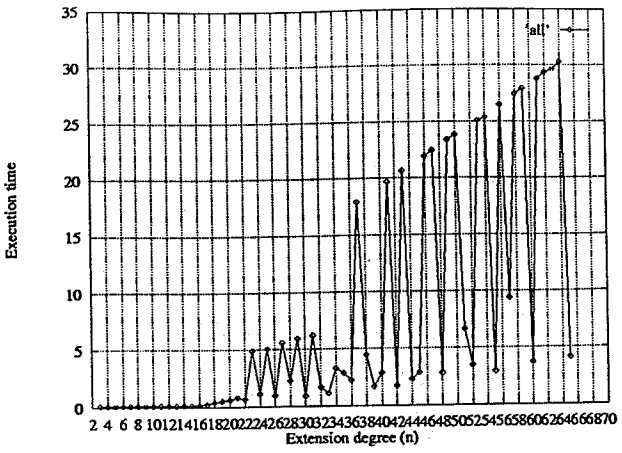


Figure 6: Execution time of x^3 over $GF(2^n)$ (μs.)

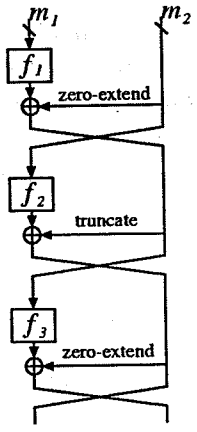


Figure 7: Matsui's recursive structure

Figure 7 shows the most inner part of the recursive structure described in [13]. When the input string is divided into m_1-bit and m_2-bit ($m_1 \geq m_2$), this division is denoted by $[[m_1, m_2]]$. For the structure shown in Figure 7, the following theorem is proved[13]. We find the block cipher with high speed and maximal differential/linear probability 2^{-62} using this theorem.

Theorem 1 [13] *In Figure 7, assume that f_1, f_2, and f_3 are bijective and the maximal differential/linear probabilities are smaller than p_1, p_2, and p_3, respectively. If the entire function F shown in the figure has more than three rounds, then the maximal differential/linear probability of F is smaller than*

$$\max\{p_1p_2,\ p_2p_3,\ 2^{m_1-m_2}p_1p_3\}.$$

5.1 Division of 16-bit input/output (i.e. when $m_1 + m_2 = 16$)

First, we consider the case where 16-bit input/output is divided into m_1-bit and m_2-bit at the most inner part of Matsui's recursive structure. We can expand it to the entire block cipher with 64-bit input/output recursively like MISTY[13].

From the view point of speed, every combination of m_1 and m_2 *s.t.* $m_1 + m_2 = 16$ is similar on an usual PC or workstation with $256 \sim 512$KB cache. This is because the execution times

of power functions over $GF(2^n)$ for $n \leq 16$ by using table lookup do not differ greatly (see Figure 3).

In terms of security, however, there is some difference. Consider the division of 16-bits into two even numbers. In this case, according to Theorem 1, the maximal differential/linear probability is smaller than

$$\max\{2^{2-m_1}2^{2-m_2},\ 2^{2-m_2}2^{2-m_1},\ 2^{m_1-m_2}2^{2-m_1}2^{2-m_1}\} = 2^{4-16} = 2^{-12}.$$

Note that if $gcd(k, 2^n - 1) \neq 1$, then x^k in $GF(2^n)$ is not bijective. For example, the cubic function is not bijective when n is even. When 16-bits is divided into two odd numbers, the maximal differential/linear probability is smaller than

$$\max\{2^{1-m_1}2^{1-m_2},\ 2^{1-m_2}2^{1-m_1},\ 2^{m_1-m_2}2^{1-m_1}2^{1-m_1}\} = 2^{2-16} = 2^{-14}.$$

From these discussions, it is concluded that the division of 16-bits into two odd numbers is better. Therefore, the candidates of the division are [[15,1]], [[13,3]], [[11,5]] and [[9,7]]. Considering the size of tables, the division [[9,7]] is the best.

5.2 Division of 32-bit input/output (i.e. when $m_1 + m_2 = 32$)

If we expand the 16-bit function discussed above (where the division is [[9,7,9,7]]) to a 32-bit function like the function FO of MISTY[13], the maximal differential/linear probability of this function is smaller than $(2^{-14})^2 = 2^{-28}$ according to Theorem 1. So is there any division of 32-bit that attains higher security?

If we divide 32-bit input/output into two even numbers, then the maximal differential/linear probability is smaller than

$$\max\{2^{2-m_1}2^{2-m_2},\ 2^{2-m_2}2^{2-m_1},\ 2^{m_1-m_2}2^{2-m_1}2^{2-m_1}\} = 2^{4-32} = 2^{-28}.$$

which is the same as the case above. If we divide 32-bit input/output into two odd numbers, then the maximal differential/linear probability becomes smaller than 2^{-30}.

Let m_1 and m_2 be odd numbers *s.t.* $m_1 + m_2 = 32$. Since it is impossible to divide an odd number into two odd numbers, some power functions in $GF(2^{m_1})$ and $GF(2^{m_2})$ have to be used. We have to use the successive extension method for $m_i \geq 17$, since the table lookup would be inefficient. Considering that m_i should be small enough for table lookup or be factorized into relatively prime integers, the candidates of the division are [[27,5]], [[25,7]], and [[21,11]].

If we calculate the execution time using the timing data in Appendix A, the division [[21,11]] attains the fastest performance;

$$0.11 + 2 \times 0.81 = 1.73\ (\mu s). \tag{1}$$

On the other hand, the execution time of the division [[9,7,9,7]] is

$$3 \times (0.10 + 2 \times 0.09) = 0.84\ (\mu s). \tag{2}$$

This means that the division [[21,11]] improves the security in terms of differential/linear probability (2^{-30} v.s. 2^{-28}), but reduces the performance by half.

5.3 Division of 64-bit input/output (i.e. when $m_1 + m_2 = 64$)

Similarly, consider the division of 64-bit input/output. Using the same argument in the previous subsections, the division into two even numbers does not achieve higher security than that based on the division of 32-bits. Therefore, we consider the division of 64-bits into two odd numbers. In this case, the maximal differential/linear probability is smaller than 2^{-62}, which is the smallest value of the upper bound for the maximal differential/linear probabilities of block ciphers with 64-bit input/output.

For higher performance, we should use table lookup, and avoid using the division where the construction of the successive extension field is complicated. Then the remaining candidates are [[55,9]] and [[51,13]]. Since these divisions have the same level of security, we calculate the execution times of the block ciphers with 18 rounds, for example, constructed based on these divisions. Note that we consider the exclusive-or operations for each round as negligible and don't consider parallel computation for simplicity.

$$\begin{aligned} [[55,9]] &: 2.93 \times 9 + 0.09 \times 9 = 27.18\ (\mu s) \\ [[51,13]] &: 6.67 \times 9 + 0.12 \times 9 = 61.11\ (\mu s) \end{aligned}$$

The block cipher based on the division [[55,9]] is about twice as fast as that based on [[51,13]]. On the other hand, the speed of the block cipher with 18 rounds constructed based on the division [[9,7]] discussed in Section 5.1 is estimated as below (using the data "0.84" in equation(2)).

$$0.84 \times 18 = 15.12\ (\mu s)$$

This also means that the division [[55,9]] improves the security in terms of differential/linear probability (2^{-62} v.s. 2^{-56}), but deteriorates the performance.

5.4 Resistance to higher order differential attack

The higher order differential attack[6] is an algebraic attack based on the higher order differentials[9] of the coordinate functions. Consider a 64-bit block cipher with the structure shown in Figure 7 with $2r$ rounds. Note that it has no recursive structure like MISTY. Because the coordinate polynomials of the function x^{2^t+1} in $GF(2^n)$ over $GF(2)$ is quadratic, this cipher has 64 coordinate functions with degree 2^r. Assuming the attack of 2-round elimination, to resist to the higher order differential attack, the following inequality must hold: $2^{r-2} > 64$, that is $r > 8$. Therefore, this cipher needs at least 17 rounds. Details of the improved higher order differential attack are written in [22].

We can construct a Feistel cipher like MISTY1[13] based on the division [[21,11]] discussed in Section 5.2. Although this block cipher has maximal differential/linear probability 2^{-60}, which is larger than 2^{-62}, it needs only at least 9 rounds to resist to the higher order differential attack, assuming the attack of 2-round elimination. The performance of this cipher is estimated as $1.73 \times 9 = 15.57\ (\mu s)$, where the data "1.73" was derived in equation(1).

6 Conclusion

We presented a block cipher which has the best performance of all 64-bit block ciphers with the smallest proven differential/linear probability 2^{-62} using the structure proposed by Matsui. It was the unbalanced Feistel cipher shown in Figure 7 where f_1 and f_3 are x^{2^t+1} in $GF(2^{55})$ and f_2 is x^{2^t+1} in $GF(2^9)$ for some positive integers t. The number of rounds should be more than 17 to resist to the higher order differential attack.

The performance depends on the computing environment and the programmer's skill of implementing. For the former, the assumed computing environment is an usual PC or workstation with 256 $\sim$ 512KB cache. For the latter, in general, power functions in $GF(2^n)$ can be implemented fast **1.** when n is small (< 20) by table lookup or **2.** when n is factorized into relatively prime integers so that the technique for successive extension be available. Therefore, the presented block cipher is expected to be the fastest of all 64-bit block ciphers with the smallest proven differential/linear probability 2^{-62} using the structure proposed by Matsui.

References

[1] K. Aoki, K. Ohta, "Fast Arithmetic Operations over F_{2^n} for Software Implementation," in Proc. of SAC'97, 1997.

[2] E. Biham, "A Fast New DES Implementation in Software," in Proc. of the Fourth Fast Software Encryption Workshop, pp.241–253, 1997.

[3] T. Itoh, S. Tsujii, "Algorithms over Finite Fields [I]," in Proc. of the 1987 Workshop in Cryptography and Information Security, WCISS87-3 pp.51–60, 1987.

[4] T. Itoh, S. Tsujii, "A Fast Algorithm for Computing Multiplicative Inverses in $GF(2^m)$ Using Normal Bases," Information and Computation 78, pp.171–177, Academic Press, Inc., 1988.

[5] T. Itoh, S. Tsujii, "Structure of Parallel Multipliers for a Class of Field $GF(2^m)$," Information and Computation 83, pp.21–40, Academic Press, Inc., 1989.

[6] T. Jacobsen, L. R. Knudsen, "The Interpolation Attack on Block Ciphers," in Proc. of the Fourth Fast Software Encryption Workshop, pp.28–40, 1997.

[7] L. R. Knudsen, "Block Ciphers – Analysis, Design and Applications," Ph.D.Thesis, Computer Science department, Aarhus University, 1994.

[8] X. Lai, J. L. Massey, "Markov Ciphers and Differential Cryptanalysis," Advances in Cryptology — EUROCRYPT'91, Lecture Notes in Computer Science 547, pp.17–38, Springer Verlag, 1991.

[9] X. Lai, "Higher Order Derivatives and Differential Cryptanalysis," in Proc. of "Symposium on Communication, Coding and Cryptography," in honor of James L. Massey on the occasion of his 60'th birthday, 1994.

[10] J. L. Massey, J. K. Omura, "Computational Method and Apparatus for Finite Field Arithmetic," US Patent No.4,587,627, 1981.

[11] T. Matsumoto, Y. Takashima, J.W. Machar, H. Imai, "Inverter-based Multiplier for Ciphers and Codes," in Proc. of the 1988 conference of IEICE, SA-7-5, A-1, pp.173-174, 1988.

[12] M. Matsui, "New Structure of Block Ciphers with Provable Security against Differential and Linear Cryptanalysis," Fast Software Encryption, Lecture Notes in Computer Science 1039, pp.205–218, Springer Verlag, 1996.

[13] M. Matsui, "New Block Encryption Algorithm MISTY," in Proc. of the Fast Software Encryption Workshop, pp.53–67, 1997.

[14] A. J. Menezes, P. C. Oorschot, S. A. Vanstone, "HANDBOOK of Applied Cryptography," CRC Reference Series in Discrete Mathematics, CRC press, 1995.

[15] M. Morii, M Kasahara, "Computation algorithms over Galois fields," in Proc. of the 1987 Workshop in Cryptography and Information Security, WCISS87-4, pp.61–70, 1987.

[16] M. Noro, T. Takeshima, "Risa/Asir – a computer algebra system," in Proc. of ISSAC'92, pp.387–396, ACM Press, 1992. (anonymous ftp from `endeavor.flab.fujitsu.co.jp`, directory `/pub/isis/asir`)

[17] K. Nyberg, "Differentially Uniform Mappings for Cryptography," Advances in Cryptology — EUROCRYPT'90, Lecture Notes in Computer Science 763, pp.55–64, Springer Verlag, 1994.

[18] K. Nyberg, L.R.Knudsen, "Provable Security Against a Differential Attack," Journal of Cryptology 1995-8, pp.27–37, Springer Verlag, 1995.

[19] J. Pieprzyk, "How to Construct Pseudorandom Permutations from Single Pseudorandom Functions," Advances in Cryptology — EUROCRYPT'90, Lecture Notes in Computer Science 437, pp.140–150, Springer Verlag, 1990.

[20] A. Pincin, "A New Algorithm for Multiplication in Finite Fields," IEEE Transactions on Computers, vol.38, No.7, pp.1045–1049, 1989.

[21] A. Pincin, "Bases for Finite Fields and A Canonical Decomposition for a Normal Basis Generator," Communications in Algebra 17(6), pp.1337–1352, 1989.

[22] T. Shimoyama, S. Moriai, T. Kaneko, "Improving the Higher Order Differential Attack and Cryptanalysis of $\mathcal{KN}$ Cipher," in Proc. of ISW'97, 1997.

[23] V. Shoup, "On The Deterministic Complexity of Factoring Polynomials over Finite Fields," Information Processing Letters 33 North-Holland pp.261–267, 1990.

[24] V. Shoup. "New Algorithms for Finding Irreducible Polynomials over Finite Fields," Mathematics of computation, Vol.54, No.189, pp.435–447, 1990.

[25] E. De Win, A. Bosselaers S. Vandenberghe, P. D. Gersen, J. Vandewalle, "A Fast Software Implementation for Arithmetic Operations in $GF(2^n)$," Asiacrypt'96. Lecture Note in Computer Science 1163, pp.65–76, Springer Verlag, 1996.

Appendices

A Execution times of x^3 in $GF(2^n)$

The following table shows the execution times of x^3 over $GF(2^n)$ and the algorithms we adopted. The tests were executed on a Sun Ultra 1 (UltraSPARC 170MHz) with 448MB memory and 512KB second cache. We used gcc compiler version 2.6.3. The meanings of the abbreviations in the column of "algorithm" are as follows:

table lookup : table lookup by using the exp-log table
$a \times b$: successive extension with degree a over $GF(2^b)$
poly. base : single extension over $GF(2)$ using polynomial basis

n	time (μs)	algorithm	34	3.30	2 ×17
3	0.10	table lookup	35	2.90	5 × 7
4	0.10	table lookup	36	2.27	4 × 9
5	0.09	table lookup	37	17.89	poly. base
6	0.09	table lookup	38	4.47	2 × 19
7	0.10	table lookup	39	1.69	3 × 13
8	0.10	table lookup	40	2.87	4 × 8
9	0.09	table lookup	41	19.66	poly. base
10	0.10	table lookup	42	1.74	3 × 14
11	0.11	table lookup	43	20.64	poly. base
12	0.13	table lookup	44	2.30	4 × 11
13	0.12	table lookup	45	2.89	5 × 9
14	0.15	table lookup	46	21.89	poly. base
15	0.14	table lookup	47	22.44	poly. base
16	0.17	table lookup	48	2.84	3 × 16
17	0.25	table lookup	49	23.37	poly. base
18	0.42	table lookup	50	23.83	poly. base
19	0.53	table lookup	51	6.67	3 × 17
20	0.61	2 ×10	52	3.49	4 × 13
21	0.81	table lookup	53	25.06	poly. base
22	0.67	2 ×11	54	25.34	poly. base
23	4.91	poly. base	55	2.93	5 × 11
24	1.12	3 ×12	56	26.48	poly. base
25	5.06	poly. base	57	9.39	3 × 19
26	0.97	2 ×13	58	27.45	poly. base
27	5.61	poly. base	59	27.91	poly. base
28	2.26	4 ×7	60	3.71	3 × 20
29	5.95	poly. base	61	28.78	poly. base
30	0.90	2 ×15	62	29.34	poly. base
31	6.21	poly. base	63	29.63	poly. base
32	1.64	2 ×16	64	30.29	poly. base
33	1.13	3 ×11	65	4.14	5 × 13

Table 2: Execution times of x^3 in $GF(2^n)$

B Irreducible polynomials over $GF(2^n)$

The following is a list of irreducible polynomials over $GF(2^n)$ for $2 \leq n \leq 69$, primitive elements, and normal generators used in our experiments. In the second column we give irreducible trinomials $x^n + x^{p_1} + 1$ represented by p_1 if they exist, otherwise $x^n + x^{p_1} + x^{p_2} + x^{p_3} + 1$ represented by p_1, p_2, p_3.

n	irreducible polynomial $p_1(, p_2, p_3)$	primitive element	normal generator	n	irreducible polynomial $p_1(, p_2, p_3)$	primitive element	normal generator
2	1	x	x	35	2	x	x^{33}
3	1	x	$x+1$	36	9	$x+1$	$x^{27}+x^3+x$
4	1	x	x^3	37	6,4,1	x	x^{31}
5	2	x	x^3	38	6,5,1	x	x^{33}
6	1	x	x^5	39	4	x	x^{35}
7	1	x	x^3+1	40	5,4,3	x	x^{35}
8	4,3,1	x	x^5	41	3	x	$x+1$
9	1	x^2+x+1	$x+1$	42	7	x^2+x+1	$x^{35}+x^3+x$
10	3	x	x^7	43	6,4,3	x	x^{37}
11	2	x	x^9	44	5	x^2+x+1	x^{39}
12	3	$x+1$	x^9+x	45	4,3,1	x	$x+1$
13	4,3,1	x	x^9	46	1	x^2+x+1	x^{45}
14	5	x^2+x+1	x^9	47	5	x	$x+1$
15	1	x	x^7+1	48	5,3,2	$x+1$	x^{43}
16	5,3,1	$x+1$	x^{11}	49	9	x	$x+1$
17	3	x	$x+1$	50	4,3,2	x	x^{47}
18	3	x^3+x	$x^{15}+x$	51	6,3,1	x	x^{45}
19	5,2,1	x	x^{17}	52	3	x	x^{49}
20	3	x	x^{17}	53	6,2,1	x	x^{47}
21	2	x	x^{19}	54	9	x^3+x	$x^{45}+x^3+x$
22	1	x	x^{21}	55	7	$x+1$	$x+1$
23	5	x	$x+1$	56	7,4,2	x	x^{49}
24	4,3,1	x	x^{21}	57	4	x^2+x+1	x^{53}
25	3	x	$x+1$	58	19	x	x^{39}
26	4,3,1	$x+1$	x^{23}	59	7,4,2	x	x^{55}
27	5,2,1	x	$x+1$	60	1	x	x^{59}
28	1	x^2+x+1	x^{27}	61	5,2,1	x	x^{59}
29	2	x	x^{27}	62	29	$x+1$	x^{33}
30	1	x^4+x+1	x^{29}	63	1	x	$x^{31}+1$
31	3	x	$x^{15}+1$	64	4,3,1	x	x^{61}
32	7,3,2	$x+1$	x^{25}	65	18	x	x^{47}
33	10	$x+1$	x^{23}	66	3	$x+1$	$x^{63}+x^5$
34	7	x^3+x+1	x^{27}	67	5,2,1	x	x^{65}
				68	9	x	x^{59}
				69	6,5,2	x	x^{63}

Fast Arithmetic Operations over $\mathbf{F}_{2^n}$ for Software Implementation

Kazumaro AOKI Kazuo OHTA
maro@isl.ntt.co.jp ohta@isl.ntt.co.jp
NTT Laboratories[1]

Abstract

This paper discusses the representation of a finite field with characteristic 2, and proposes arithmetic operations over the field using a successive extension of the field. We develop a fast inversion algorithm, and compare its speed with the previous algorithms, and consider their applications. We confirm that our algorithm is effective and suitable for software implementation.

1 Introduction

Recently, many secret key block cipher algorithm construction methodologies have been presented because the two strong attacks of differential cryptanalysis [BS93] and linear cryptanalysis [M94] have broken various cipher algorithms designed by old construction methodologies. An important subject for the construction of a strong F-function is how to construct an S-box which is nonlinear primitive of a cipher. For example, Nyberg and Knudsen proves that a Feistel cipher, including strong F-functions, is secure against differential cryptanalysis [NK95] and Nyberg proves the cipher is secure against linear cryptanalysis [N95]. At present, we know that some types of a monomial function, for example x^3, and an inversion function, x^{-1} over a finite field are most secure against differential and linear cryptanalysis [NK95, CV95]. So we can construct a block cipher algorithm using the monomial functions as an element which permits security against differential and linear attack[2].

It is well known that the arithmetic operations over finite fields are very important because they are used by cryptology and coding theory and so on. The arithmetic operations over finite fields with characteristic 2 must be discussed because their data structure is sufficiently clear to simplify implementation. Though hardware implementation of fast arithmetic operations over finite fields has been well studied, software implementation did not so sufficiently. Since recent computer power has increased dramatically, software implementations become also important.

This paper discusses representations of finite fields with characteristic 2 at first, and presents fast algorithms for arithmetic operations over the fields and their applications. Our inversion algorithm requires about 1.5 times of complexity of multiplication, while the previous algorithms requires greater than 2 times of complexity of multiplication.

Note that we use the notation $\mathbf{F}_q$ as a finite field which contains q elements.

2 Previous Algorithms and Our Results

Research on speeding up arithmetic operations over finite fields, especially those with characteristic 2, has a long history, because they have many practical applications. An important problem is how to represent the element of a finite field. The previous implementations use a standard basis, or a normal basis, or subfields or combinations of these representations.

Let $f(X)$ be an irreducible polynomial with degree n, then we have $\mathbf{F}_2[X]/(f(X)) \cong \mathbf{F}_{2^n}$. So, we can regard an element of $\mathbf{F}_{2^n}$ as a polynomial over $\mathbf{F}_2$ with degree less than n. Then, a multiplication can be calculated by simple polynomial multiplication and modulo reduction, and an inversion can be calculated by the extended Euclidean algorithm.

Massey and Omura used a normal basis for a multiplication over $\mathbf{F}_{2^n}$ [MO81]. If we use a normal basis over $\mathbf{F}_2$, a squaring can be calculated by a cyclic shift, and a multiplier can be permitted by parallel

[1] 1-1 Hikarinooka, Yokosuka-shi, Kanagawa-ken, 239 Japan

[2] However, since there are many attacks without differential or linear attack, a block cipher construction requires other security elements.

construction. However, an inversion is not easy to calculate. Itoh and Tsujii proposed an inversion algorithm over $\mathbf{F}_{q^n}$ [IT87] using a normal basis. Their algorithm is based on the formula of $x^{-1} = x^{q^n-2}$.

Almost of earlier algorithms are considered for hardware implementation except the works described below. So, we consider algorithms for software implementation. If we use a standard basis or a normal basis over the prime field $\mathbf{F}_2$, we have slow software implementations, because the construction based on $\mathbf{F}_2$ require many bit-operations instead of word-operations which are fast for software execution. On the other hand, an algorithm using a subfield yields a fast software implementation if we specify an appropriate subfield, for example $\mathbf{F}_{2^8}$, because we can calculate all of operations over the subfield and memorize them in advance, and then we can reduce a field operation to subfield operations using word-operations.

Pincin proposed an algorithm for the finite field $\mathbf{F}_{p^n}$ using a normal basis over subfields and successive extension [P89]. The algorithm combines several operations which are basically the same operation into one operation for enhancing the speed. It is effective if n has many divisors.

Harper, Menezes, and Vanstone implemented arithmetic operations over $\mathbf{F}_{2^{104}}$ using a standard basis over subfield $\mathbf{F}_{2^8}$ [HMV93]. They also implemented arithmetic operations over $\mathbf{F}_{2^{105}}$ using an optimal normal basis. As a result, their subfield implementation is faster than a normal basis implementation. Recently, Win et al. implemented arithmetic operations using a standard basis over a subfield [WBV+96]. They optimized arithmetic operations using an irreducible trinomial with coefficients 1.

Paar introduced useful condition of a quadratic extension of a field with characteristic 2 for a effective multiplication algorithm [P96, Sect. 4.2]. However, this condition was not well studied. This paper discuss a successive quadratic extension of a field with characteristic 2 and arithmetic operations not only a multiplication but also a squaring and an inversion.

3 Finite Extension of $\mathbf{F}_2$

Earlier algorithms for arithmetic operations did not give a generic construction method for $\mathbf{F}_{2^n}$. So, an irreducible polynomial and its normal basis were found heuristically. Thus, special algorithm, for example [S90], is required for getting an irreducible polynomial. We will explain how to construct a quadratic irreducible polynomial and its normal basis.

3.1 Construction of a Quadratic Extension of a Field with Characteristic 2

Lemma 1

Assume that a quadratic polynomial of X,

$$X^2 + X + z \quad (z \in \mathbf{F}_{2^n}) \tag{1}$$

is irreducible over $\mathbf{F}_{2^n}$. Then, a quadratic polynomial of X,

$$X^2 + X + z\omega$$

is irreducible over $\mathbf{F}_{2^{2n}} = \mathbf{F}_{2^n}(\omega)$ where ω is a root of $X^2 + X + z = 0$.

Proof) The proof is by contradiction. Let $x + y\omega$ $(x, y \in \mathbf{F}_{2^n})$ be a root of $X^2 + X + z\omega = 0$. Using $\omega^2 + \omega + z = 0$, we have

$$(x + y\omega)^2 + (x + y\omega) + z\omega = (x^2 + y^2 z + x) + (y^2 + y + z)\omega = 0.$$

Since $\mathbf{F}_{2^{2n}}$ is a vector space over $\mathbf{F}_{2^n}$, and 1 and ω are linearly independent,

$$y^2 + y + z = 0$$

holds. The condition contradicts with Eq. (1). Then, $X^2 + X + z\omega$ is irreducible over $\mathbf{F}_{2^{2n}}$. ∎

3.2 Normal Basis of $\mathbf{F}_{2^n}$ for a Quadratic Extension

Based on the construction presented in Sect. 3.1, we assume

$$\mathbf{F}_{2^{2n}} = \mathbf{F}_{2^n}(\alpha).$$

That is; α is a root of

$$X^2 + X + a = 0 \quad (a \in \mathbf{F}_{2^n}). \tag{2}$$

Since the left side of Eq. (2) is irreducible over $\mathbf{F}_{2^n}$, we have $\alpha \notin \mathbf{F}_{2^n}$. Then, we have

$$(\alpha+1)^2 + (\alpha+1) + a = \alpha^2 + \alpha + a.$$

So, if α is a root of Eq. (2), $(\alpha+1)$ is also a root of Eq. (2).

Let both sides of Eq. (2) be raised to the 2^n-th power. Since $(\alpha^2+\alpha+a)^{2^n} = (\alpha^{2^n})^2 + \alpha^{2^n} + a = 0$, we have $\alpha^{2^n} = \alpha$ or $\alpha+1$. If $\alpha^{2^n} = \alpha$ holds, then $\alpha \in \mathbf{F}_{2^n}$ holds. That is a contradiction of the assumption. Thus, we have $\alpha^{2^n} = \alpha + 1$. That is; α is a normal basis generator, and

$$[\alpha\ \alpha^{2^n}] = [\alpha\ \alpha+1]$$

holds.

4 Arithmetic Operations in a Quadratic Extension Field with Characteristic 2 Using a Successive Extension

Pincin pointed out that a successive extension using a normal basis is effective [P89], and Paar pointed out that a quadratic extension using a standard basis is effective [P96, Sect. 4.2] for multiplication. So, we expect that other operations which are a squaring and an inversion are also effective. This section discusses successive quadratic extension and compares with a standard basis and a normal basis. We assume that the finite field $\mathbf{F}_{2^{2n}}$ is represented using α as the quadratic extension of $\mathbf{F}_{2^n}$ based on the previous section which means that α satisfies the equation $\alpha^2 + \alpha + a = 0$, and $\mathbf{F}_{2^n}$ contains variables x_i and y_i $(i = 1, 2, 3)$.

4.1 Standard Basis

In this section, we use the standard basis, $[1\ \alpha]$. So, any element of $\mathbf{F}_{2^{2n}}$ can be represented by the form of $x + y\alpha$ $(x, y \in \mathbf{F}_{2^n})$.

4.1.1 Addition

It's trivial to calculate by the following equation.

$$(x_1 + y_1\alpha) + (x_2 + y_2\alpha) = (x_1 + x_2) + (y_1 + y_2)\alpha$$

Based on the equation above, we develop the following algorithm.

Algorithm 1 (Addition (Standard Basis))

Input: (x_1, y_1), (x_2, y_2)
Output: (x_3, y_3)

Step 1: $x_3 := x_1 + x_2$
Step 2: $y_3 := y_1 + y_2$

So, we can calculate an $\mathbf{F}_{2^{2n}}$-addition using 2 $\mathbf{F}_{2^n}$-additions.

4.1.2 Multiplication

Refer to [P96, Sect. 4.2], we have

$$(x_1 + y_1\alpha) \times (x_2 + y_2\alpha) = (x_1x_2 + ay_1y_2) + ((x_1 + y_1)(x_2 + y_2) + x_1x_2)\alpha.$$

Based on the equation above, we develop the following algorithm.

Algorithm 2 (Multiplication (Standard Basis))

Input: (x_1, y_1), (x_2, y_2)
Output: (x_3, y_3)

Step 1: $t_1 := x_1 + y_1$
Step 2: $t_2 := x_2 + y_2$
Step 3: $t_1 := t_1 \times t_2$ $(= (x_1 + y_1)(x_2 + y_2))$
Step 4: $t_2 := x_1 \times x_2$
Step 5: $y_3 := t_1 + t_2$ $(= (x_1 + y_1)(x_2 + y_2) + x_1x_2)$
Step 6: $t_1 := y_1 \times y_2$
Step 7: $t_1 := a \times t_1$ $(= ay_1y_2)$
Step 8: $x_3 := t_1 + t_2$ $(= x_1x_2 + ay_1y_2)$

So, we can calculate an $\mathbf{F}_{2^{2n}}$-multiplication using 4 $\mathbf{F}_{2^n}$-additions and 4 $\mathbf{F}_{2^n}$-multiplications.

4.1.3 Squaring

Simplify the equation used by the multiplication algorithm, then we have

$$(x_1 + y_1\alpha)^2 = (x_1^2 + ay_1^2) + y_1^2\alpha.$$

Based on the equation above, we develop the following algorithm.

Algorithm 3 (Squaring (Standard Basis))

Input: (x_1, y_1)
Output: (x_3, y_3)

Step 1: $y_3 := y_1^2$
Step 2: $t_1 := a \times y_3$ $(= ay_1^2)$
Step 3: $t_2 := x_1^2$
Step 4: $x_3 := t_1 + t_2$ $(= x_1^2 + ay_1^2)$

So, we can calculate an $\mathbf{F}_{2^{2n}}$-squaring using an $\mathbf{F}_{2^n}$-addition, an $\mathbf{F}_{2^n}$-multiplication, and 2 $\mathbf{F}_{2^n}$-squarings.

4.1.4 Inversion

Using the property of norm, since Sect. 3.2 shows that the conjugate of α is $\alpha + 1$, we have

$$(x_1 + y_1\alpha)(x_1 + y_1(\alpha + 1)) = x_1(x_1 + y_1) + ay_1^2.$$

So the inverse is obtained by the following equation[3].

$$(x_1 + y_1\alpha)^{-1} = (x_1(x_1 + y_1) + ay_1^2)^{-1}((x_1 + y_1) + y_1\alpha)$$

Based on the equation above, we develop the following algorithm.

Algorithm 4 (Inversion (Standard Basis))

[3]This equation can also be obtained by the extended Euclidean algorithm.

Input: (x_1, y_1)
Output: (x_3, y_3)

Step 1: $t_1 := x_1 + y_1$
Step 2: $t_2 := t_1 \times x_1$ $(= x_1(x_1 + y_1))$
Step 3: $t_3 := y_1^2$
Step 4: $t_3 := a \times t_3$ $(= ay_1^2)$
Step 5: $t_2 := t_2 + t_3$ $(= x_1(x_1 + y_1) + ay_1^2)$
Step 6: $t_2 := t_2^{-1}$ $(= (x_1(x_1 + y_1) + ay_1^2)^{-1})$
Step 7: $x_3 := t_1 \times t_2$ $(= (x_1 + y_1)(x_1(x_1 + y_1) + ay_1^2)^{-1})$
Step 8: $y_3 := y_1 \times t_2$ $(= y_1(x_1(x_1 + y_1) + ay_1^2)^{-1})$

So, we can calculate an $\mathbf{F}_{2^{2n}}$-inversion using 2 $\mathbf{F}_{2^n}$-additions, 4 $\mathbf{F}_{2^n}$-multiplications, an $\mathbf{F}_{2^n}$-squaring, and an $\mathbf{F}_{2^n}$-inversion.

4.2 Normal Basis

In this section, we use the normal basis, $[\alpha \ \alpha + 1]$. So, any element of $\mathbf{F}_{2^{2n}}$ can be represented by the form of $x\alpha + y(\alpha + 1)$ $(x, y \in \mathbf{F}_{2^n})$.

4.2.1 Addition

It's trivial to calculate by the following equation.

$$(x_1\alpha + y_1(\alpha + 1)) + (x_2\alpha + y_2(\alpha + 1)) = (x_1 + x_2)\alpha + (y_1 + y_2)(\alpha + 1)$$

Based on the equation above, we develop the following algorithm.

Algorithm 5 (Addition (Normal Basis))

Input: (x_1, y_1), (x_2, y_2)
Output: (x_3, y_3)

Step 1: $x_3 := x_1 + x_2$
Step 2: $y_3 := y_1 + y_2$

So, we can calculate an $\mathbf{F}_{2^{2n}}$-addition using 2 $\mathbf{F}_{2^n}$-additions.

4.2.2 Multiplication

A multiplication of two elements is calculated by

$$\begin{aligned}(x_1\alpha + y_1(\alpha + 1)) \times (x_2\alpha + y_2(\alpha + 1)) \ &= \ (x_1x_2 + a(x_1 + y_1)(x_2 + y_2))\alpha \\ &\quad + (y_1y_2 + a(x_1 + y_1)(x_2 + y_2))(\alpha + 1).\end{aligned}$$

Based on the equation above, we develop the following algorithm.

Algorithm 6 (Multiplication (Normal Basis))

Input: (x_1, y_1), (x_2, y_2)
Output: (x_3, y_3)

Step 1: $t_1 := x_1 + y_1$
Step 2: $t_2 := x_2 + y_2$
Step 3: $t_2 := t_1 \times t_2$ $(= (x_1 + y_1)(x_2 + y_2))$
Step 4: $t_1 := a \times t_2$ $(= a(x_1 + y_1)(x_2 + y_2))$
Step 5: $t_2 := x_1 \times x_2$
Step 6: $x_3 := t_1 + t_2$ $(= x_1x_2 + a(x_1 + y_1)(x_2 + y_2))$
Step 7: $t_2 := y_1 \times y_2$
Step 8: $y_3 := t_1 + t_2$ $(= y_1y_2 + a(x_1 + y_1)(x_2 + y_2))$

So, we can calculate an $\mathbf{F}_{2^{2n}}$-multiplication using 4 $\mathbf{F}_{2^n}$-additions and 4 $\mathbf{F}_{2^n}$-multiplications.

4.2.3 Squaring

Arranging the formula of the multiplication for a squaring, we have

$$(x_1\alpha + y_1(\alpha+1))^2 = (x_1^2 + a(x_1^2 + y_1^2))\alpha + (y_1^2 + a(x_1^2 + y_1^2))(\alpha+1).$$

Based on the equation above, we develop the following algorithm.

Algorithm 7 (Squaring (Normal Basis))

Input: (x_1, y_1)
Output: (x_3, y_3)

Step 1: $t_1 := x_1^2$
Step 2: $t_2 := y_1^2$
Step 3: $t_3 := t_1 + t_2$ $(= x_1^2 + y_1^2)$
Step 4: $t_3 := a \times t_3$ $(= a(x_1^2 + y_1^2))$
Step 5: $x_3 := t_1 + t_3$ $(= x_1^2 + a(x_1^2 + y_1^2))$
Step 6: $y_3 := t_2 + t_3$ $(= y_1^2 + a(x_1^2 + y_1^2))$

So, we can calculate an $\mathbf{F}_{2^{2n}}$-squaring using 3 $\mathbf{F}_{2^n}$-additions, an $\mathbf{F}_{2^n}$-multiplication, and 2 $\mathbf{F}_{2^n}$-squarings.

4.2.4 Inversion

We calculate similar to the case of the standard basis. Using the property of norm, then we have

$$(x_1\alpha + y_1(\alpha+1))(y_1\alpha + x_1(\alpha+1)) = a(x_1 + y_1)^2 + x_1y_1.$$

So the inverse is obtained by the following equation[4].

$$(x_1\alpha + y_1(\alpha+1))^{-1} = (a(x_1 + y_1)^2 + x_1y_1)^{-1}(y_1\alpha + x_1(\alpha+1))$$

Based on the equation above, we develop the following algorithm.

Algorithm 8 (Inversion (Normal Basis))

Input: (x_1, y_1)
Output: (x_3, y_3)

Step 1: $t_1 := x_1$
Step 2: $t_2 := x_1 + y_1$
Step 3: $t_2 := t_2^2$ $(= (x_1 + y_1)^2)$
Step 4: $t_2 := a \times t_2$ $(= a(x_1 + y_1)^2)$
Step 5: $t_3 := x_1 \times y_1$
Step 6: $t_2 := t_2 + t_3$ $(= x_1y_1 + a(x_1 + y_1)^2)$
Step 7: $t_2 := t_2^{-1}$ $(= (x_1y_1 + a(x_1 + y_1)^2)^{-1})$
Step 8: $x_3 := y_1 \times t_2$ $(= y_1(x_1y_1 + a(x_1 + y_1)^2)^{-1})$
Step 9: $y_3 := t_1 \times t_2$ $(= x_1(x_1y_1 + a(x_1 + y_1)^2)^{-1})$

So, we can calculate an $\mathbf{F}_{2^{2n}}$-inversion using 2 $\mathbf{F}_{2^n}$-additions, 4 $\mathbf{F}_{2^n}$-multiplications, an $\mathbf{F}_{2^n}$-squaring, and an $\mathbf{F}_{2^n}$-inversion.

4.3 Comparison with the Standard Basis and the Normal Basis

The results of Sects. 4.1 and 4.2 are summarized in Table 1. This table shows that the standard basis is superior to the normal basis.

[4]This equation can also be obtained using a basis transformation and the extended Euclidean algorithm.

Table 1: Comparison between the Standard Basis and the Normal Basis

$\mathbf{F}_{2^{2n}}$-operation		Number of $\mathbf{F}_{2^n}$-operations: Addition	Multiplication	Squaring	Inversion	Depth
Addition	Standard Basis	2				1
	Normal Basis	2				1
Multiplication	Standard Basis	4	4			3
	Normal Basis	4	4			4
Squaring	Standard Basis	1	1	2		3
	Normal Basis	3	1	2		4
Inversion	Standard Basis	2	4	1	1	5
	Normal Basis	2	4	1	1	6

Note that 'depth' means a circuit depth which regards $\mathbf{F}_{2^n}$ circuit as 1 circuit.

5 Estimation of Complexity

The algorithms proposed above reduce a field $\mathbf{F}_{2^{c2^t}}$ operation to subfield $\mathbf{F}_{2^{c2^{t-1}}}$ operations recursively[5]. So, if we choose an appropriate subfield considering an implementation environment of computers and calculate all subfield operations and store them in the memory in advance, we can enhance the performance of the computer well.

We summarize the complexity of an $\mathbf{F}_{2^{c2^t}}$-addition in Table 2, and the complexity of an $\mathbf{F}_{2^{c2^t}}$-multiplication in Table 3, and the complexity of an $\mathbf{F}_{2^{c2^t}}$-squaring in Table 4, and the complexity of an $\mathbf{F}_{2^{c2^t}}$-inversion in Table 5 using subfield $\mathbf{F}_{2^c}$ operations. This table is given by solving following recurrences. We define A_i^{op} as the number of additions, and M_i^{op} as the number of multiplications, and S_i^{op} as the number of squarings, and I_i^{op} as the number of inversions over $\mathbf{F}_{2^{c2^{t-i}}}$ required for an operation 'op' over $\mathbf{F}_{2^{c2^t}}$. 'Addition' is abbreviated to 'add,' and 'multiplication' is abbreviated to 'mul,' and 'squaring' is abbreviated to 'sqr,' and 'inversion' is abbreviated to 'inv.' These recurrences are easily derived from Table 1 based on the standard basis and easily solved (see Appendix A).

For example, we explain how to derive the recurrence of multiplication. Our multiplication algorithm requires the subfield multiplication and addition operations. So, we should consider the recurrence of A_i^{mul} and M_i^{mul}. Since $\mathbf{F}_{2^n}$-multiplication is only derived from $\mathbf{F}_{2^{2n}}$-multiplication and an $\mathbf{F}_{2^{2n}}$-multiplication requires 4 $\mathbf{F}_{2^n}$-multiplications, $M_{i+1}^{\mathrm{mul}} = 4M_i^{\mathrm{mul}}$ holds. However, since $\mathbf{F}_{2^n}$-addition is derived from $\mathbf{F}_{2^{2n}}$-multiplications and $\mathbf{F}_{2^{2n}}$-additions, and an $\mathbf{F}_{2^{2n}}$-multiplication requires 4 $\mathbf{F}_{2^n}$-additions and an $\mathbf{F}_{2^{2n}}$-addition requires 2 $\mathbf{F}_{2^n}$-additions, $A_{i+1}^{\mathrm{mul}} = 4M_i^{\mathrm{mul}} + 2A_i^{\mathrm{mul}}$ holds.

$$\begin{cases} A_0^{\mathrm{add}} &= 1 \\ A_{i+1}^{\mathrm{add}} &= 2A_i^{\mathrm{add}} \end{cases}$$

$$\begin{cases} A_0^{\mathrm{mul}} &= 0 \\ A_{i+1}^{\mathrm{mul}} &= 2A_i^{\mathrm{mul}} + 4M_i^{\mathrm{mul}} \end{cases}$$

$$\begin{cases} M_0^{\mathrm{mul}} &= 1 \\ M_{i+1}^{\mathrm{mul}} &= 4M_i^{\mathrm{mul}} \end{cases}$$

[5]Our algorithms have a recursive structure. So, you may worry about the overhead of a recursive call. However, since almost application of the finite field $\mathbf{F}_{2^n}$ fixes n, the inline expansion can reduce the overhead.

$$\begin{cases} A_0^{\mathrm{sqr}} &= 0 \\ A_{i+1}^{\mathrm{sqr}} &= 2A_i^{\mathrm{sqr}} + 4M_i^{\mathrm{sqr}} + S_i^{\mathrm{sqr}} \end{cases}$$
$$\begin{cases} M_0^{\mathrm{sqr}} &= 0 \\ M_{i+1}^{\mathrm{sqr}} &= 4M_i^{\mathrm{sqr}} + S_i^{\mathrm{sqr}} \end{cases}$$
$$\begin{cases} S_0^{\mathrm{sqr}} &= 1 \\ S_{i+1}^{\mathrm{sqr}} &= 2S_i^{\mathrm{sqr}} \end{cases}$$

$$\begin{cases} A_0^{\mathrm{inv}} &= 0 \\ A_{i+1}^{\mathrm{inv}} &= 2A_i^{\mathrm{inv}} + 4M_i^{\mathrm{inv}} + S_i^{\mathrm{inv}} + 2I_i^{\mathrm{inv}} \end{cases}$$
$$\begin{cases} M_0^{\mathrm{inv}} &= 0 \\ M_{i+1}^{\mathrm{inv}} &= 4M_i^{\mathrm{inv}} + S_i^{\mathrm{inv}} + 4I_i^{\mathrm{inv}} \end{cases}$$
$$\begin{cases} S_0^{\mathrm{inv}} &= 0 \\ S_{i+1}^{\mathrm{inv}} &= 2S_i^{\mathrm{inv}} + I_i^{\mathrm{inv}} \end{cases}$$
$$\begin{cases} I_0^{\mathrm{inv}} &= 1 \\ I_{i+1}^{\mathrm{inv}} &= I_i^{\mathrm{inv}} \end{cases}$$

Table 2: The Complexity of an $\mathbf{F}_{2^{e2^t}}$-Addition Using $\mathbf{F}_{2^e}$-Operations

Number of $\mathbf{F}_{2^e}$-additions (A_t^{add})	2^t

Table 3: The Complexity of an $\mathbf{F}_{2^{e2^t}}$-Multiplication Using $\mathbf{F}_{2^e}$-Operations

Number of $\mathbf{F}_{2^e}$-additions (A_t^{mul})	$2 \cdot 4^t - 2 \cdot 2^t$
Number of $\mathbf{F}_{2^e}$-multiplications (M_t^{mul})	4^t
Size of multiplication table	$e4^e$-bit

Table 4: The Complexity of an $\mathbf{F}_{2^{e2^t}}$-Squaring Using $\mathbf{F}_{2^e}$-Operations

Number of $\mathbf{F}_{2^e}$-additions (A_t^{sqr})	$4^t - (1 + \frac{t}{2})2^t$
Number of $\mathbf{F}_{2^e}$-multiplications (M_t^{sqr})	$\frac{1}{2}4^t - \frac{1}{2}2^t$
Number of $\mathbf{F}_{2^e}$-squaring (S_t^{sqr})	2^t

Table 5: The Complexity of an $\mathbf{F}_{2^{e2^t}}$-Inversion Using $\mathbf{F}_{2^e}$-Operations

Number of $\mathbf{F}_{2^e}$-additions (A_t^{inv})	$3 \cdot 4^t - (6 + \frac{t}{2})2^t + 3$
Number of $\mathbf{F}_{2^e}$-multiplications (M_t^{inv})	$\frac{3}{2}4^t - \frac{1}{2}2^t - 1$
Number of $\mathbf{F}_{2^e}$-squaring (S_t^{inv})	$2^t - 1$
Number of $\mathbf{F}_{2^e}$-inversions (I_t^{inv})	1

For example, these tables show that an $\mathbf{F}_{2^{32}}$-multiplication requires 16 $\mathbf{F}_{2^8}$-multiplications and 24 $\mathbf{F}_{2^8}$-additions[6] in the case of $t = 2$ and $e = 8$. In this case, this algorithm requires 64KB[7] memory. For another example, this table shows that an $\mathbf{F}_{2^{32}}$-multiplication requires 256 $\mathbf{F}_{2^2}$-multiplications and 480 $\mathbf{F}_{2^2}$-additions. In this case, this algorithm requires only 32B memory.

We point out that there exists a trade-off between the complexity and the size of memory required. This trade-off permits us to select an appropriate subfield for each computer.

6 Comparison with Previous Algorithms

6.1 Multiplication

Pincin proposed a multiplication algorithm over $\mathbf{F}_{p^n}$ which is fast if n is highly composite [P89]. We compare our algorithm to his algorithm in Table 6 for the case of a multiplication over $\mathbf{F}_{2^{2^t}}$. Note that this case is most effective for his algorithm.

Our algorithm reduce the number of bit-multiplications from 5^t to 4^t and the number of bit-additions from $\frac{7}{3}5^t - \frac{7}{3}2^t$ to $2 \cdot 4^t - 2 \cdot 2^t$. The larger t is, the more superior our algorithm is to Pincin's algorithm.

[6] If e is sufficiently small, several additions require only 1 computer *exclusive or* operation in a well represented implementation.

[7] 1B means 8-bit.

Table 6: Number of Bit Operations

	Pincin's algorithm	Our algorithm
Number of multiplications	5^t	4^t
Number of additions	$\frac{7}{3}5^t - \frac{7}{3}2^t$	$2 \cdot 4^t - 2 \cdot 2^t$

6.2 Inversion

We compare our algorithm to previous algorithm from two points of view.

6.2.1 Exponentiation with a Normal Basis and a Successive Extension

We compare our algorithm with Itoh and Tsujii's $\mathbf{F}_{q^n}$-inversion algorithm [IT87][8]. Their algorithm uses the fact that $x^{q^n-1} = 1$ over $\mathbf{F}_{q^n}$ for calculating an inversion, while our algorithm is based on the extended Euclidean algorithm and successive extension as explained in Sect. 4.1.4.

Table 7: Number of Operations Required for Itoh's Algorithm for an Inversion over $\mathbf{F}_{q^n}$

Number of $\mathbf{F}_{q^n}$-multiplication	$\lfloor \log_2(n-1) \rfloor + H_w(n-1) + 1$
Number of $\mathbf{F}_q$-inversion	1

(H_w: Hamming weight)

We compare them over $\mathbf{F}_{2^{16 \cdot 2^t}}$ realized in Table 8. This table is derived from Table 5 and Table 7 [IT87]. We assume that the complexity of an $\mathbf{F}_{2^{16}}$-inversion can be ignored because both algorithms require only one $\mathbf{F}_{2^{16}}$-inversion which has roughly the same complexity of an $\mathbf{F}_{2^{16}}$-multiplication. Then, we show the two cases which are regarded as smallest and largest complexity. First, we assume that the complexity of an $\mathbf{F}_{2^{16}}$-addition can be ignored for our algorithm. Secondly, we assume that the complexity of an $\mathbf{F}_{2^{16}}$-addition has the same complexity of $\mathbf{F}_{2^{16}}$-multiplication for our algorithm. Both case we also assume that an $\mathbf{F}_{2^{16}}$-squaring has the same complexity of $\mathbf{F}_{2^{16}}$-multiplication for our algorithm. Since our algorithm does not require an $\mathbf{F}_{2^{16 \cdot 2^t}}$-multiplication for calculating an $\mathbf{F}_{2^{16 \cdot 2^t}}$-inversion, we show the ratio of the number of $\mathbf{F}_{2^{16}}$-multiplications for calculating an $\mathbf{F}_{2^{16 \cdot 2^t}}$-inversion over the number of $\mathbf{F}_{2^{16}}$-multiplications for an $\mathbf{F}_{2^{16 \cdot 2^t}}$-multiplication in Table 8.

Table 8: Number of Multiplications Required for an Inversion

		Our algorithm	
	Itoh's algorithm	Additions ignored	Addition regarded as the same complexity of multiplication
$\mathbf{F}_{2^{32}}$ $(= \mathbf{F}_{2^{16 \cdot 2^1}})$	2	1.5	1.2
$\mathbf{F}_{2^{128}} (= \mathbf{F}_{2^{16 \cdot 2^3}})$	6	1.5	1.3
$\mathbf{F}_{2^{256}} (= \mathbf{F}_{2^{16 \cdot 2^4}})$	8	1.5	1.4

The number of multiplications for our algorithm converges about 1.5 if we ignore the number of $\mathbf{F}_{2^{16}}$-additions, or also converges about 1.5 if we regard that the complexity of $\mathbf{F}_{2^{16}}$-addition has the same complexity of $\mathbf{F}_{2^{16}}$-multiplication, while the number of multiplications for Itoh's algorithm asymptotically diverges to infinity.

[8] Itoh's algorithm is the most efficient technique in terms of minimizing the number of multiplications to compute an inverse according to [ABMV93].

6.2.2 Standard Basis and Successive Extension

Win et al. proposed an $\mathbf{F}_{2^{16\cdot s}}$-inversion algorithm [WBV+96]. Since Win's algorithm uses a standard basis derived from an irreducible trinomial over the subfield $\mathbf{F}_{2^{16}}$ with coefficients 1[9], Win's algorithm can be applicable to the case where there exists an irreducible trinomial over a subfield, while our algorithm can be applicable to the case where the field is 2^t-th extension of a subfield. The concrete construction of the irreducible polynomial has been given in Sect. 3 for our cases.

The complexity of an inversion using subfield depends on how to choose a subfield considering the table size required by implementation environment of a computer. Win's algorithm directly uses the extended Euclidean algorithm. Roughly speaking, the basic loop of the extended Euclidean algorithm requires about extension degree subfield-multiplication/inversion and the number of the iteration of the loop is about two times of extension degree, so this algorithm requires about two times of squaring of extension degree subfield-multiplication/inversion. On the other hand, our algorithm requires about 1.5 times of squaring of extension degree subfield-multiplication/inversion according to Sect. 5. The superiority of algorithms depends on the extension degree.

For example, we consider an $\mathbf{F}_{2^{176}}$-inversion. Win's algorithm requires about $2 \cdot 11^2 (= 242)$ $\mathbf{F}_{2^{16}}$-multiplication/inversions, since they can choose a subfield as $\mathbf{F}_{2^{16}}$ and $\dim_{\mathbf{F}_{2^{16}}} \mathbf{F}_{2^{176}} = 11$. Our algorithm requires about $1.5 \cdot 16^2 (= 384)$ $\mathbf{F}_{2^{11}}$-multiplication/inversions, since we can choose a subfield as $\mathbf{F}_{2^{11}}$ and $\dim_{\mathbf{F}_{2^{11}}} \mathbf{F}_{2^{176}} = 16$. Thus, Win's algorithm for an $\mathbf{F}_{2^{176}}$-inversion may be faster than our algorithm in this case. If $\mathbf{F}_{2^{22}}$-multiplication/inversion table is acceptable in our case, our algorithm may be faster than theirs, since $\dim_{\mathbf{F}_{2^{22}}} \mathbf{F}_{2^{176}} = 8$ and $1.5 \cdot 8^2 = 96$.

Let us consider another example, an $\mathbf{F}_{2^{240}}$-inversion. Win's algorithm requires about $2 \cdot 15^2 (= 450)$ $\mathbf{F}_{2^{16}}$-multiplication/inversions, since they can choose a subfield as $\mathbf{F}_{2^{16}}$ and $\dim_{\mathbf{F}_{2^{16}}} \mathbf{F}_{2^{240}} = 15$. Our algorithm requires about $1.5 \cdot 16^2 (= 384)$ $\mathbf{F}_{2^{15}}$-multiplication/inversions, since we can choose a subfield as $\mathbf{F}_{2^{15}}$ and $\dim_{\mathbf{F}_{2^{15}}} \mathbf{F}_{2^{240}} = 16$. Thus, Win's algorithm for an $\mathbf{F}_{2^{240}}$-inversion may be slower than our algorithm.

7 Implementation of Application

We have implemented our algorithm written in ANSI-C. We timed our routines using $\mathbf{F}_{2^8}$-multiplication table and $\mathbf{F}_{2^{16}}$-inversion table which require 256KB of cache memory. We used the GNU C compiler version 2.7.2.2 and executed the tests on a HyperSPARC/125MHz based WS.

7.1 Block Cipher

We implemented prototype of 64-bit block cipher based on Knudsen-Nyberg construction [NK95, N95] using $\mathbf{F}_{2^{32}}$-inversion as F-function and having 8 round. This cipher is provably secure against differential attack [BS93] and linear attack [M94], and may be secure against higher order differential attack [K95] since algebraic degree of $\mathbf{F}_{2^{32}}$-inversion is 31 [N94, Proposition 4][10]. The prototype cipher achieves about 20Mb/s.

7.2 Elliptic Curve

We estimate the timings for calculating non-supersingular elliptic curve operation and compare with Win's algorithm [WBV+96, Table 2]. The result is described in Table 9. Win et al. execute the tests on a Pentium/133MHz based PC. We assume that elliptic curve addition requires 8 $\mathbf{F}_{2^n}$-addition, 2 $\mathbf{F}_{2^n}$-multiplication, an $\mathbf{F}_{2^n}$-squaring, and an $\mathbf{F}_{2^n}$-inversion, and elliptic curve doubling requires 5 $\mathbf{F}_{2^n}$-addition, 2 $\mathbf{F}_{2^n}$-multiplication, 2 $\mathbf{F}_{2^n}$-squaring, and an $\mathbf{F}_{2^n}$-inversion. We also assume that our full-bit exponentiation uses simple double-and-add/subtract algorithm which requires 256 doublings and on average 86 additions/subtractions.

According to Table 9, we expect that our algorithm is superior to Win's algorithm. However, because we compare our algorithm with Win's algorithm on a similar CPU clock speed but different architecture, we cannot conclude the expectation.

[9] Computer exhaustive search proves that there does not exist an irreducible trinomial with degree 2^t $(3 \leq t \leq 11)$ over $\mathbf{F}_2$.

[10] However, this cipher is very vulnerable against the interpolation attack [JK97]. So this cipher cannot be used practically. Against the interpolation attack, Matsui's construction methodology of unbalanced network may be effective [M96].

Table 9: Implementation Speed Compared with Win's Algorithm

	Our algorithm ($\mathbf{F}_{2^{256}}$)	Win's algorithm ($\mathbf{F}_{2^{176}}$)
Multiplication	27.9μs	64.5μs
Inversion	40.9μs	160μs
Squaring	14.0μs	5.9μs
Addition	0.1μs	1.2μs
EC addition (estimation)	112μs	306μs
EC doubling (estimation)	125μs	309μs
EC exponentiation with full bit exponent (estimation)	41ms	72ms

Note that the speeds of our algorithm are estimated on HyperSPARC/125MHz, while the speeds of Win's algorithm are estimated on Pentium/133MHz.

8 Conclusion

This paper has proposed the construction of a quadratic extension with characteristic 2 and fast inversion algorithms over this extension field. Our algorithms are regarded as a hybrid of Pincin's and Paar's algorithms [P89, P96]. While Pincin's and Paar's papers only show a multiplication algorithm, we developed an inversion algorithm. Our inversion algorithm requires only 1.5 times of the complexity of a multiplication. Our algorithms are well suited for software implementation using table reference because recent processors have large cache memory.

Acknowledgments

We thank Dr. Tsunogai of Sophia University for noticing that the effective construction technique of a quadratic extension (Lemma 1) and thank anonymous referees for helpful comments improving our paper.

References

[ABMV93] G. B. Agnew, T. Beth, R. C. Mullin, and S. A. Vanstone. Arithmetic Operations in $GF(2^m)$. *Journal of Cryptology*, Vol. 6, No. 1, pp. 3–13, 1993.

[BS93] E. Biham and A. Shamir. *Differential Cryptanalysis of the Data Encryption Standard.* Springer-Verlag, Berlin, Heidelberg, New York, 1993.

[CV95] F. Chabaud and S. Vaudenay. Links Between Differential and Linear Cryptanalysis. In A. D. Santis, editor, *Advances in Cryptology — EUROCRYPT'94*, Volume 950 of *Lecture Notes in Computer Science*, pp. 356–365. Springer-Verlag, Berlin, Heidelberg, New York, 1995.

[HMV93] G. Harper, A. Menezes, and S. Vanstone. Public-Key Cryptosystems with Very Small Key Lengths. In R. A. Rueppel, editor, *Advances in Cryptology — EUROCRYPT'92*, Volume 658 of *Lecture Notes in Computer Science*, pp. 163–173. Springer-Verlag, Berlin, Heidelberg, New York, 1993.

[IT87] T. Itoh and S. Tsujii. A Fast Algorithm for Computing Multiplicative Inverses in Finite Fields Using Normal Bases. *IEICE Transactions Fundamentals of Electronics, Communications and Computer Sciences (Japan)*, Vol. J70–A, No. 11, pp. 1637–1645, 1987. (in Japanese).

[JK97] T. Jakobsen and L. R. Knudsen. The Interpolation Attack on Block Cipher. In E. Biham, editor, *Fast Software Encryption Workshop (FSE4) Preproceedings*, pp. 28–40, Technion, Haifa, Israel, 1997.

[K95] L. R. Knudsen. Truncated and Higher Order Differentials. In B. Preneel, editor, *Fast Software Encryption — Second International Workshop*, Volume 1008 of *Lecture Notes in Computer Science*, pp. 196–211. Springer-Verlag, Berlin, Heidelberg, New York, 1995.

[M94] M. Matsui. Linear Cryptanalysis Method for DES Cipher. In T. Helleseth, editor, *Advances in Cryptology — EUROCRYPT'93*, Volume 765 of *Lecture Notes in Computer Science*, pp. 386–397. Springer-Verlag, Berlin, Heidelberg, New York, 1994. (Preliminary version written in Japanese was presented at SCIS93-3C).

[M96] M. Matsui. New Structure of Block Ciphers with Provable Security against Differential and Linear Cryptanalysis. In D. Gollmann, editor, *Fast Software Encryption, Third International Workshop, Cambridge, UK, February 1996, Proceedings*, Volume 1039 of *Lecture Notes in Computer Science*, pp. 205–218. Springer-Verlag, Berlin, Heidelberg, New York, 1996.

[MO81] J. L. Massey and J. K. Omura. *Computational Method and Apparatus for Finite Field Arithmetic*, 1981. US Patent Number 4,587,627.

[N94] K. Nyberg. Differentially uniform mappings for cryptography. In T. Helleseth, editor, *Advances in Cryptology — EUROCRYPT'93*, Volume 765 of *Lecture Notes in Computer Science*, pp. 55–64. Springer-Verlag, Berlin, Heidelberg, New York, 1994.

[N95] K. Nyberg. Linear Approximation of Block Ciphers. In A. D. Santis, editor, *Advances in Cryptology — EUROCRYPT'94*, Volume 950 of *Lecture Notes in Computer Science*, pp. 439–444. Springer-Verlag, Berlin, Heidelberg, New York, 1995.

[NK95] K. Nyberg and L. R. Knudsen. Provable Security Against a Differential Attack. *Journal of Cryptology*, Vol. 8, No. 1, pp. 27–37, 1995. (A Preliminary version was presented at CRYPTO'92 rump session).

[P89] A. Pincin. A New Algorithm for Multiplication in Finite Fields. *IEEE Transactions on computers*, Vol. 38, No. 7, pp. 1045–1049, 1989.

[P96] C. Paar. A New Architecture for a Parallel Finite Field Multiplier with Low Complexity Based on Composite Fields. *IEEE Transactions on computers*, Vol. 45, No. 7, pp. 856–861, 1996.

[S90] V. Shoup. New Algorithms for Finding Irreducible Polynomials over Finite Fields. *Mathematics of Computation*, Vol. 54, No. 189, pp. 435–447, 1990.

[WBV+96] E. D. Win, A. Bosselaers, S. Vandenberghe, P. D. Gersem, and J. Vandewalle. A Fast Software Implementation for Arithmetic Operations in GF(2^n). In K. Kim and T. Matsumoto, editors, *Advances in Cryptology — ASIACRYPT'96*, Volume 1163 of *Lecture Notes in Computer Science*, pp. 65–76. Springer-Verlag, Berlin, Heidelberg, New York, 1996.

A Solving Recurrences

In Sect. 5, we have many recurrences. This section solves them.

A.1 Addition

We solve the following recurrence.

$$\begin{cases} A_0^{\mathrm{add}} &= 1 \\ A_{i+1}^{\mathrm{add}} &= 2A_i^{\mathrm{add}} \end{cases}$$

It's trivial that

$$A_i^{\mathrm{add}} = 2^i.$$

A.2 Multiplication

We solve the following recurrences.

$$\begin{cases} A_0^{\mathrm{mul}} &= 0 \\ A_{i+1}^{\mathrm{mul}} &= 2A_i^{\mathrm{mul}} + 4M_i^{\mathrm{mul}} \end{cases} \qquad \begin{cases} M_0^{\mathrm{mul}} &= 1 \\ M_{i+1}^{\mathrm{mul}} &= 4M_i^{\mathrm{mul}} \end{cases}$$

First we solve M_i^{mul}. It's trivial that

$$M_i^{\mathrm{mul}} = 4^i.$$

Secondly, we solve A_i^{mul}. Using the results of M_i^{mul}, we have

$$\begin{aligned} & A_{i+1}^{\mathrm{mul}} = 2A_i^{\mathrm{mul}} + 4 \cdot 4^i \\ \Leftrightarrow \quad & A_{i+1}^{\mathrm{mul}} - 2 \cdot 4^{i+1} = 2(A_i^{\mathrm{mul}} - 2 \cdot 4^i). \end{aligned}$$

Therefore, we have

$$\begin{aligned} & A_i^{\mathrm{mul}} - 2 \cdot 4^i = 2^i(A_0^{\mathrm{mul}} - 2 \cdot 4^0) = -2 \cdot 2^i \\ \Leftrightarrow \quad & A_i^{\mathrm{mul}} = 2 \cdot 4^i - 2 \cdot 2^i. \end{aligned}$$

A.3 Squaring

We solve the following recurrences.

$$\begin{cases} A_0^{\mathrm{sqr}} &= 0 \\ A_{i+1}^{\mathrm{sqr}} &= 2A_i^{\mathrm{sqr}} + 4M_i^{\mathrm{sqr}} + S_i^{\mathrm{sqr}} \end{cases} \qquad \begin{cases} M_0^{\mathrm{sqr}} &= 0 \\ M_{i+1}^{\mathrm{sqr}} &= 4M_i^{\mathrm{sqr}} + S_i^{\mathrm{sqr}} \end{cases} \qquad \begin{cases} S_0^{\mathrm{sqr}} &= 1 \\ S_{i+1}^{\mathrm{sqr}} &= 2S_i^{\mathrm{sqr}} \end{cases}$$

First we solve S_i^{sqr}. It's trivial that

$$S_i^{\mathrm{sqr}} = 2^i.$$

Secondly, we solve M_i^{sqr}. Using the results of S_i^{sqr}, we have

$$\begin{aligned} & M_{i+1}^{\mathrm{sqr}} = 4M_i^{\mathrm{sqr}} + 2^i \\ \Leftrightarrow \quad & M_{i+1}^{\mathrm{sqr}} + \frac{1}{2}2^{i+1} = 4(M_i^{\mathrm{sqr}} + \frac{1}{2}2^i). \end{aligned}$$

Therefore, we have

$$\begin{aligned} & M_i^{\mathrm{sqr}} + \frac{1}{2}2^i = 4^i(M_0^{\mathrm{sqr}} + \frac{1}{2}2^0) = \frac{1}{2}4^i \\ \Leftrightarrow \quad & M_i^{\mathrm{sqr}} = \frac{1}{2}4^i - \frac{1}{2}2^i. \end{aligned}$$

Thirdly, we solve A_i^{sqr}. Using the results of S_i^{sqr} and M_i^{sqr}, we have

$$\begin{aligned} & A_{i+1}^{\mathrm{sqr}} = 2A_i^{\mathrm{sqr}} + 2 \cdot 4^i - 2^i \\ \Leftrightarrow \quad & \frac{A_{i+1}^{\mathrm{sqr}} - 4^{i+1}}{2^{i+1}} = \frac{A_i^{\mathrm{sqr}} - 4^i}{2^i} - \frac{1}{2} \end{aligned}$$

Therefore, we have

$$\begin{aligned} & \frac{A_i^{\mathrm{sqr}} - 4^i}{2^i} = \frac{A_0^{\mathrm{sqr}} - 4^0}{2^0} - \frac{i}{2} = -(1 + \frac{i}{2}) \\ \Leftrightarrow \quad & A_i^{\mathrm{sqr}} = 4^i - (1 + \frac{i}{2})2^i. \end{aligned}$$

A.4 Inversion

We solve the following recurrences.

$$\begin{cases} A_0^{\mathrm{inv}} &= 0 \\ A_{i+1}^{\mathrm{inv}} &= 2A_i^{\mathrm{inv}} + 4M_i^{\mathrm{inv}} + S_i^{\mathrm{inv}} + 2I_i^{\mathrm{inv}} \end{cases} \qquad \begin{cases} M_0^{\mathrm{inv}} &= 0 \\ M_{i+1}^{\mathrm{inv}} &= 4M_i^{\mathrm{inv}} + S_i^{\mathrm{inv}} + 4I_i^{\mathrm{inv}} \end{cases}$$

$$\begin{cases} S_0^{\mathrm{inv}} &= 0 \\ S_{i+1}^{\mathrm{inv}} &= 2S_i^{\mathrm{inv}} + I_i^{\mathrm{inv}} \end{cases} \qquad \begin{cases} I_0^{\mathrm{inv}} &= 1 \\ I_{i+1}^{\mathrm{inv}} &= I_i^{\mathrm{inv}} \end{cases}$$

First we solve I_i^{inv}. It's trivial that

$$I_i^{\text{inv}} = 1$$

Secondly, we solve S_i^{inv}. Using the results of I_i^{inv}, we have

$$\begin{aligned} & S_{i+1}^{\text{inv}} = 2S_i^{\text{inv}} + 1 \\ \Leftrightarrow \quad & S_{i+1}^{\text{inv}} + 1 = 2(S_i^{\text{inv}} + 1). \end{aligned}$$

Therefore, we have

$$\begin{aligned} & S_i^{\text{inv}} + 1 = 2^i(S_0^{\text{inv}} + 1) = 2^i \\ \Leftrightarrow \quad & S_i^{\text{inv}} = 2^i - 1. \end{aligned}$$

Thirdly, we solve M_i^{inv}. Using the results of I_i^{inv} and S_i^{inv}, we have

$$\begin{aligned} & M_{i+1}^{\text{inv}} = 4M_i^{\text{inv}} + 2^i + 3 \\ \Leftrightarrow \quad & M_{i+1}^{\text{inv}} + \frac{1}{2}2^{i+1} + 1 = 4(M_i^{\text{inv}} + \frac{1}{2}2^i + 1). \end{aligned}$$

Therefore, we have

$$\begin{aligned} & M_i^{\text{inv}} + \frac{1}{2}2^i + 1 = 4^i(M_0^{\text{inv}} + \frac{1}{2}2^0 + 1) = \frac{3}{2}4^i \\ \Leftrightarrow \quad & M_i^{\text{inv}} = \frac{3}{2}4^i - \frac{1}{2}2^i - 1. \end{aligned}$$

Lastly, we solve A_i^{inv}. Using the results of I_i^{inv}, S_i^{inv}, and M_i^{inv}, we have

$$\begin{aligned} & A_{i+1}^{\text{inv}} = 2A_i^{\text{inv}} + 6 \cdot 4^i - 2^i - 3 \\ \Leftrightarrow \quad & \frac{A_{i+1}^{\text{inv}} - 3 \cdot 4^{i+1} - 3}{2^{i+1}} = \frac{A_i^{\text{inv}} - 3 \cdot 4^i - 3}{2^i} - \frac{1}{2} \end{aligned}$$

Therefore, we have

$$\begin{aligned} & \frac{A_i^{\text{inv}} - 3 \cdot 4^i - 3}{2^i} = \frac{A_0^{\text{inv}} - 3 \cdot 4^0 - 3}{2^0} - \frac{i}{2} = -(6 + \frac{i}{2}) \\ \Leftrightarrow \quad & A_i^{\text{inv}} = 3 \cdot 4^i - (6 + \frac{i}{2})2^i + 3. \end{aligned}$$

A Block-Ciphering Algorithm Based on Addition-Multiplication Structure in GF(2^n)

Feng Zhu
Dept. of Computer Science and Technology, Tsinghua Univ.
Beijing, 100084, P.R.China
Email: zhufeng@theory.cs.tsinghua.edu.cn

Bao-An Guo
Dept. of Computer Science and Technology, Tsinghua Univ.
Beijing, 100084, P.R.China
Email: guoba@theory.cs.tsinghua.edu.cn

(Extended Abstract)

1. introduction

This paper describes a new block encryption algorithm. The block length is 64 bits for plaintext and ciphertext; the user-selected key is 192 bits in length. This block-ciphering algorithm is an iterated cipher in the sense that encrypting is performed by applying the same transformation repeatedly for r rounds, then applying an output transformation; r=6 is recommended but larger values of r can be used if desired for even greater security. Each round uses two 64-bit subkeys , four 16-bit subkeys determined by a key schedule from the secret 192-bit user-selected key. The output transformation uses other four 16-bit subkeys determined by the key schedule. To achieve security with encipher processing , this block-ciphering algorithm exploits one new cryptographic concept, namely Addition-Multiplication Structure. In encipher processing, the plaintext and subkey are looked upon as two element in the Galois Field GF(2^{64}), and then they are mixed by the additive and multiplicative operation in GF(2^{64}). It is proved that this secret-key block-enciphering algorithm is a Markov cipher and its maximum probability of 1-round differential is $1/(2^{64}-1)$, so it can resist differential cryptanalysis with few rounds.

The cipher is described in Section 2. The design principles for the cipher are discussed in

Section 3. Section 4 considers the maximum probability of 1-round differential of this block-ciphering algorithm and its property against differential cryptanalysis. .

2. Description of this block-ciphering algorithm

The computational graph of the encryption process is shown in Fig.1. The process consists of 6 similar rounds followed by an output transformation. The complete first round and the output transformation are depicted in this figure.

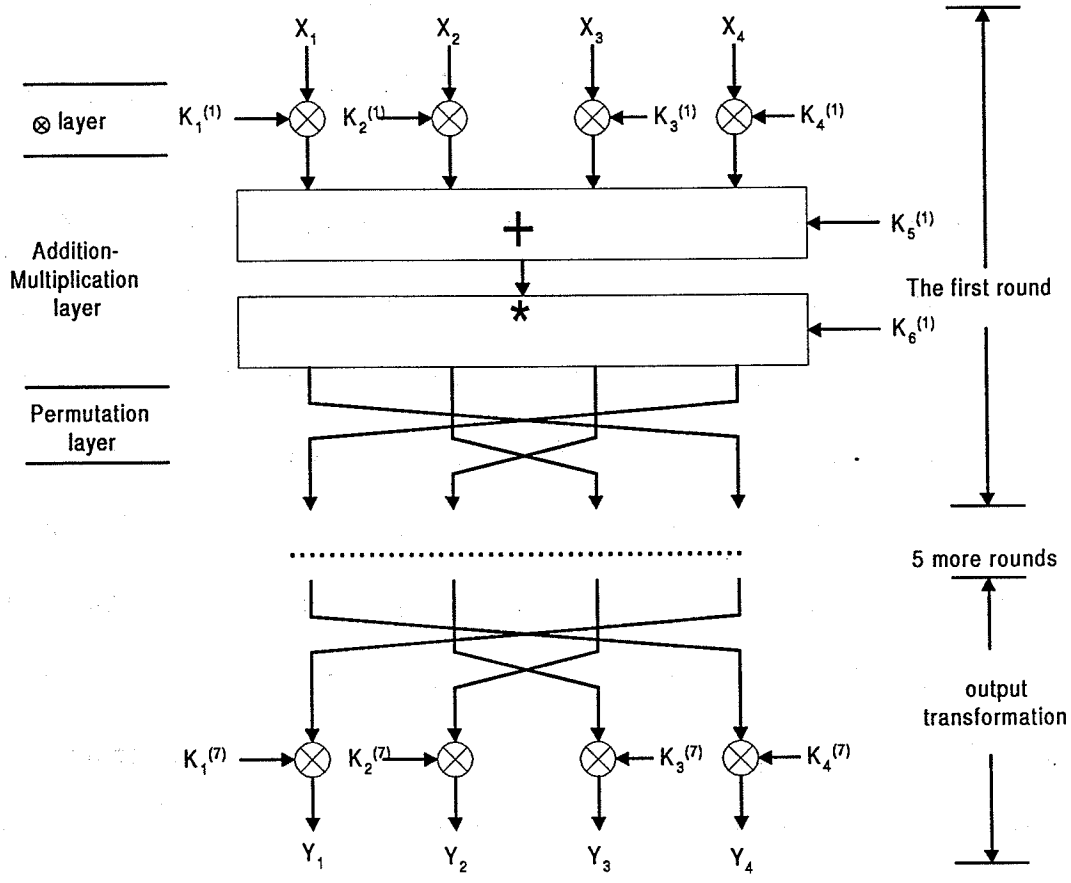


Fig.1

2.1 The encryption process

In the encryption process shown in Fig.1, two different operation on pairs of 64-bit block and one other operation on pairs of 16-bit subblock is used, namely,

— multiplication of polynomial modulo an irreducible polynomial in GF(2)[x] of which the

degree is 64 .The 64-bit block there is treated as a polynomial in GF(2)[x]. The resulted operation is denoted as * .

$x^{64}+x^4+x^3+x+1$ is recommended as the irreducible polynomial of degree 64 in GF(2)[x].

— bit-by-bit exclusive-OR of two 64-bit subblocks, denoted as +

Note that, this operation can be looked upon as an addition of polynomial modulo an irreducible polynomial of degree 64 in GF(2)[x]. The 64-bit block there is treated as a polynomial in GF(2)[x].

Meanwhile, it is well know that the field of polynomials over GF(2) modulo an irreducible polynomial of degree 64 forms the Galois field of 2^{64} elements $GF(2^{64})$.

— multiplication of integers modulo $2^{16}+1$ where the 16-bit subblock is treated as the usual radix-two representation of an integer except that the all-zero subblock is treated as 2^{16}; the resulted operation is denoted as $\otimes$.

As an example of these operations, note that

(0,0,.....0,0,0,1,0)*(1,0,.....0,,1,1,0,1)=(0,0,.....0,0,0,0,1)

because $x(x^{63}+x^3+x^2+1) \bmod (x^{64}+x^4+x^3+x+1) = 1$ in GF(2)[x]

(0,0, ... 0,0)$\otimes$(1,1,.....,1,1)=(0,0, ...,1,0)

because $2^{16}(2^{16}-1) \bmod (2^{16}+1)=2$.

The 64-bit plaintext block X is partitioned into four 16-bit subblocks X_1,X_2,X_3,X_4, i.e., $X=(X_1,X_2,X_3,X_4)$. The four plaintext subblock are then transformed into four 16-bit ciphertext subblocks Y_1,Y_2,Y_3,Y_4 , under the control of 12 key subblocks of 64 bits and 28 key subblocks of 16 bits that are formed from the 192 bit secret key in a manner to be described below. The ciphertext block is $Y=(Y_1,Y_2,Y_3,Y_4)$. The four key subblocks of 16 bits used in the *r*-th round

are denoted as $K_1^{(r)},K_2^{(r)},K_3^{(r)},K_4^{(r)}$. The two key subblocks of 64 bits used in the *r*-th round are denoted as $K_5^{(r)},K_6^{(r)}$. Four key subblocks of 16 bits that are used in the output transformation are denoted as $K_1^{(7)},K_2^{(7)},K_3^{(7)},K_4^{(7)}$.

In the *i*-th round, the input data, which is denoted as S, is partitioned into four 16-bit subblocks S_1,S_2,S_3,S_4 first. Then they are passed through a $\otimes$ layer which is controlled by subkey $K_1^{(i)},K_2^{(i)},K_3^{(i)},K_4^{(i)}$. The result of this transformation, which is denoted as T, is a 64-bit block consists of four 16-bit subblocks. This transformation is denoted as $\otimes_4$, i.e.,

$$S \otimes_4(K_1^{(i)},K_2^{(i)},K_3^{(i)},K_4^{(i)})=(S_1,S_2,S_3,S_4)\otimes_4(K_1^{(i)},K_2^{(i)},K_3^{(i)},K_4^{(i)})$$

$$=(S_1 \otimes K_1^{(i)},S_2 \otimes K_2^{(i)},S_3 \otimes K_3^{(i)},S_4 \otimes K_4^{(i)})=T$$

Then the 64-bits of the result of $\otimes$ layer , T, passes through an Addition-Multiplication layer which is under the control of subkey $K_5^{(r)},K_6^{(r)}$. The result of this transformation, which is denoted as U, is a 64-bit block. This transformation is denoted as AM, i.e.,

$$AM(T,K_5^{(r)},K_6^{(r)})=(T+K_5^{(r)})*K_6^{(r)}=U.$$

After that, the 64-bits of the result of the Addition-Multiplication layer, U, is partitioned into four 16-bit subblocks U_1,U_2,U_3,U_4 . They are then passed through a permutation layer. The result of this transformation, which is denoted as V, is a 64-bit block, consists of four 16-bit subblock. This transformation is denoted as Pe, i.e.,

$$Pe(U)=Pe((U_1,U_2,U_3,U_4))=(U_4,U_3,U_2,U_1)=V$$

The 64-bit block, V, is the output data of the *i*-th round and also is the input data of the (*i*+1)-th round.

In the output transformation the input data, which is denoted as V, is partitioned into four 16-bit subblocks V_1,V_2,V_3,V_4 first. Then they are passed through a permutation layer. The result of this transformation, which is denoted as W, is one 64-bit block, consists of four 16-bit subblocks, i.e.,

$$Pe(V)=Pe((V_1,V_2,V_3,V_4))=(V_4,V_3,V_2,V_1)=(W_1,W_2,W_3,W_4)=W$$

After that, W are passed through a $\otimes$ layer which is controlled by subkey $K_1^{(r+1)}$, $K_2^{(r+1)}$, $K_3^{(r+1)}$,$K_4^{(r+1)}$, where r is the total number of rounds. The result of this transformation, which is the ciphertext denoted as Y, is one 64-bit block, consists of four 16-bit subblocks. i.e.,

$$W \otimes_4 (K_1^{(r+1)},K_2^{(r+1)},K_3^{(r+1)},K_4^{(r+1)})=(W_1,W_2,W_3,W_4)\otimes_4 (K_1^{(r+1)},K_2^{(r+1)},K_3^{(r+1)},K_4^{(r+1)})$$

$$=(W_1 \otimes K_1^{(r+1)},W_2 \otimes K_2^{(r+1)},W_3 \otimes K_3^{(r+1)},W_4 \otimes K_4^{(r+1)})$$

$$=(Y_1,Y_2,Y_3,Y_4)=Y$$

Note that, the permutation in the output transformation counteracts the effect of the permutation in the last round.

2.2 The key schedule

The 12 key subblocks of 64 bits and 28 key subblocks of 16 bits used in the encryption process are generated from the 192-bit user-selected key.

First, initialize the array S of 1216 bit. The 192-bit user-selected key is directly used as the first 192 bits of S. Then 192-bit user-selected key is cyclic shifted to the left by 31 bit positions, after which the resulted 192-bit block is taken as the next 192 bits of S. The obtained 192-bit block is again cyclic shifted to the left by 31 bit positions to produce the next 192 bits of S, and

this procedure is repeated until all S have been filled.

Second , add biases to S. S is partitioned into 38 32-bit subblocks and then the j-th subblock is added to the integer part of 2^{32}*abs(sin(j)), where j is in radians.

Thirdly, check S. Set variable j equal to 39. S is partitioned into 19 64-bit subblocks. Check each subblock from left to right. If a subblock is (0,...0), the left half of the subblock is added to the integer part of 2^{32} × abs(sin(j)) and the right half of the subblock is added to the integer part of 2^{32} × abs(sin(j+1)), where j is in radians and then add 2 to variable j. The purpose of this step is to avoid weak subkey and to make it possible that decryption can be done correctly.

Finally, produce the key subblock. Let $K_1^{(1)}$ be the segment of S from the 1st bit to the 64th bits. Let $K_2^{(1)}$ be the segment of S from the 65th bit to the 128th bits. This procedure is repeated until all 40 key subblocks have been generated.

2.3 The decryption process

The computational graph of the decryption process is essentially the same as that of the encryption process, the only difference is the decryption key subblock $Z_i^{(r)}$ which are computed from the encryption key subblock $K_i^{(r)}$ as follows:

$(Z_1^{(r)}, Z_2^{(r)}, Z_3^{(r)}, Z_4^{(r)}) = ((K_1^{(8-r)})^{-1}, K_2^{(8-r)})^{-1}, K_3^{(8-r)})^{-1}, K_4^{(8-r)})^{-1})$ for r=1 and 7

$(Z_1^{(r)}, Z_2^{(r)}, Z_3^{(r)}, Z_4^{(r)}) = ((K_4^{(8-r)})^{-1}, K_3^{(8-r)})^{-1}, K_2^{(8-r)})^{-1}, K_1^{(8-r)})^{-1})$ for r=2,3,4,5,6

$(Z_5^{(r)}, Z_6^{(r)}) = (K_5^{(7-r)} * K_6^{(7-r)}, (K_6^{(7-r)})^{-1\,\star})$ for r=1,2,....,6

where K^{-1} denote the multiplicative inverse (modulo 2^{16}+1) of K, i.e. , $K \otimes K^{-1}=1$, and $K^{-1\,\star}$ denote the polynomial multiplicative inverse modulo $x^{64}+x^4+x^3+x+1$ of K, i.e. ,

$K*K^{-1\star}=1$.

Thus the same device or subprogram can be used for both encryption and decryption. The extra cost is the computation for generating the key subblocks from 192-bit user-selected key.

To display why the decryption works, let's note first that the $\otimes$ layer controlled by subkey $(Z_1^{(1)},Z_2^{(1)},Z_3^{(1)},Z_4^{(1)})$ in the first round for decryption undoes the $\otimes$ layer controlled by subkey $(K_1^{(r+1)},K_2^{(r+1)}, K_3^{(r+1)},K_4^{(r+1)})$ in the output transformation for encryption, where *r* is the total number of rounds.

Then the Addition-Multiplication layer controlled by $(Z_5^{(1)},Z_6^{(1)})$ in the first round for decryption undoes the Addition-Multiplication layer controlled by $(K_5^{(r)},K_6^{(r)})$ in the last round for encryption , where *r* is the total number of rounds. i.e.

if, in encryption,

$U=AM(T,K_5^{(r)},K_6^{(r)})=(T+K_5^{(r)})*K_6^{(r)}$.

where T, a 64-bit block , is input data for Addition-Multiplication layer in encryption.

U is output data for Addition-Multiplication layer in encryption.

then, in decryption,

$T'=AM(U,Z_5^{(1)},Z_6^{(1)}))=AM(U, K_5^{(r)}*K_6^{(r)}, (K_6^{(r)})^{-1\star})$

$=(U+ K_5^{(r)}*K_6^{(r)})* (K_6^{(r)})^{-1\star}=((T+K_5^{(r)})*K_6^{(r)} + K_5^{(r)}*K_6^{(r)})* (K_6^{(r)})^{-1\star}$

$=(T*K_6^{(r)})* (K_6^{(r)})^{-1\star}=T$

(note that, all these operation is in the GF(2)[x] modulo an irreducible polynomial of degree 64 in GF(2)[x].)

where T' is the output data for Addition-Multiplication layer in decryption.

So, It can be seen that T', which is the output data for Addition-Multiplication layer in decryption, is equal to T, which is the output data for Addition-Multiplication layer in encryption.

Next, the permutation layer in the first round and the $\otimes$ layer controlled by subkey $(Z_1^{(2)},Z_2^{(2)},Z_3^{(2)},Z_4^{(2)})$ in the 2nd round for decryption undoes the $\otimes$ layer controlled by subkey $(K_1^{(r)},K_2^{(r)}, K_3^{(r)},K_4^{(r)})$ in the last round and the permutation in the round before the last round for encryption, where *r* is the total number of rounds.

In the same way, other decryption rounds undoes the transformation performed in corresponding encryption rounds. The $\otimes$ layer controlled by subkey $(Z_1^{(r)},Z_2^{(r)},Z_3^{(r)},Z_4^{(r)})$ in the output transformation for decryption undoes the $\otimes$ layer controlled by subkey $(K_1^{(1)},K_2^{(1)}, K_3^{(1)},K_4^{(1)})$ in the first round for encryption.

So that the original plaintext is recovered .

3. Design principles for this cipher

This block cipher is designed in accordance with Shannon's principles of confusion and diffusion for obtaining security in secret-key cipher.

The confusion required for a secure cipher is achieved in this block cipher by mixing three incompatible operation, i.e., $\otimes_4$,+,*.

The three operation are incompatible in the sense that:

1. $\otimes_4$ and + do not satisfies a distribution law. i.e.,

   there exist a, b, c, where a,b,c are 64-bit block , such that $a \otimes_4 (b+c)$ is not equal to $(a \otimes_4 b)+(a \otimes_4 c)$, and there exist a, b, c, where a,b,c are 64-bit block , such that $a+(b \otimes_4 c)$ is not

equal to $(a+b)\otimes_4(a+c)$,

2. $\otimes_4$ and * do not satisfies a distribution law. i.e.,

there exist a, b, c, where a,b,c are 64-bit block , such that $a\otimes_4(b*c)$ is not equal to $(a\otimes_4 b)*(a\otimes_4 c)$, and there exist a, b, c, where a,b,c are 64-bit block , such that $a*(b\otimes_4 c)$ is not equal to $(a*b)\otimes_4(a*c)$,

3 $\otimes_4$ and + do not satisfies a generalized associative law , i.e.,

there exist a, b, c, where a,b,c are 64-bit block , such that $a\otimes_4(b+c)$ is not equal to $(a\otimes_4 b)+c$

4 $\otimes_4$ and * do not satisfies a generalized associative law ,i.e.,

there exist a, b, c, where a,b,c are 64-bit block , such that $a\otimes_4(b*c)$ is not equal to $(a\otimes_4 b)*c$

5. $\otimes_4$ is an operation in the group $C_m \times C_m \times C_m \times C_m$, where $m=2^{16}$. + is an operation in the group $(C_2)^n$, where n=64. * is an operation in the group C_m, where $m=2^{64}-1$. (note, where C_j is a cyclic group of degree j). No pair of these 3 groups is isomorphism.

Thus, one can not change the order of operations arbitrarily to simplify analysis or replace one operation with the other by applying bijective mappings on the inputs and outputs.

A check by direct computation has shown that the round function is "complete" ,i.e., that each output bit of the round depends on every bit of the input bit of that round. This diffusion is provided in this block cipher by the transformation called the addition-multiplication structure.

The addition-multiplication structure transform one 64-bit block into another 64-bit block controlled by two 64-bit key blocks. This structure has the properties of "complete diffusion effect" in the sense that each output bit depends on every input bit .

In the other hand, because the "*" operation and the "$\otimes_4$" operation are highly nonlinear transformation with enough number of input bits and output bits, and the "*" operation make each output bit depending on every input bit, so the transformation combined by the "*" operation and the "$\otimes_4$" operation is "far from" linear transformation. This property increases the resistance to linear cryptanalysis[8].

The security of this cipher against differential cryptanalysis will be discussed in detail in next section.

As to the other kinds of attack, several tests have been conducted. Up to now, we have not found any weakness of this cipher.

4. The property against differential cryptanalysis

The differential cryptanalysis proposed by Biham and Shamir[4][5] showed that a lot of iterated block ciphers are theoretically cryptanalysable[6][7] . In their attack, they introduced a new notion which they called characteristic . Characteristic describe the behavior of input and output differences for some number of consecutive rounds. The probability of a one-round characteristic is the conditional probability that given a certain difference in the inputs to the round we get a certain difference in the outputs of the round. Lai and Massey[3] introduced a similar notion, which they called differential. The probability of an s-round differential is the condition probability that given an input difference at the first round, the output difference at the *s*-th round will be some fixed value. The probability of an s-round differential with input

difference A and output difference B is the sum of the probabilities of all s-round characteristics with input difference A and output difference B. For $s \leq 2$ the probabilities for a differential and for the corresponding characteristic are equal. Both the characteristic and the differential can equally be used for successful attack. The key to success on carrying out a differential attack in iterated block cipher is to use either a high characteristic probability or high differential probability. So to resist differential cryptanalysis, we should make no nontrivial *s*-round differential is useful. This goal can be achieved by reducing the maximum probability of nontrivial one-round characteristic of an iterated cipher.

Now, Let's turn to the cipher we proposed.

First, we will show this cipher is a Markov cipher, a fact that great simplifies its analysis for resistance to differential cryptanalysis.

In[1], Lai proved that, if the round function of an iterated cipher is in the form $f(X,Z)=g(X \odot Z_A,Z_B)$, where $\odot$ is a group operation, and $g(\cdot,Z_B)$ is invertible for any Z_B, then the iterated cipher is a Markov cipher where the definition of difference between a pair of plaintext is $\Delta X=X \odot (X')^{-1}$.

The round function of this iterated cipher is in the form $f(X,Z)=g(X \odot (K_1^{(i)},K_2^{(i)}, K_3^{(i)},K_4^{(i)}),(K_5^{(i)},K_6^{(i)}))$, where $\odot$ is $\otimes_4$, and $g(\cdot,(K_5^{(i)},K_6^{(i)}))$ is $Pe(AM(\cdot,K_5^{(r)},K_6^{(r)})) = Pe((\cdot +K_5^{(r)})*K_6^{(r)})$, and is invertible for any Z_B, so this iterated cipher is a Markov cipher where the definition of difference between a pair of plaintext is $\Delta X=X \otimes_4 (X')^{-1}$.

Next we will prove that the maximum probability of nontrivial one-round characteristic of

this iterated cipher reaches its possible minimum, i.e., $\frac{1}{2^{64}-1}$.

Theorem 1. Assume the round keys (i.e., K_1,K_2,K_3,K_4,K_5,K_6) of this cipher are independent and K_5 uniformly random in $GF(2^n)$, K_6 uniformly random in $GF(2^n)\backslash\{0\}$, then the maximum probability of nontrivial one-round characteristic of this iterated cipher is

$\max_{a\neq 0} \max_b P(\Delta Y = b|\Delta X = a) = \frac{1}{2^{64}-1}$, where the definition of difference between a pair of plaintext is $\Delta X = X \otimes_4 (X')^{-1}$ and 0 is the unit element of the corresponding group.

Proof. In this cipher,

$Y=Pe(((X \otimes_4 (K_1,K_2,K_3,K_4))+K_5)*K_6)$ and $Y'=Pe(((X' \otimes_4 (K_1,K_2,K_3,K_4))+K_5)*K_6)$

For any ,except the unit ,elements a and b in their corresponding group

$$P(\Delta Y = b|\Delta X = a) = \frac{P(\Delta Y = b, \Delta X = a)}{P(\Delta X = a)} \qquad (1)$$

But,

$$P(\Delta Y = b, \Delta X = a)$$

$$= \sum_{x\in GF(2^{64})} P(\Delta Y = b, X = x, X' = a^{-1} \otimes_4 x)$$

$$= \sum_{x\in GF(2^{64})} P(\Delta Y = b|X = x, X' = \bar{x}) P(X = x, X' = \bar{x}) \quad (\text{where } \bar{x} = a^{-1} \otimes_4 x)$$

$$= \sum_{x\in GF(2^{64})} P((((x \otimes_4 (K_1,K_2,K_3,K_4)) + K_5) * K_6) \otimes_4 (((\bar{x} \otimes_4 (K_1,K_2,K_3,K_4)) + K_5) * K_6)^{-1} = b) P(X = x, X' = \bar{x})$$

$$= \sum_{x \in GF(2^{64})} P(X = x, X' = \bar{x}) \sum_{y \in GF(2^{64})} P(((x \otimes_4 (K_1, K_2, K_3, K_4)) + K_5) * K_6 = y)$$
$$, (((\bar{x} \otimes_4 (K_1, K_2, K_3, K_4)) + K_5) * K_6) = \bar{y})$$

(where $\bar{y} = \mathrm{Pe}(b^{-1}) \otimes_4 y$)

$$= \sum_{x \in GF(2^{64})} P(X = x, X' = \bar{x}) \sum_{y \in GF(2^{64})} \sum_{k1 \in GF(2^{16})} \sum_{k2 \in GF(2^{16})} \sum_{k3 \in GF(2^{16})} \sum_{k4 \in GF(2^{16})} \cdot$$
$$P((((x \otimes_4 (k1, k2, k3, k4)) + K_5) * K_6) = y, (((\bar{x} \otimes_4 (k1, k2, k3, k4)) + K_5) * K_6) = \bar{y})$$
$$P(K_1 = k1) P(K_2 = k2) P(K_3 = k3) P(K_4 = k4)$$

$$= \sum_{x \in GF(2^{64})} P(X = x, X' = \bar{x}) \sum_{y \in GF(2^{64})} \sum_{k1 \in GF(2^{16})} \sum_{k2 \in GF(2^{16})} \sum_{k3 \in GF(2^{16})} \sum_{k4 \in GF(2^{16})} \cdot$$
$$P(\mathrm{K}_5 = \mathrm{k5}, \mathrm{K}_6 = \mathrm{k6})\ P(K_1 = k1) P(K_2 = k2) P(K_3 = k3) P(K_4 = k4)$$
(where $k5 = (\mathrm{y} + \bar{\mathrm{y}})^{-1^*} * (\underline{\mathrm{x}}\bar{\mathrm{y}} + \bar{\underline{\mathrm{x}}}\mathrm{y})$, $\mathrm{k6} = (\mathrm{y} + \bar{\mathrm{y}}) * (\underline{\mathrm{x}} + \bar{\underline{\mathrm{x}}})^{-1^*}$
$\underline{\mathrm{x}} = x \otimes_4 (k1, k2, k3, k4), \bar{\underline{\mathrm{x}}} = \bar{x} \otimes_4 (k1, k2, k3, k4),$)

$$= \sum_{x \in GF(2^{64})} P(X = x, X' = \bar{x}) \sum_{y \in GF(2^{64})} \sum_{k1 \in GF(2^{16})} \sum_{k2 \in GF(2^{16})} \sum_{k3 \in GF(2^{16})} \sum_{k4 \in GF(2^{16})} \cdot$$
$$\frac{1}{2^{64}(2^{64} - 1)} \quad P(K_1 = k1) P(K_2 = k2) P(K_3 = k3) P(K_4 = k4)$$
(because the round keys (i.e., $\mathrm{K}_1, \mathrm{K}_2, \mathrm{K}_3, \mathrm{K}_4, \mathrm{K}_5, \mathrm{K}_6$) of this cipher are
independent and K_5 uniformly random in GF(2^{64}) and K_6 uniformly random in GF(2^{64}) \ {0})

$$= \frac{1}{(2^{64} - 1)} \sum_{x \in GF(2^{64})} P(X = x, X' = \bar{x}) = \frac{1}{(2^{64} - 1)} P(\Delta x = a)$$

$$\text{So, } P(\Delta Y = b | \Delta X = a) = \frac{P(\Delta Y = b, \Delta X = a)}{P(\Delta X = a)} = \frac{1}{2^{64} - 1} = \text{constant}$$

Thus we get : $\max_{a \neq 0} \max_b P(\Delta Y = b | \Delta X = a) = \frac{1}{2^{64} - 1}$ □

By applying the theorem proved in [2], we can also prove that the maximum probability of one round differential of this block cipher is $\frac{1}{2^{64} - 1}$, regardless of the definition of difference between a pair of plaintext blocks (or a pair of ciphertext blocks).

For the evidence shown above , we can said that this block cipher is secure against differential cryptanalysis.

5. Conclusion

A new secret-key block-enciphering algorithm with an Addition-Multiplication Structure in Galois Field GF(2^{64}) in each round has been proposed. New cryptographic feature in this block-enciphering algorithm is the use of an Addition-Multiplication Structure in Galois Field GF(2^{64}) in each round, to achieve the desired "diffusion" of small changes in the plaintext or the key over the resulted ciphertext, and more importantly, to reduce the probability of 1-round differential. It has been proved that this secret-key block-enciphering algorithm is a Markov cipher and its maximum probability of 1-round differential is $1/(2^{64}-1)$, so it can resist differential cryptanalysis with few rounds.

In all of the statistical tests conducted up to now, we have not found any weakness of this cipher. Yet, of course, the security of this cipher needs further intensive investigation.

Acknowledgment

We would like to thank the anonymous referees for comments that improved the paper.

References

[1]X.Lai, On the Designed and security of block cipher, Ph.D. Thesis, Swiss Federal Institute of Technology, Zurich, 1992

[2]F.Zhu and B.Guo, A multiplication-addition structure against differential attack, To appear.

[3] X.Lai, J.massey, and S.Murphy Markov cipher and differential cryptanalysis. *Advances in Cryptology - CRYPTO'91.* Lecture Notes in Computer Science , Vol. 547 Springer-Verlag, Berlin,1992,pp.17-38.

[4] E.Biham and Shamir. Differential cryptanalysis of DES-like systems. *Journal of Cryptology,* Vol4,No.1,1991,pp.3-72.

[5]E.Biham and A.Shamir . Differential cryptanalysis of the full 16-round DES, *Advances in Cryptology: Proceeding of CRYPTO'92,* Springer-Verlag, Berlin, 1993, pp. 487-496.

[6]E.Biham and A.Shamir . Differential cryptanalysis of FEAL and N-Hash, *Advances in Cryptology: Proceeding of EUROCRYPT'91,* Springer-Verlag, Berlin, 1991, pp. 1-16

[7]E.Biham and A.Shamir . Differential cryptanalysis of Snefru, Khafre,REDOC-II, LOKI, and Lucifer. *Advances in Cryptology: Proceeding of CRYPTO'91,* 1992, pp. 156-171

[8]M.Matsui. Linear cryptanalysis method for DES-cipher. *Advances in Cryptology: Proceeding of EUROCRYPT'93,* Springer-Verlag, Berlin, 1994, pp. 386-397

DES-80*

Carlisle M. Adams
Entrust Technologies
750 Heron Road
Ottawa, Canada, K1V 1A7

1. Abstract

In the Fall of 1996, the Canadian Government issued a request for a study on the feasibility of strengthening the Data Encryption Standard (DES) by increasing the key length to 80 bits. What made this request both interesting and challenging was the overall constraint placed on the project: there was to be no change whatsoever made to the actual encryption / decryption algorithm; rather, changes were to be confined solely to the key scheduling algorithm. Any method may be used to input and process the (maximum) 80-bit primary key, but the result of the process must be sixteen 48-bit round keys suitable for keying the DES rounds in the standard manner.

This paper summarizes the results of the above study, including a new key scheduling algorithm that appears to satisfy all requirements of the DES-80 project. The full report (dated May 2, 1997) is the property of the Canadian Government.

2. Introduction

The Data Encryption Standard (DES) is perhaps the most widely known and widely used cryptographic algorithm in the world today. Since its introduction to the public in the mid-1970s, it has undergone intensive scrutiny by academics, government agencies, industry, and a host of would-be cryptanalysts (both the serious and the hobbyist). Two decades of such focused attention has convinced many that the algorithm itself is basically sound and that the principles used in its design have resulted in a cipher with intrinsic cryptographic strength.

Unfortunately, however, the DES algorithm as originally proposed has a keysize that is too small for some environments. Furthermore, with the rapid advances in computing performance, even on relatively low-priced desktop machines, it is clear that this keysize will become too small for many (or most) environments in the fairly near future. It appears that only two alternatives are possible if security is to be maintained: find a suitable DES replacement algorithm that has cryptographic strength commensurate with the predicted need for the next several years; or modify DES itself in order to increase its strength.

Although the first alternative is being pursued vigorously by many researchers (and has resulted in a number of candidate ciphers), it is recognized that the probability is relatively low that any other cipher will undergo the length and breadth of analysis that DES has so far undergone. Consequently, it may take a considerable length of time before any significant degree of confidence in the security of any given cipher is gained.

The second alternative is the ultimate goal of the DES-80 project. The intention is to leave the algorithm entirely as originally proposed, thereby drawing on the confidence built up over two decades of intense DES-related study. It is hoped that by altering the key schedule alone it may be possible to increase the key length of DES without weakening the cipher in any respect.

The remainder of this paper is organized as follows. Section 3 discusses key schedule design criteria for DES-like ciphers. Section 4 provides a number of candidate proposals for the DES-80 key schedule, and Section 5 evaluates these proposals with respect to the design criteria and the constraints of the DES-80 project. Section 6 describes a new key scheduling algorithm designed specifically for this project and compares it with the other candidates. The paper closes in Section 7 with some concluding remarks.

3. Key Schedule Design Criteria

Over the years a number of researchers have discussed and proposed design criteria for key scheduling algorithms in DES-like ciphers and/or have attempted to derive from the specification of DES the criteria that went

* The work described in this paper was funded by the Communications Security Establishment of the Canadian Government.