Wykład 4

Temat: Algorytm symetryczny Twofish: cele projektowane, budowa bloków, opis algorytmu, wydajność algorytmu.

W roku 1972 Narodowe Biuro Standardów (obecnie Narodowy Instytut Standardów i Technologii – NIST) ogłosiło konkurs na standard szyfrowania. Zwycięzcą okazał się DES, prawdopodobnie najszerzej stosowany algorytm kryptograficzny ostatnich 30 lat. Wbrew jego popularności algorytmowi ciągle towarzyszą kontrowersje. Od początku stosowania DES-a zastanawiano się czy w algorytmie nie umieszczono „tylnych drzwi" umożliwiających łatwiejsze odtwarzanie tekstu jawnego. Zadawano pytania o sposób tworzenia S-bloków. Bardzo duże poruszenie wywołała również decyzja o skróceniu klucza 128-bitowego stosowanego w algorytmie Lucyfer (w oparciu o ten algorytm tworzono DES) do 56 bitów. Obecnie klucz 56 bitowy okazuje się zbyt krótkim aby zapewnić wymagany poziom bezpieczeństwa dla nowoczesnych aplikacji. Jako tymczasowe rozwiązanie dla aplikacji wymagających wysokiego poziomu bezpieczeństwa (np. systemy bankowe) zaczęto stosować modyfikację algorytmu DES – potrójny DES, aczkolwiek tak zmodyfikowany algorytm okazuje się zbyt wolny dla wielu zastosowań. W odpowiedzi na rosnące potrzeby użytkowników – NIST w roku 1997 ogłasza konkurs na zaawansowany standard szyfrowania (AES). Nowy standard szyfrowania powinien według zamierzeń NIST zostać wyłoniony w procesie publicznej dyskusji. Organizacja ta narzuciła ogólne wymagania dla proponowanych algorytmów. Algorytm powinien mieć dłuższy klucz, większy rozmiar bloków, możliwość szybkiej pracy, w odniesieniu do algorytmu DES. Nowy standard szyfrowania musi zapewniać możliwość jego użytkowania przez następną dekadę, bez konieczności jego modyfikacji, czy też wymiany. Jednym ze zgłoszonych algorytmów był właśnie Twofish, spełniał on te podstawowe wymagania. Algorytm twofish pracuje z wykorzystaniem 128-

bitowych bloków; 128-, 192- i 256-bitowymi kluczami. Zapewnia możliwość implementacji na różnych platformach sprzętowych.

4.1. Cele projektowane

Twofish został zaprojektowany tak by spełnić wymagania stawiane przez NIST dla AES. Twofish spełnia następujące kryteria:

- jest 128-bitowym symetrycznym algorytmem blokowym;
- obsługuje klucze długości 128-, 192- oraz 256 bitów;
- nie ma kluczy słabych;
- zapewnia możliwość implementacji na różnych platformach sprzętowych i w różnych aplikacjach;
- elastyczne projektowanie: akceptuje dodatkowe długości klucza, i jest odpowiedni dla szyfrów strumieniowych, funkcji hashującej, funkcji MAC;
- proste projektowanie, aby ułatwić analizę algorytmu oraz jego implementację.

Dodatkowo projektanci narzucili następujące kryteria projektowe:

- akceptuje każdą długość klucza do 256 bitów;
- szyfruje dane w mniej niż 500 cyklach zegarowych na blok przy wykorzystaniu procesorów Intel Pentium, Pentium Pro oraz Pentium II dla w pełni zoptymalizowanej wersji algorytmu;
- jest zdolny do ustawiania 128-bitowych kluczy w czasie mniejszym niż czas wymagany do szyfrowania 32 bloków na Pentium, Pentium Pro i Pentium II;
- szyfruje dane w mniej niż 5000 cykli zegarowych na blok dla Pentium, Pentium Pro i Pentium II bez czasu ustawienia klucza;
- nie zawiera żadnych operacji nieskutecznych na 8-, 16-, 32- bitowych mikroprocesorach;

- nie zawiera żadnych operacji redukujących skuteczność na 64-bitowych mikroprocesorach;
- nie zawiera jakichkolwiek operacji nie wykonywalnych w wersjach sprzętowych;
- ma różne osiągi odnośnie planu klucza;
- szyfruje dane w mniej niż 10 milisekundach na 8-bitowym mikroprocesorze;
- może być implementowany na 8-bitowych mikroprocesorach z 64 bajtami RAM;
- może być implementowany przy użyciu mniej niż 20,000 bramek.

Cele kryptograficzne były następujące:

- 16-rundowy Twofish (z wybielaniem) powinien uniemożliwić atak ze znanym tekstem jawnym bez użycia 2^{80} tekstów jawnych i mniej niż w 2^{N} czasu, gdzie N jest długością klucza;
- 12-rundowy Twofish (z wybielaniem) powinien uniemożliwić atak ze znanym tekstem jawnym bez użycia 2^{64} tekstów jawnych i mniej niż w $2^{N/2}$ czasu.

Ostatecznie narzucono następujące cele projektowe:

- algorytm powinien posiadać różne warianty ze zmienną liczbą cykli;
- algorytm powinien zapewniać maksymalną możliwą szybkość, mieć możliwe najmniejsze wymagania co do pamięci. Dodatkowo powinien unikać konieczności stosowania np. dużych tablic w aplikacjach;
- powinien umożliwiać zastosowania dla funkcji hashującej, funkcji MAC, czy też dla generatora liczb pseudolosowych.

4.2. Budowa bloków w algorytmie Twofish

4.2.1. Sieci Feistela

Sieci Feistela są ogólną metodą przekształcania jakiejkolwiek funkcji (zwykle zwanej F funkcją) do postaci permutacji. Metoda ta została odkryta przez Horsta Feistela przy okazji projektowania algorytmu Lucyfer a spopularyzowana przez DES. Metoda ta jest wykorzystywana przez większość szyfrów blokowych np. Feal, GOST, Loki czy też Blowfish. Podstawą budowania bloków na podstawie sieci Feistela jest F funkcja tj. zależne od klucza przekształcenie ciągu wejściowego w ciąg wyjściowy. Funkcja F jest zawsze nieliniowa:

$$F : \{0,1\}^{n/2} \times \{0,1\}^{N} \Rightarrow \{0,1\}^{n/2}$$

gdzie n jest rozmiarem bloku sieci Feistela, a F jest funkcją która z n/2 bitów bloku oraz N bitów klucza wytwarza n/2 bitów na wyjściu. W każdym cyklu „źródłowy blok" jest wejściem dla funkcji F, która zwraca blok poddawany operacji XOR, oba bloki są zamieniane miejscami w następnym cyklu. Jeżeli funkcja F jest odpowiednio dobrana to powstanie mocny algorytm szyfrujący.

4.2.2. S-bloki

S-bloki są tablicą, nieliniową substytucją operacji używanych w większości algorytmów symetrycznych. Wartości S-bloków mogą być dobierane losowo lub drogą badań nad efektywnością (odpornością) danej kombinacji. S-bloki po raz pierwszy użyto w algorytmie Lucyfer, a ich wykorzystanie rozpowszechniło się po przez ich wykorzystanie w algorytmie DES. Twofish używa czterech różnych, zależnych od klucza S-bloków, które są budowane w oparciu o klucz wykorzystując operacje mieszania oraz permutacji.

4.2.3. Macierz MDS

Macierz maksymalnej odległości rozłącznej (MDS – Maximum Distance Separable) jest liniowym odwzorowaniem elementów zbioru a do zbioru elementów b, wytwarzając łączny wektor a+b elementów, z własnością że minimalna liczba niezerowych elementów w niezerowym wektorze wynosi co najmniej b+1. Inaczej odległość (tj. liczba elementów różniących się) w dwóch dowolnych, odmiennych wektorach wytworzonych przez odwzorowanie MDS wynosi przynajmniej b+1. Odwzorowanie MDS jest reprezentowane przez MDS matrycę składającą się z a x b elementów. Konieczny i wystarczający warunek dla a x b macierzy MDS jest taki, że wszystkie możliwe macierze kwadratowe uzyskane po przez odrzucenie wierszy lub kolumn macierzy pierwotnej nie są liczbami pojedynczymi. Twofish używa pojedynczej 4 na 4 macierzy MDS dla $GF(2^8)$ (GF – Galois Field – ciało Galois).

4.2.4. Przekształcenie PHT

Przekształcenie PHT (Pseudo – Hadamard Transform) jest prostą operacją mieszania, która jest szybko realizowana w programowej wersji algorytmu. Dane wejściowe a i b dla 32-bitowego PHT są definiowane następująco:

$$\left.\begin{array}{l} a' = a + b \bmod 2^{32} \\ b' = a + 2b \bmod 2^{32} \end{array}\right\}$$

Twofish używa 32-bitowego PHT w celu wytworzenia wartości końcowej dla dwóch, równoległych funkcji g (zatem mikroprocesor musi zapewnić równoległe przetwarzanie funkcji g).

4.2.5. Wybielanie

Wybielanie jest to technika polegająca na przetwarzaniu ciągu klucza przed pierwszym i po ostatnim cyklu algorytmu. Celem tej metody jest przeciwdziałanie kryptoanalizie badanego algorytmu na podstawie uzyskanej pary: tekstu jawnego i szyfrogramu. Metoda ta zmusza kryptoanalityka nie tylko do odgadywania klucza algorytmu, ale także jednej z wartości wybielających. Twofish poddaje operacji XOR 128 bitów klucza przed pierwszą rundą sieci Feistela (drugą wartością jest ciąg wejściowy) oraz 128 bitów po ostatniej rundzie sieci Feistela. Klucze tymczasowe poddawane operacji wybielania są tworzone w taki sam sposób jak klucze poszczególnych cykli sieci Feistela, należy zatem unikać stosowania tych kluczy gdziekolwiek indziej.

4.3. Opis algorytmu Twofish

Rysunek poniżej przedstawia algorytm Twofish. Używa on 16-rundowej sieci Feistela z dodatkowym wybielaniem na wejściu i na wyjściu algorytmu. Jedynie elementy nie wchodzące w skład sieci Feistela są poddawane 1-bitowej rotacji. Rotacje mogą być używane tylko dla F funkcji, aby zachować czystą strukturę sieci Feistela, ale takie postępowanie wymaga dodatkowej rotacji przed końcowym wybielaniem. Tekst jawny jest dzielony na cztery 32-bitowe słowa. Na wejściu (krok wstępny wybielania) słowa te są poddawane operacji XOR z czteroma słowami 32-bitowymi klucza, w każdym cyklu następuje ich modyfikacja.

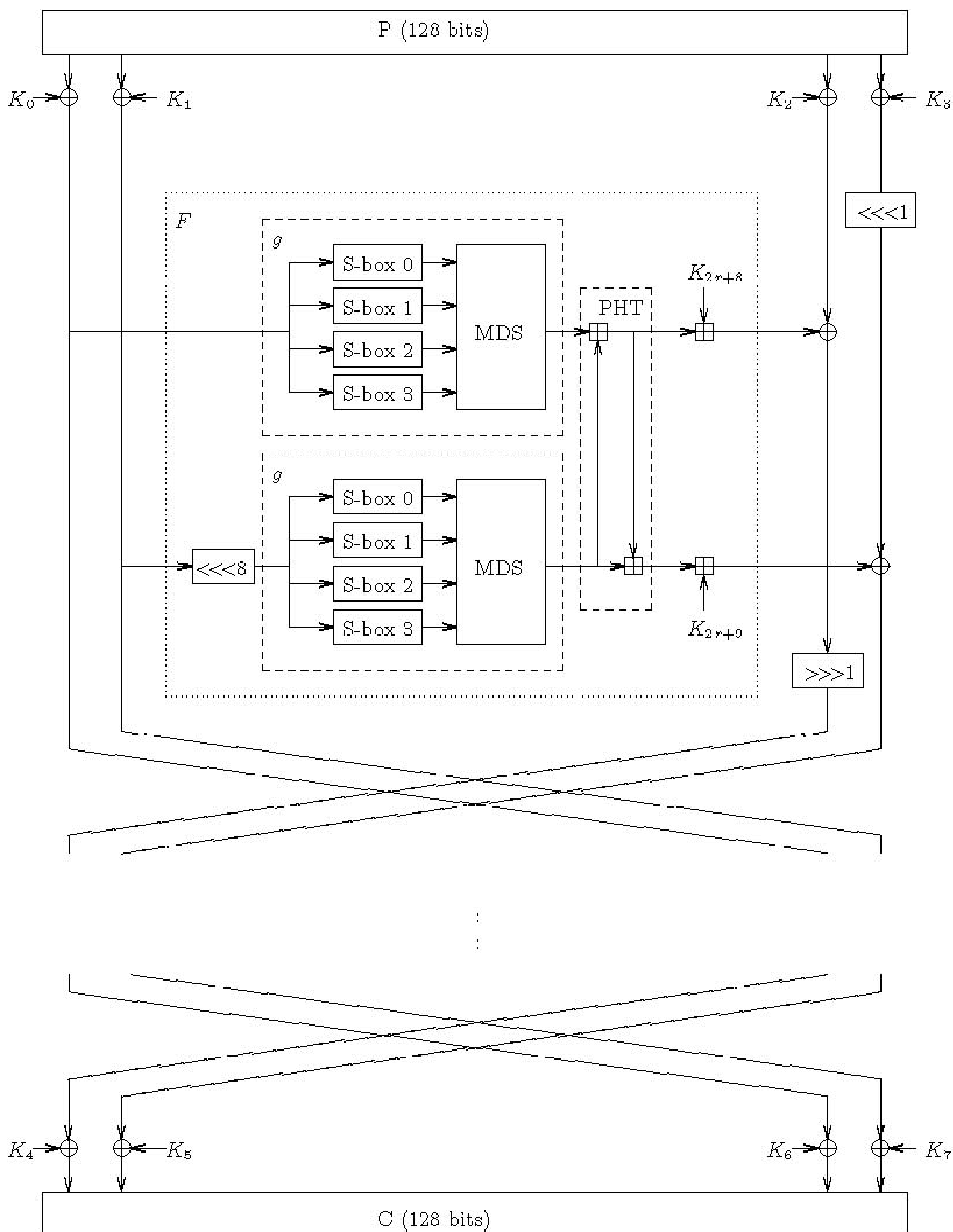
W każdej z rund, dwa słowa na lewo są używane jako wejście dla funkcji g (jedno z tych słów jest przesuwane o 8 bitów w lewo). Funkcja g składa się z czterech zależnych od klucza S-bloków liniowo mieszanych z matrycą MDS. Wynik dwóch kolejnych operacji g jest łączony za pomocą transformacji PHT (operacja XOR).

Następnie wyniki transformacji są wymieniane pomiędzy prawą i lewą stroną, wyniki te będą używane w kolejnym cyklu algorytmu. Po wszystkich rundach cztery wynikowe słowa są poddawane operacji XOR z czteroma słowami klucza tworząc szyfrogram.

Bardziej formalnie, 16 bajtów tekstu jawnego $p_0, \ldots, p_{15}$ jest dzielonych na cztery słowa 32 bitowe $P_0, \ldots, P_3$. Proces ten można zapisać następująco:

$$P_i = \sum_{j=1}^{3} p_{(4i+j)} \cdot 2^{8j}$$

gdzie i = 0,...3.



Rys. 4.1. Schemat blokowy algorytmu Twofish

Na początku wykonywania algorytmu wejściowe słowa (4 słowa 32-bitowe) są poddawane operacji XOR (wybielanie) z czteroma słowami będącymi rozszerzeniem klucza. Proces ten można zapisać następująco:

$$R_{0,i} = P_i \oplus K_i$$

gdzie i = 0,...3.

W każdej z 16 rund, pierwsze dwa słowa są używane jako wartości funkcji F, której parametrem jest również numer rundy. Trzecie słowo jest poddawane operacji XOR z wartością wyjściową funkcji F, a następnie otrzymana wartość jest przesuwana o jeden bit w prawo. Czwarte słowo jest najpierw przesuwane w lewo o jeden bit, a następnie poddawane operacji XOR z wartością wyjściową funkcji F. W ten sposób otrzymujemy dwie wartości, które są argumentami funkcji F w kolejnym cyklu (wartości są zamieniane – słowo otrzymane za pośrednictwem pierwszej funkcji g staje się wejściową wartością dla drugiej funkcji g i odwrotnie). Proces ten możemy przedstawić następująco:

$$(F_{r,0}, F_{r,1}) = F(R_{r,0}, R_{r,1}, r)$$

$$R_{r+1,0} = ROR(R_{r,2} \oplus F_{r,0}, 1)$$

$$R_{r+1,1} = ROL(R_{r,3}, 1) \oplus F_{r,1}$$

$$R_{r+1,2} = R_{r,0}$$

$$R_{r+1,3} = R_{r,1}$$

dla r = 0, … , 15.

Funkcje ROR i ROL są funkcjami, które przesuwają pierwszy argument (32-bitowe słowo) w lewo lub prawo o ilość bitów wskazaną drugim argumentem funkcji. Na końcu algorytmu następuje proces wybielania. Słowa wejściowe funkcji F dla ostatniej rundy, oraz wartości otrzymane po tej rundzie są poddawane operacji XOR z 4 słowami będącymi rozszerzeniem klucza. Proces ten można przedstawić następująco:

$$C_i = R_{16,\,(i+2) \bmod 4} \oplus K_{i+4}$$

gdzie i = 0,...3.

Cztery słowa szyfrogramu C_0, ... , C_3 są zapisywane w postaci 16 bajtów c_0, ... , c_{15} używając następującego przekształcenia:

$$c_i = \left[\frac{C_{(i/4)}}{2^{8(i \bmod 4)}} \right] \bmod 2^8$$

gdzie i = 0,...15.

4.3.1. Funkcja F

Funkcja F jest zależną od klucza permutacją na 64-bitowej wartości. Funkcja ta posiada trzy argumenty, wejściowe słowa R_0 i R_1 oraz liczbę r równą numerowi bieżącej rundy. Liczba r odpowiada za wybieranie podkluczy. R_0 jest wartością wejściową funkcji g, która zwraca wartość T_0. R_1 po przesunięciu o 8 bitów w lewo staje się wejściową wartością funkcji g, która zwraca wartość T_1. Wyniki T_0 i T_1 są następnie łączone przy pomocy przekształcenia PHT, do których dodaje się dwa słowa otrzymane z klucza. Przekształcenia te można zapisać następująco:

$$T_0 = g(R_0)$$
$$T_1 = g(ROL(R_1,8))$$
$$F_0 = (T_0 + T_1 + K_{2r+8}) \bmod 2^{32}$$
$$F_1 = (T_0 + 2T_1 + K_{2r+9}) \bmod 2^{32}$$

gdzie (F_0, F_1) jest wynikiem zwracanym przez funkcję F.

4.3.2. Funkcja g

Funkcja g jest najważniejszym elementem algorytmu Twofish. Wejściowe słowo X (32 bity) jest dzielone na cztery bajty. Każdemu bajtowi przyporządkowano zależny od klucza S-blok. Każdy S-blok przetwarza 8 bitów wejściowych na 8 bitów wyjściowych. Uzyskane cztery bajty wyjściowe są interpretowane jako wektor o długości 4 w ciele skończonym $GF(2^8)$. Te cztery bajty są następnie mnożone przez matrycę MDS – matryca 4x4 (używając ciała skończonego $GF(2^8)$ do obliczeń). Przebieg funkcji g można przedstawić następująco:

$$x_i = [X/2^{8i}] \bmod 2^8 \qquad i = 0, \ldots , 3$$
$$y_i = s_i[x_i] \qquad i = 0, \ldots , 3$$
$$Z = \sum_{i=0}^{3} z_i \cdot 2^{8i}$$

gdzie s_i jest zależnym od klucza S-blokiem, natomiast Z jest wynikiem zwracanym przez funkcję g. Należy zdefiniować zależność pomiędzy bajtem wartości i elementami $GF(2^8)$. Ciało skończone $GF(2^8)$ można przedstawić jako $GF(2)[x]/v(x)$, przy czym $v(x) = x^8 + x^6 + x^5 + x^3 + 1$ jest pierwotnym wielomianem stopnia 8 w GF(2).
Elementy ciała skończonego można zapisać następująco:

$$a = \sum_{i=0}^{7} a_i \cdot x^i \qquad \text{dla } a_i \in GF(2).$$

Elementy te (a_i) są naturalnym odwzorowaniem pojedynczego bajta w ciele skończonym GF, dodatkowo dla $GF(2^8)$ elementy te odpowiadają operacji XOR. Poniżej przedstawiono matrycę MDS, której wartości zostały podane w systemie szesnastkowym (dwa znaki odpowiadają jednemu bajtowi).

$$MDS = \begin{bmatrix} 01\ EF\ 5B\ 5B \\ 5B\ EF\ EF\ 01 \\ EF\ 5B\ 01\ EF \\ EF\ 01\ EF\ 5b \end{bmatrix}$$

4.3.3. Generowanie kluczy wewnętrznych

Mechanizm generowania kluczy wewnętrznych musi dostarczać 40 słów będących rozszerzeniem klucza użytkownika $K_0, \dots, K_{39}$ oraz 4 zależnych od klucza S-bloków używanych przez funkcje g. Twofish obsługuje klucze o długości N = 128, N = 192, N = 256 bitów. Algorytm współpracuje również z kluczami krótszymi niż 256 bitów, ale w takim przypadku ciąg klucza jest uzupełniany zerami do osiągnięcia jednej z długości wskazanych jako właściwe długości klucza. Zdefiniujmy wartość pomocniczą k = N/64. Klucz M składa się z 8k bajtów $m_0, \dots, m_{8k-1}$. Wejściowe bajty klucza są dzielone na 2k słów 32-bitowych.

$$M_i = \sum_{j=0}^{3} m_{(4i+j)} \cdot 2^{8j}$$

gdzie i = 0, ... , 2k-1.

Otrzymujemy dwa wektory o długości k (wektory złożone z słów 32-bitowych):

$$M_e = (M_0, M_2, \dots, M_{2k-2})$$

$$M_o = (M_1, M_3, \dots, M_{2k-1})$$

Trzeci wektor o długości k otrzymujemy również z klucza. Jest on tworzony po przez grupowanie bajtów klucza po 8, a następnie mnożony przez macierz 4x8 zwaną macierzą RS. Każde 4 kolejne bajty wynikowe są interpretowane jako 32-bitowe słowa. Proces ten przedstawiono poniżej:

$$\begin{bmatrix} s_{i,0} \\ s_{i,1} \\ s_{i,2} \\ s_{i,3} \end{bmatrix} = \begin{bmatrix} & & RS & & \end{bmatrix} \cdot \begin{bmatrix} m_{8i} \\ m_{8i+1} \\ m_{8i+2} \\ m_{8i+3} \\ m_{8i+4} \\ m_{8i+5} \\ m_{8i+6} \\ m_{8i+7} \end{bmatrix}$$

$$S_i = \sum_{j=0}^{3} s_{i,j} \cdot 2^{8j}$$

dla i = 0, ..., k-1 oraz dla $S = (S_{k-1}, S_{k-2}, \dots, S_0)$.

Można zauważyć że wartości S są podawane w odwrotnej kolejności. Stosuje się dodatkową macierz mnożącą RS, ciało skończone $GF(2^8)$ jest reprezentowane przez $GF(2)[x]/w(x)$, gdzie $w(x) = x^8 + x^6 + x^3 + x^2 + 1$ jest innym pierwotnym wielomianem stopnia 8 w GF(2). Odwzorowanie pomiędzy bajtem wartości, a elementami $GF(2^8)$ jest takie same jak w przypadku macierzy MDS. Macierz RS można przedstawić następująco:

$$RS = \begin{bmatrix} 01\ A4\ 55\ 87\ 5A\ 58\ DB\ 9E \\ A4\ 56\ 82\ F3\ 1E\ C6\ 68\ E5 \\ 02\ A1\ FC\ C1\ 47\ AE\ 3D\ 19 \\ A4\ 55\ 87\ 5A\ 58\ DB\ 9E\ 03 \end{bmatrix}$$

Trzy wektory M_e, M_o oraz S są podstawowymi, najważniejszymi elementami mechanizmu generowania kluczy wewnętrznych.

Dodatkowe długości klucza

Algorytm Twofish akceptuje klucze o długości poniżej 256 bitów. Dla kluczy o długości innej niż długości zdefiniowane (128, 192 i 256 bitów) ciąg klucza jest uzupełniany zerami do osiągnięcia najbliższej zdefiniowanej długości. Dla przykładu klucz 80 bitowy $m_0, \ldots, m_9$ jest rozszerzany po przez przypisanie $m_i = 0$ dla $i = 10, \ldots, 15$. W ten sposób uzyskuje się wcześniej zdefiniowane klucze, tj. w tym przypadku klucz 128-bitowy.

Funkcja h

Rysunek poniżej przedstawia funkcję h. Funkcja ta posiada dwie wartości wejściowe – 32-bitowe słowo X i wektor $L = (L_0, \ldots, L_{k-1})$ słów 32-bitowych o długości k, natomiast funkcja zwraca jedno słowo 32-bitowe. Funkcja ta pracuje w k cyklach. W każdym cyklu, cztery bajty przechodzą przez S-blok, a następnie są poddawane operacji XOR z elementem wektora L. Na końcu, bajty przechodzą jeszcze raz przez S-bloki i cztery końcowe bajty są mnożone przez macierz MDS. Bardziej formalnie można proces ten zapisać:

$$l_{i,j} = [L_i/2^{8j}] \bmod 2^8$$

$$x_j = [X/2^{8j}] \bmod 2^8$$

dla i = 0,..., k-1 oraz j = 0,...,3. Wtedy kolejność operacji jest następująca:

$$y_{k,j} = x_j \quad \text{dla } j = 0,\ldots,3.$$

1. Jeśli k = 4 wtedy:

$$y_{3,0} = q_1[y_{4,0}] \oplus l_{3,0}$$

$$y_{3,1} = q_0[y_{4,1}] \oplus l_{3,1}$$

$$y_{3,2} = q_0[y_{4,2}] \oplus l_{3,2}$$

$$y_{3,3} = q_1[y_{4,3}] \oplus l_{3,3}$$

2. Jeśli k ≥ 3 wtedy:

$$y_{2,0} = q_1[y_{3,0}] \oplus l_{2,0}$$

$$y_{2,1} = q_1[y_{3,1}] \oplus l_{2,1}$$

$$y_{2,2} = q_0[y_{3,2}] \oplus l_{2,2}$$

$$y_{2,3} = q_0[y_{3,3}] \oplus l_{2,3}$$

3. W pozostałych przypadkach:

$$y_0 = q_1[q_0[q_0[y_{2,0}] \oplus l_{1,0}] \oplus l_{0,1}]$$

$$y_1 = q_0[q_0[q_1[y_{2,1}] \oplus l_{1,1}] \oplus l_{0,2}]$$

$$y_2 = q_1[q_1[q_0[y_{2,2}] \oplus l_{1,2}] \oplus l_{0,3}]$$

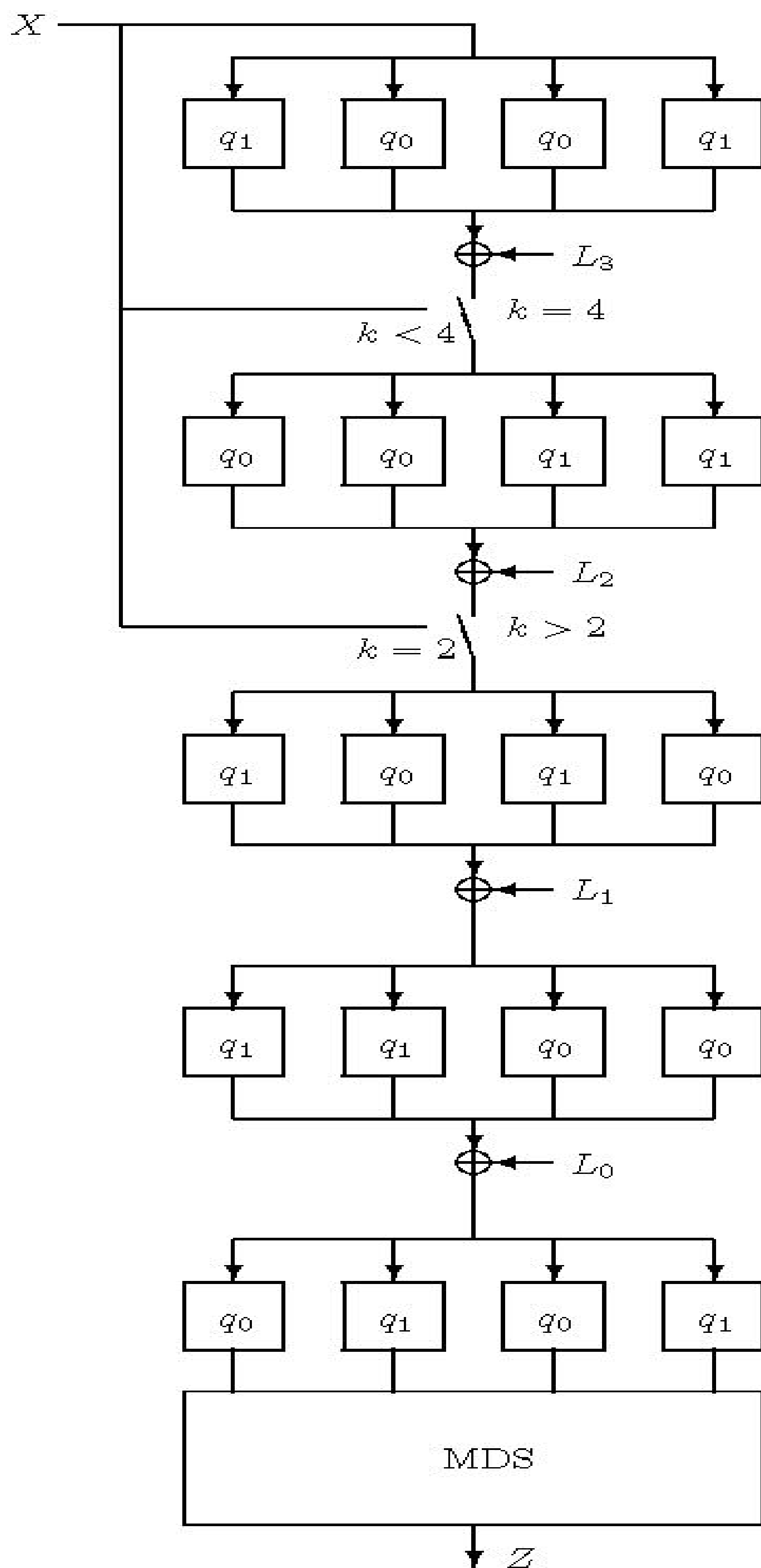
$$y_3 = q_0[q_1[q_1[y_{2,3}] \oplus l_{1,3}] \oplus l_{0,4}]$$

W podanych zależnościach q_0, $q_1$oznaczają permutację na 8-bitowych wartościach i zastaną przedstawione w następnych częściach pracy. Wynikowy wektor y_i jest mnożony przez macierz MDS.

$$\begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} & \text{MDS} & \end{bmatrix} \cdot \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

$$Z = \sum_{i=0}^{3} z_i \cdot 2^{8i}$$

gdzie Z jest wartością zwracaną przez funkcję h.



Rys. 4.2. Funkcja h

S-bloki

Można zdefiniować S-bloki w funkcji g:

$$g(X) = h(X, S)$$

gdzie i = 0,…,3.

Zależne od klucza S-bloki s_i są tworzone po przez odwzorowanie x_i do y_i w funkcji h, gdzie wektor L jest równy wektorowi S zależnemu od klucza.

Rozszerzanie klucza K_j

Rozszerzanie klucza K_j jest definiowane z użyciem funkcji h. Proces rozszerzania klucza można zapisać w następujący sposób:

$$\rho = 2^{24} + 2^{16} + 2^{8} + 2^{0}$$

$$A_i = h(2i\rho, M_e)$$

$$B_i = ROL(h((2i+1)\rho, M_o), 8)$$

$$K_{2i} = (A_i + B_i) \bmod 2^{32}$$

$$K_{2i+1} = ROL((A_i + 2B_i) \bmod 2^{32}, 9)$$

Stała ρ jest użyta w celu podwojenia bajtów; i ma taką właściwość że dla i = 0, ... , 255 słowo iρ składa się z czterech równych bajtów, o wartości równej i każdy. Funkcja h jest właśnie stosowana w odniesieniu do takich słów. Dla A_i bajtem wartości jest 2i, a drugim argumentem funkcji h jest M_e. W przypadku B_i jest podobnie, z tym że bajtem wartości jest 2i+1, natomiast drugim argumentem funkcji h jest M_o. Dodatkowo uzyskana wartość jest przesuwana o 8 bitów. Wartości A_i i B_i są łączone przy pomocy transformacji PHT. Dodatkowo jeden z wyników jest przesuwany o 9 bitów. Dwa wyniki tworzą natomiast dwa słowa będące rozszerzeniem klucza.

Permutacje q_0 i q_1

Permutacje q_0 i q_1 są operacjami na 8-bitowych wartościach. Są one budowane z czterech różnych 4-bitowych permutacji każda. Wartością wejściową jest x, wartość wyjściową y można zdefiniować następująco:

$$a_0, b_0 = [x/16], x \bmod 16$$

$$a_1 = a_0 \oplus b_0$$

$$b_1 = a_0 \oplus ROR_4(b_0, 1) \oplus 8a_0 \bmod 16$$

$$a_2, b_2 = t_0[a_1], t_1[b_1]$$

$$a_3 = a_2 \oplus b_2$$

$$b_3 = a_2 \oplus ROR_4(b_2, 1) \oplus 8a_2 \bmod 16$$

$$a_4, b_4 = t_2[a_3], t_3[b_3]$$

$$y = 16b_4 + a_4$$

Gdzie ROR_4 jest funkcją dokonującą przesunięcia o 4 bity. Każda z wartości wejściowych (po wstępnych przekształceniach) przechodzi przez 4-bitowe S-bloki. Następnie następuje proces łączenia tak ukształtowanych części w bajt wynikowy. Dla permutacji q_0 4-bitowe S-bloki przedstawiają się następująco:

$$t_0 = [\ 8\ 1\ 7\ D\ 6\ F\ 3\ 2\ 0\ B\ 5\ 9\ E\ C\ 4\ ]$$

$$t_1 = [\ E\ C\ B\ 8\ 1\ 2\ 3\ 5\ F\ 4\ 6\ 7\ 0\ 9\ D\ ]$$

$$t_2 = [\ B\ 5\ E\ 6\ D\ 9\ 0\ C\ 8\ F\ 3\ 2\ 4\ 7\ 1\ ]$$

$$t_3 = [\ D\ 7\ F\ 4\ 1\ 2\ 6\ E\ 9\ B\ 3\ 0\ 8\ 5\ C\ ]$$

gdzie każdy 4-bitowy S-blok jest reprezentowany listą zapisaną w notacji szesnastkowej. Podobnie można przedstawić 4-bitowe S-bloki dla permutacji q_1:

$$t_0 = [\ 2\ 8\ B\ D\ F\ 7\ 6\ E\ 3\ 1\ 9\ 4\ 0\ C\ 5\ ]$$

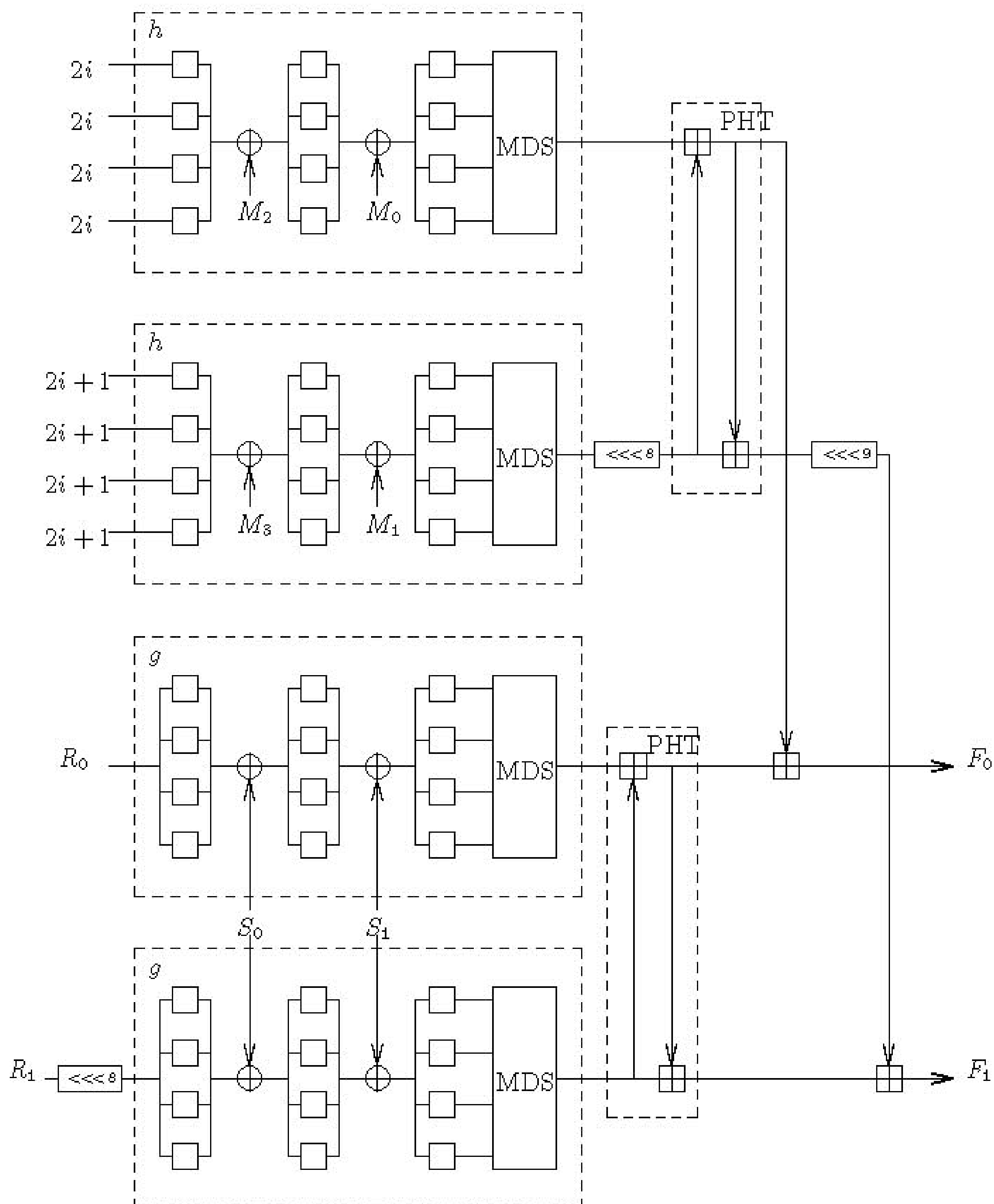
$$t_1 = [\ 1\ E\ 2\ B\ 4\ C\ 3\ 7\ 6\ D\ 5\ F\ 9\ 0\ 8\ ]$$

$$t_2 = [\ 4\ C\ 7\ 5\ 1\ 6\ 9\ 0\ E\ D\ 8\ 2\ B\ 3\ F\ ]$$

$$t_3 = [\ B\ 9\ 5\ 1\ C\ 3\ D\ E\ 6\ 4\ 7\ F\ 2\ 0\ 8\ ]$$

4.3.4 Cykle algorytmu

Rysunek poniżej przedstawia bardziej szczegółowo przebieg funkcji F w każdej z rund dla klucza o długości 128 bitów.



Rys. 4.3. Pojedynczy cykl funkcji F

4.4. Wydajność algorytmu Twofish

Twofish został zaprojektowany tak, by maksymalizować efektywność jego stosowania. Projektowany był tak, by móc go implementować na różnych platformach: 32-bitowym CPU, 8-bitowej karcie inteligentnej czy też dedykowanym sprzęcie VLSI. Jednak bardziej znamienne jest to że Twofish został zaprojektowany tak by wyważyć wpływ różnych czynników, często stojących w sprzeczności ze sobą na efektywną wydajność (np. szybkość szyfrowania, ustawianie klucza, wymagana pamięć, liczba potrzebnych bramek itd.). Wynikiem jest algorytm który może być wykorzystywany w różnych aplikacjach kryptograficznych, nie zależnych od siebie – aczkolwiek mogących ze sobą współpracować.

4.4.1 Wydajność na mikroprocesorach

Tabela poniżej przedstawia wydajność algorytmu Twofish dla różnych wariantów kluczy wewnętrznych, przy wykorzystaniu różnych języków programowania i różnych kompilatorów. Czasy szyfrowania i deszyfrowania są zwykle zbliżone lub nawet takie same zatem podawany jest tylko czas szyfrowania. Całkowity czas potrzebny na wykonanie algorytmu jest podawany w cyklach zegarowych na blok lub w cyklach zegarowych na bajt danych. Znacznie większe znaczenie ma sposób tworzenia klucza. Istnieją cztery różne sposoby generowania klucza, które zostaną przedstawione poniżej. W wszystkich opisywanych opcjach tworzenia klucza K_i dla i = 0, ... , 39 używa się 160 bajtów pamięci RAM. Różnice natomiast pojawiają się w sposobie realizacji funkcji g.

1. ***Klucz pełny*** – Ta opcja wykonuje wszystkie czynności związane z tworzeniem klucza. Używa 4 Kb tablicy, S-bloki są realizowane w postaci 8 x 32-bitowych tablic. Używając tej opcji funkcja g tworzy cztery pomocnicze tablice i wykonuje trzy operacje XOR. Szybkości szyfrowania i deszyfrowania są stałe i niezależne od długości klucza.
2. ***Klucz częściowy*** – Ta opcja jest użyteczna w sytuacji szyfrowania kilku bloków z użyciem tego samego klucza. W takiej sytuacji nie ma sensu przeprowadzania pełnego procesu tworzenia klucza. W tej opcji tworzone są cztery S-bloki, w postaci tablic 8 x 8 bitów. Opcja ta również używa czterech tablic MDS 8 x 32 bity. Redukcji ulega rozmiar tablicy do 1Kb. Dla każdego bajta ostatni q-blok jest w rzeczywistości realizowany przez macierz MDS, jedynie k q-bloków jest realizowanych w postaci 8 x 8 bitowych tablic S-blokowych. Szybkości szyfrowania i deszyfrowania są stałe i niezależne od długości klucza.
3. ***Klucz minimalny*** – Istnieje możliwość dalszej optymalizacji procesu generowania klucza, w przypadku szyfrowania kilku bloków przy użyciu tego samego klucza. W porównaniu z kluczem częściowym, jedna warstwa q-bloków jest realizowana z wykorzystaniem S-bloków, natomiast pozostałe są tworzone podczas procesu szyfrowania. Dla klucza o długości 128 bitów opcja ta jest szczególnie efektywna (optymalne tworzenie S-bloków). Ta opcja używa 1Kb tablicy do przechowywania pierwotnych S-bloków.
4. ***Klucz zerowy*** – Opcja ta nie wymaga predefiniowanych S-bloków i żadnych dodatkowych tablic. Wymaga jednak w procesie tworzenie klucza dodatkowego czasu na ustawienie wartości K_i oraz S.
5. ***Klucz kompilacyjny*** – W tej opcji, dostępnej tylko w wersjach asemblerowych algorytmu, podklucze są umieszczane jako stałe w kodzie źródłowym. Wymagany jest dodatkowy czas na zainicjowanie stałych oraz dodatkowych 5000 bajtów pamięci, ale opcja ta w wielu sytuacjach okazuje się najbardziej efektywna.

Języki programowania, kompilatory oraz wybór procesora

Wybór języka programowania oraz kompilatora mogą mieć bardzo duży wpływ na wydajność algorytmu. Jako podstawowy kompilator (punkt odniesienia) został wybrany kompilator Borland C 5.0 – aczkolwiek okazuje się że nie jest on optymalny. Przykładowo kompilator Microsoft Visual C++ 4.2 generuje kod algorytmu Twofish 20% szybciej niż Borland C 5.0, również porównanie szybkości optymalizacji kodu wypada na korzyść kompilatora Microsoft Visual C++ 4.2. Różnice te wynikają z niezdolności kompilatora Borland do generacji wewnętrznych operacji przesunięcia. Ogólnie można przyjąć że niezależnie jakim sprzętem dysponujemy kompilator Borland C 5.0 okazuje się wolniejszy i mniej efektywny od kompilatora Microsoft Visual C++ 4.2. Znaczenie dla efektywności algorytmu ma również wybór procesora. Drastycznie różne wyniki uzyskuje się stosując procesor Pentium w porównaniu z procesorami Pentium II czy też Pentium Pro. Wynika to z ilości jednostek arytmetyczno – logicznych (ALU) czy też z szybkości dostępu do pamięci.

Procesor	Język	Opcja klucza	Rozmiar kodu	Klucz			Szyfrowanie		
				128	192	256	128	192	256
PentiumPro/II	Asembler	kompilacyjny	8900	12700	15400	18100	285	285	285
PentiumPro/II	Asembler	Pełny	8450	7800	10700	13500	315	315	315
PentiumPro/II	Asembler	Częściowy	10700	4900	7600	10500	460	460	460
PentiumPro/II	Asembler	Minimalny	13600	2400	5300	8200	720	720	720
PentiumPro/II	Asembler	Zerowy	9100	1250	1600	2000	860	1130	1420
PentiumPro/II	MS C	Pełny	11200	8000	11200	15700	600	600	600
PentiumPro/II	MS C	Częściowy	13200	7100	9700	14100	800	800	800
PentiumPro/II	MS C	Minimalny	16600	3000	7800	12200	1130	1130	1130
PentiumPro/II	MS C	Zerowy	10500	2450	3200	4000	1310	1750	2200
PentiumPro/II	Borland C	Pełny	14100	10300	13600	18800	640	640	640
PentiumPro/II	Borland C	Częściowy	14300	9500	11200	16600	840	840	840
PentiumPro/II	Borland C	Minimalny	17300	4600	10300	15300	1160	1160	1160
PentiumPro/II	Borland C	Zerowy	10100	3200	4200	4800	1910	2670	3470
Pentium	Asembler	kompilacyjny	8900	24600	26800	28800	290	290	290
Pentium	Asembler	Pełny	8200	11300	14100	16000	315	315	315
Pentium	Asembler	Częściowy	10300	5500	7800	9800	430	430	430
Pentium	Asembler	Minimalny	12600	3700	5900	7900	740	740	740
Pentium	Asembler	Zerowy	8700	1800	2100	2600	1000	1300	1600
Pentium	MS C	Pełny	11800	11900	15100	21500	630	630	630
Pentium	MS C	Częściowy	14100	9200	13400	19800	900	900	900
Pentium	MS C	Minimalny	17800	3800	11100	16900	1460	1460	1460
Pentium	MS C	Zerowy	11300	2800	3900	4900	1740	2260	2760
Pentium	Borland C	Pełny	12700	14200	18100	26100	870	870	870
Pentium	Borland C	Częściowy	14200	11200	16500	24100	1100	1100	1100
Pentium	Borland C	Minimalny	17500	4700	12100	19200	1860	1860	1860
Pentium	Borland C	Zerowy	11800	3700	4900	6100	2150	2730	3270
UltraSPARC	C	Pełny		16600	21600	24900	750	750	750
UltraSPARC	C	Częściowy		8300	13300	19900	930	930	930
UltraSPARC	C	Minimalny		3300	11600	16600	1200	1200	1200
UltraSPARC	C	Zerowy		1700	3300	5000	1450	1680	1870
PowerPC 750	C	Pełny		12200	17100	22200	590	590	590
PowerPC 750	C	Częściowy		7800	12200	17300	780	780	780
PowerPC 750	C	Minimalny		2900	9100	14200	1280	1280	1280
PowerPC 750	C	Zerowy		2500	3600	4900	1030	1580	2040
68040	C	Pełny	16700	53000	63500	96700	3500	3500	3500
68040	C	Częściowy	18100	36700	47500	78500	4900	4900	4900
68040	C	Minimalny	23300	11000	40000	71800	8150	8150	8150
68040	C	Zerowy	16200	9800	13300	17000	6800	8600	10400

Tab. 4.1. Wydajność algorytmu Twofish dla różnych długości klucza i różnych opcji

Rozmiar kodu źródłowego

Rozmiar kodu źródłowego (w bajtach) zaprezentowano w tabeli 4.5.1. Rozmiar kodu dla w pełni zoptymalizowanej wersji algorytmu Twofish przy wykorzystaniu procesora Pentium Pro wynosi od 8450 bajtów dla asemblera do 14100 bajtów dla Borland C. Na rozmiar kodu źródłowego wpływają macierz MDS (4600 bajtów) oraz permutacje q_0 i q_1 które wymagają dodatkowych tablic (4300 bajtów dla każdego klucza – pełna opcja generacji klucza).

Całkowity czas szyfrowania

Aby wyznaczyć całkowity czas potrzebny na przeprowadzenie procesu szyfrowania trzeba zsumować czas potrzebny na generację klucza oraz czas potrzebny na właściwy proces szyfrowania. Dla dużych wiadomości czas potrzebny na generację klucza można pomijać, ale już dla krótkich wiadomości może się okazać że czas potrzebny na ustawienie klucza przekracza czas właściwego szyfrowania. Tabela poniżej przedstawia wydajność algorytmu Twofish w wersji asemblerowej z wykorzystaniem procesora Pentium Pro dla klucza o długości 128 bitów dla różnych długości wiadomości.

Tekst jawny [bajty]	**Opcja klucza**	**Cykle klucza**	**Cykle szyfrowania**	**Liczba cykli na bajt danych**
16	Zerowy	1250	860	131.9
32	Zerowy	1250	1720	92.8
64	Zerowy	1250	4690	73.3
128	Zerowy	1250	6880	63.5
256	Częściowy	4900	7360	47.9
512	Pełny	7800	10080	34.9
1K	Pełny	7800	20160	27.3
2K	Pełny	7800	40320	23.5
4K	Kompilacyjny	12700	72960	20.9
8K	Kompilacyjny	12700	145920	19.4
16K	Kompilacyjny	12700	291840	18.6
32K	Kompilacyjny	12700	583680	18.2
64K	Kompilacyjny	12700	1167360	18.0
1M	Kompilacyjny	12700	18677760	17.8

Tab. 4.2 Szybkość szyfrowania Twofish z 128-bitowym kluczem na procesorze Pentium Pro

4.4.2 Wydajność dla kart inteligentnych

Wydajność algorytmu Twofish dla kart inteligentnych zostanie omówiona dla karty z procesorem 6805 CPU. Wyniki pomiarów przedstawiają się następująco:

RAM [bajty]	Rozmiar kodu	Liczba cykli na blok	Czas szyfrowania bloku
60	2200	26500	6.6 msec
60	2150	32900	8.2 msec
60	2000	35000	8.7 msec
60	1760	37100	9.3 msec

Tab. 4.3. Wydajność karty inteligentnej z procesorem 6805 C

Czasy szyfrowania i deszyfrowania są identyczne. Jeśli wymagane jest tylko szyfrowanie to kod źródłowy będzie oczywiście mniejszy. Czas generowania klucza jest czasem potrzebnym na odwzorowanie Reed – Solomon, używane do generowania S-bloków i nieznacznie przekracza 1750 cykli na klucz. Ten czas można ograniczyć po przez wprowadzenie dodatkowych dwóch 512-bajtowych pamięci ROM. Rozmiary i ocena szybkości zostaną przeprowadzone dla implementacji algorytmu Twofish w karcie inteligentnej z kluczem 128-bitowym. Dla algorytmów z kluczem 192-bitowym rozmiar kodu wzrasta o około 100 bajtów, a dla algorytmu z kluczem 256-bitowym rozmiar kodu wzrasta o około 200 bajtów. Czas szyfrowania bloku wzrasta dla algorytmu z kluczem 192-bitowym o 2600 cyklów, natomiast dla algorytmu z kluczem 256-bitowym o 5200 cykli zegarowych. Podobnie sytuacja wygląda z czasem ustawiania klucza. Dla algorytmu z kluczem 192-bitowym liczba potrzebnych cykli wzrasta o 2550, natomiast dla klucza 256-bitowego o 3400. Wymagana pamięć RAM dla takiej karty to co najmniej 60 bajtów, na blok tekstu jawnego lub szyfrogramu przeznaczonych jest 16 bajtów, kolejnych 16 bajtów jest przeznaczonych na klucz. Jeśli 16 bajtów klucza i 8 dodatkowych bajtów przeznaczonych na wynik odwzorowania Reed -Solomon są zachowywane w pamięci między operacjami szyfrowania, pozyskujemy łącznie 36 bajtów dostępnych dla innych operacji. Oczywiście algorytmy z dłuższymi kluczami wymagają większych pamięci.

4.4.3 Wydajność sprzętowa

Istnieje wiele możliwości i alternatyw implementacji sprzętowej algorytmu. Przykładowo klucze używane w cyklach mogą być wcześniej wytworzone i przechowywane w pamięci RAM lub generowane dopiero w czasie szyfrowania. Jeżeli projektant zdecyduje się na drugie rozwiązanie to realizacja funkcji h może być różna. Można funkcje h wykonywać pomiędzy operacjami generacji klucza a funkcją rund, co z kolei pociągnie za sobą zmniejszenie szybkości przetwarzania, lub zdecydować się na dodanie bramek realizujących tą funkcję. Takie rozwiązanie powoduje utrzymanie dużej szybkości przetwarzania kosztem podwojenia liczby używanych bramek. Jeśli klucze byłyby wytwarzane przed szyfrowaniem to funkcja h może być przetwarzana w czasie ustawiania klucza, ograniczamy wtedy liczbę bramek kosztem czasu przetwarzania (tracimy w przybliżeniu czas potrzebny na szyfrowanie jednego bloku). Inny problem to sposób realizacji S-bloków. Mogą być one tworzone przed operacją szyfrowania i przechowywane w dodatkowej pamięci RAM (osiem 256-bajtowych pamięci RAM) przyśpieszając sam proces szyfrowania. Oczywistą wadą takiego rozwiązania jest konieczność inicjalizacji klucza po przez pamięć RAM, ale z drugiej strony poprawna implementacja takiego typu może zwiększyć przepustowość nawet dwukrotnie (szczególnie dla dłuższych kluczy). Rozwiązanie polegające na tworzeniu 8-bitowych permutacji q_0 i q_1 na podstawie czterech 4-bitowych permutacji wybrane zostało po to by zmniejszyć liczbę koniecznych bramek w implementacjach sprzętowych.

Permutacje mogą być realizowane w postaci bramek lub za pomocą 256-bajtowej pamięci ROM. Każdy pełny blok funkcji h wykorzystuje po sześć bloków q_0 i q_1 (dla N=128). Opóźnienie wynikające z konieczności realizacji bloków q_0 i q_1 przy wykorzystaniu bramek logicznych i pamięci ROM są porównywalne. W trakcie projektowania należy zwrócić uwagę na to że indywidualnie niektóre elementy (np. MDS, PHT, operacje XOR) wykazują się dużą szybkością mimo to niekoniecznie tak musi być gdy ze sobą współpracują. W trybie ECB można zwiększyć uzyskiwaną przepustowość po przez przetwarzanie potokowe. Można zaprojektować nawet wielopoziomowe przetwarzanie potokowe aby zwiększyć uzyskiwaną przepustowość. Jest to szczególnie użyteczne ponieważ nie wymaga jakiegoś drastycznego wzrostu ilości wymaganych bramek. Tabela poniżej przedstawia wymagania sprzętowe i uzyskiwane szybkości przy implementacji algorytmu Twofish z kluczem 128-bitowym. Dla dłuższych kluczy liczba bramek będzie wzrastać.

Ilość bramek	**Ilość bloków h**	**Ilość cykli na blok**	**Ilość poziomów**	**Zegar [MHz]**	**Przepustowość [Mb/s]**	**Uwagi**
14000	1	64	1	40	80	*
19000	1	32	1	40	160	
23000	2	16	1	40	320	
26000	2	32	2	80	640	
28000	2	48	3	120	960	
30000	2	64	4	150	1200	
80000	2	16	1	80	640	S-bloki w RAM

Tab. 4.4. Wydajność algorytmu Twofish dla implementacji sprzętowej z 128-bitowym kluczem

* - klucze tworzone w czasie procesu szyfrowania.