Biuletyn WAT
Vol. LX, Nr 3, 2011

Implementacja algorytmu SOSEMANUK w strukturze FPGA

KAMIL KACZYŃSKI

Wojskowa Akademia Techniczna, Wydział Cybernetyki,
Instytut Matematyki i Kryptologii, 00-908 Warszawa, ul. S. Kaliskiego 2
email: kaczynski.kamil@wp.pl

Streszczenie. W artykule przedstawiono implementację algorytmu SOSEMANUK w strukturze FPGA Altera Stratix II. Przedstawiona została specyfikacja algorytmu wraz z charakterystyką bezpieczeństwa. Wykonano analizę możliwości implementacji, zajętości zasobów oraz wydajności algorytmu SOSEMANUK w przedstawionej platformie sprzętowej. Wykonane zostało porównanie uzyskanych wyników z algorytmami profilu sprzętowego konkursu eSTREAM oraz z przedstawioną przez twórców implementacją programową.
Słowa kluczowe: algorytmy strumieniowe, implementacja sprzętowa, kryptoanaliza algorytmów strumieniowych

1. Wstęp

SOSEMANUK jest algorytmem obecnym w profilu 1 (do zastosowań programowych) portfolio konkursu eSTREAM. W ramach niniejszej pracy została przeprowadzona implementacja tego szyfru w strukturze programowalnej FPGA. Do implementacji użyto platformy programowej Altera Quartus II 9.1 sp1, a układ docelowy należy do rodziny Altera Stratix II. Wykonana została analiza możliwości implementacji, zajętości zasobów oraz wydajności algorytmu SOSEMANUK. Następnie porównana została wydajność zaimplementowanego algorytmu z algorytmami profilu sprzętowego konkursu eSTREAM, a także z testową implementacją programową. Na podstawie uzyskanych wyników został także przedstawiony sposób skrócenia czasu wymaganego do inicjalizacji algorytmu.

2. Specyfikacja algorytmu SOSEMANUK

SOSEMANUK jest synchronicznym szyfrem strumieniowym do zastosowań programowych. Został zgłoszony do profilu I projektu eSTREAM. Szyfr ten jest połączeniem szyfru strumieniowego SNOW 2.0 oraz blokowego Serpent. Celem stworzenia tego algorytmu było podniesienie zarówno wydajności jak i bezpieczeństwa algorytmu SNOW 2.0. Stało się to możliwe dzięki wykorzystaniu szybszej procedury ustawiania wartości początkowych, a także dzięki redukcji ilości danych statycznych, co zaowocowało poprawą wydajności na niektórych platformach sprzętowych.

2.1. Serpent i jego odmiany

Serpent jest szyfrem blokowym zgłoszonym do konkursu AES. Operuje na 128-bitowych blokach tekstu jawnego, które ulegają podziałowi na cztery 32-bitowe słowa, na których wykonywane są operacje w trybie „bitslice". Wejścia i wyjścia algorytmu Serpent numerujemy w zakresie $0, \ldots, 3$ i zapisujemy w porządku (Y_3, Y_2, Y_1, Y_0), gdzie Y_0 jest najmniej znaczącym słowem i zawiera najmniej znaczące bity wszystkich czterech 32-bitowych wejść do skrzynek podstawieniowych algorytmu Serpent. Kiedy wyjście algorytmu jest zapisywane w postaci 16 bajtowej (128 bitowej), wartości Y_i są zapisywane w konwencji *little-endian* (najmniej znaczący bit najpierw) i Y_0 jest wysyłane najpierw, następnie Y_1 itd. W trakcie opracowywania algorytmu SOSEMANUK wykorzystane zostały dwa rodzaje algorytmu Serpent – Serpent1 oraz Serpent24.

Serpent1 jest jedną rundą algorytmu Serpent, bez operacji dodania klucza i przekształceń liniowych. Serpent używa ośmiu różnych s-boxów (skrzynek podstawieniowych, *substitution boxes*), numerowanych $S_0, \ldots, S_7$ z 4-bajtowymi słowami. Na potrzeby algorytmu Serpent1 został zdefiniowany jako aplikacja skrzynki S_2 w trybie „bitslice". Jest to trzecia warstwa podstawieniowa Serpenta. Jako argument Serpent1 pobiera cztery 32-bitowe słowa i dostarcza cztery 32-bitowe słowa jako wynik swojego działania.

Serpent24 jest zredukowanym algorytmem Serpent z 32 do 24 rund. Jest on tożsamy z pierwszymi 24 rundami algorytmu Serpent, gdzie ostatnia runda jest kompletna – zawiera transformacje liniowe i operacje XOR z 25 podkluczem. Poniżej przedstawione jest równanie przedstawiające działanie 24 rundy:

$$R_{23}(X) = L(\hat{S}_{23}(X \otimes \acute{K}_{23})) \otimes \acute{K}_{24}.$$

Serpent24 używa tylko 25 128-bitowych podkluczy, które są pierwszymi 25 podkluczami wygenerowanymi przez pełny Serpent. SOSEMANUK wykorzystuje podczas inicjalizacji Serpent24 w trybie szyfrowania. Tryb deszyfrowania tego algorytmu nie jest potrzebny.

2.2. LFSR (*Linear Feedback Shift Register*)

2.2.1. Podstawowe ciało skończone

SOSEMANUK przetrzymuje swój stan wewnętrzny w LFSR (rejestr przesuwny z liniowym sprzężeniem zwrotnym) zawierającym 10 elementów z $GF(2^{32})$. Elementy $GF(2^{32})$ są reprezentowane identycznie jak w przypadku SNOW 2.0. Niech $GF(2)$ oznacza ciało skończone z dwoma elementami, a β pierwiastek wielomianu pierwotnego:

$$Q(X) = X^8 + X^7 + X^5 + X^3 + 1.$$

Ciało $GF(2^8)$ możemy zdefiniować jako iloraz $GF(2)[X]/Q(X)$. Każdy element z $GF(2^8)$ jest reprezentowany przez bazę $(\beta^7, \beta^6, \ldots, \beta, 1)$. β jest generatorem multiplikatywnym każdego odwracalnego elementu z $GF(2^8)$: każdy niezerowy element jest równy β^k dla pewnej liczby całkowitej k $(0 \leqslant k \leqslant 254)$. Każdy element z $GF(2^8)$ może być określony za pomocą następującej bijekcji:

$$\phi : \ GF(2^8) \to \{0, 1, \ldots, 255\}$$

$$x = \sum_{i=0}^{7} x_i \beta^i \mapsto \sum_{i=0}^{7} x_i 2,$$

gdzie każde x_i jest równe 0 lub 1. Dla przykładu β^{23} jest reprezentowane przez liczbę $\varphi(\beta^{23}) = 0\text{xE1h}$ (193). Dodawanie dwóch elementów w $GF(2^8)$ odpowiada operacji bitowej XOR pomiędzy odpowiadającymi sobie reprezentacjami binarnymi liczb całkowitych. Mnożenie przez β jest przesunięciem w lewo o jeden bit, a następnie wykonaniem operacji XOR z poprawioną maską, w przypadku gdy bit na pozycji x^8 był równy 1.

Niech α będzie pierwiastkiem wielomianu pierwotnego:

$$P(X) = X^4 + \beta^{23} X^3 + \beta^{245} X^2 + \beta^{48} X + \beta^{239}$$

w $GF(2^8)[X]$. Ciało $GF(2^{32})$ jest wtedy zdefiniowane jako iloraz $GF(2^8)[X]/P(X)$, np. jego elementy są reprezentowane przez bazę

$(\alpha^3, \alpha^2, \alpha, 1)$. Każdy element z $GF(2^{32})$ może być zdefiniowany jako 32-bitowa liczba całkowita poprzez następującą bijekcję:

$$\psi : GF(2^{32}) \to \{0, 1, \ldots, 2^{32} - 1\},$$

$$x = \sum_{i=0}^{3} x_i \alpha^i \mapsto \sum_{i=0}^{3} \phi(y_i) 2^{8i}.$$

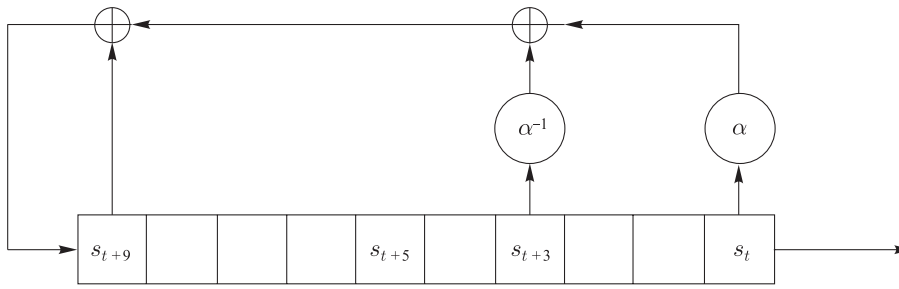
Stąd dodawanie dwóch elementów w $GF(2^{32})$ odpowiada wykonaniu operacji bitowej XOR pomiędzy ich reprezentacjami binarnymi. Ta operacja będzie określana od tej chwili jako $\oplus$. SOSEMANUK także wykorzystuje mnożenie i dzielenie w $GF(2^{32})$ przez α. Mnożenie $z \in GF(2^{32})$ przez α odpowiada przesunięciu o 8 bitów z $\psi(z)$ w lewo, a następnie wykonaniu operacji XOR z 32-bitową maską, która zależy wyłącznie od najbardziej znaczącego bajtu $\psi(z)$. Dzielenie $z \in GF(2^{32})$ przez α jest przesunięciem o 8 bitów z $\psi(z)$ w prawo, następnie wykonuje się operację XOR z 32-bitową maską, która zależy tylko od najmniej znaczącego bajtu $\psi(z)$.

2.2.2. Definicja LFSR

LFSR operuje na elementach z $GF(2^{32})$. Stan początkowy, przy $t = 0$, tworzy dziesięć 32-bitowych wartości s_1 do s_{10}. W każdym kroku algorytmu nowa wartość jest wyliczona za pomocą następującej rekurencji:

$$s_{t+10} = s_{t+9} \oplus \alpha^{-1} s_{t+3} \oplus \alpha s_t, \quad \forall t \geqslant 1.$$

następnie rejestr jest przesuwany.



Rys. 1. Rekurencja opisująca LFSR algorytmu SOSEMANUK [3]

LFSR jest powiązany z następującym wielomianem:

$$\Pi(X) = \alpha X^{10} + \alpha^{-1} X^7 + X + 1 \in GF(2^{32})[X].$$

Jeśli LFSR jest niejednorodny i Π jest wielomianem pierwotnym, sekwencja 32-bitowych słów $(s_t)_{t \geqslant 1}$ jest okresowa i ma maksymalny okres $(2^{320} - 1)$.

2.3. Skończona Maszyna Stanów (FSM)

Skończona Maszyna Stanów jest komponentem posiadającym 64-bitową pamięć, zawartą w dwóch 32-bitowych rejestrach $R1$ i $R2$. W każdym kroku FSM jako wejście pobiera słowa z LFSR, co powoduje zaktualizowanie pamięci i stworzenie 32-bitowego słowa na wyjściu.

$$\mathrm{FSM}_t\colon\ (R1_{t-1}, R2_{t-1}, s_{t+1}, s_{t+8}, s_{t+9}) \to (R1_t, R2_t, f_t),$$

gdzie:

$$R1_t = (R2_{t-1} + \mathrm{mux}(\mathrm{lsb}(R1_{t-1}), s_{t+1}, s_{t+1} \oplus s_{t+8})) \bmod 2^{32}, \tag{1}$$

$$R2_t = Trans(R1_{t-1}), \tag{2}$$

$$f_t = (s_{t+9} + R1_t \bmod 2^{32}) \oplus R2_t. \tag{3}$$

$\mathrm{lsb}(x)$ to najmniej znaczący bit x, $\mathrm{mux}(c, x, y)$ jest równe x, jeśli $c = 0$, lub y, jeśli $c = 1$. Wewnętrzna funkcja przejścia $Trans$ na ciele $GF(2^{32})$ jest zdefiniowana jako:

$$Trans(z) = (M \times z \bmod 2^{32}) \lll 7,$$

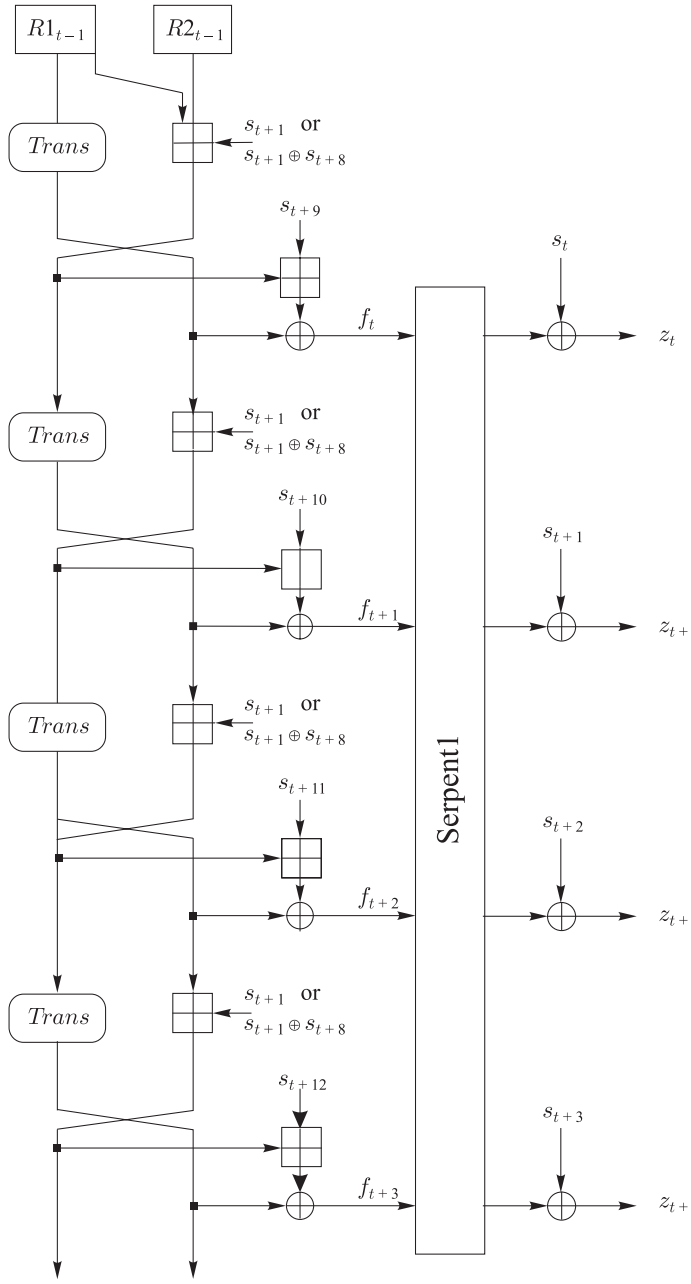
gdzie M zdefiniowane jest jako stała 0x54655307 (szesnastkowe rozwinięcie pierwszych dziesięciu cyfr Π), a $\lll$ oznacza bitową rotację w lewo.

2.4. Transformacja wyjściowa

Dane wychodzące z FSM są grupowane w czwórki, a na każdej z nich wykonywany jest Serpent1. Wynik jest poddawany operacji XOR z odpowiadającymi wartościami z LFSR:

$$(z_{t+3}, z_{t+2}, z_{t+1}, z_t) = \mathrm{Serpent1}(f_{t+3}, f_{t+2}, f_{t+1}, f_t) \oplus (s_{t+3}, s_{t+2}, s_{t+1}, s_t).$$

Schemat kolejnych czterech rund jest przedstawiony na obrazie poniżej.

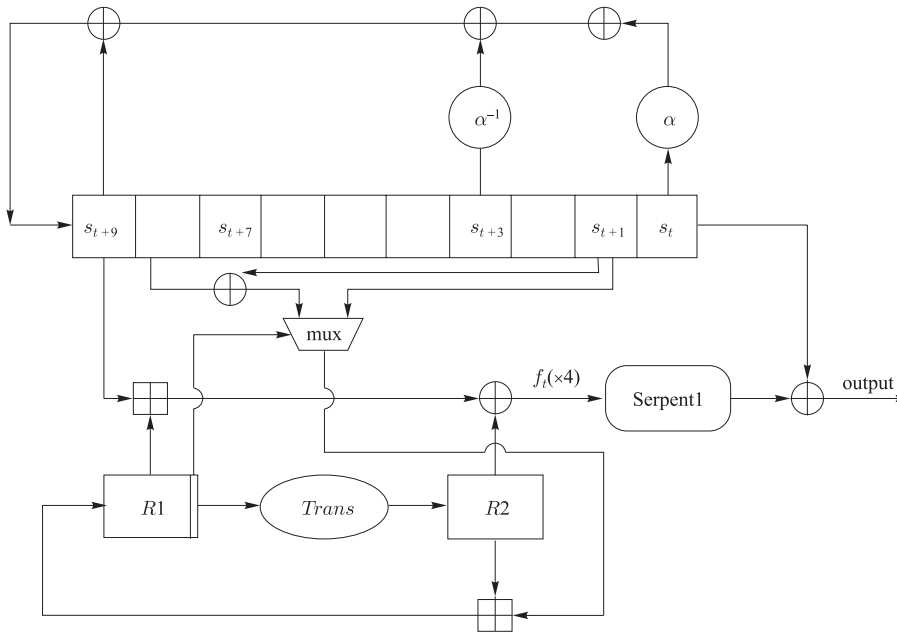


Rys. 2. Schemat kolejnych czterech rund algorytmu SOSEMANUK [3]

2.5. Przepływ zadań

SOSEMANUK łączy FSM i LFSR w celu wytworzenia wartości wyjściowych z_t. Czas $t = 0$ wyznacza stan wewnętrzny po zainicjalizowaniu, pierwszą wartością wyjściową jest z_1. W czasie $t \geqslant 1$ wykonywane są następujące operacje:

- FSM jest aktualizowane, $R1_t, R2_t$ i wartość pośrednia f_t są wyliczane z R_{t-1}, $R2_{t-1}$, s_{t+1}, s_{t+8} i s_{t+9}.
- LFSR jest aktualizowane: s_{t+10} jest obliczane na podstawie s_t, s_{t+3} i s_{t+9}. Wartość s_t jest wysyłana do wewnętrznego bufora, LFSR jest przesuwany.



Rys. 3. Przepływ zadań algorytmu [3]

Raz na każde cztery kroki, cztery wartości wyjściowe z_t, z_{t+1}, z_{t+2} i z_{t+3} są tworzone na podstawie zgromadzonych f_t, f_{t+1}, f_{t+2}, f_{t+3} oraz s_t, s_{t+1}, s_{t+2}, s_{t+3}. W ten sposób SOSEMANUK tworzy 32-bitowe wartości. Rekomendowane jest zapisywanie ich w grupach 4-bajtowych, zapisanych w konwencji *little-endian*, ponieważ zwiększa to wydajność na większości nowoczesnych platform PC, a także dlatego, że SERPENT również używa takiej konwencji. Stąd pierwsze cztery iteracje algorytmu wyglądają następująco:

- Stan wewnętrzny LFSR zawiera wartości s1 do s10, nie ma zdefiniowanej wartości s0. Stan wewnętrzny FSM zawiera $R1_0$, $R2_0$.
- Podczas pierwszego kroku $R1_1$, $R2_1$ i f_1 są obliczane z $R1_0$, $R2_0$, s_2, s_9 oraz s_{10}.
- Pierwszy krok generuje pośrednie, buforowane wartości s_1 i f_1.
- Podczas pierwszego kroku sprzężone słowo s_{11} jest obliczane z s_{10}, s_4 i s_1 a stan wewnętrzny LFSR jest aktualizowany przy wykorzystaniu danych z s_2 do s_{11}.
- Pierwsze cztery wartości wyjściowe z_1, z_2, z_3, z_4 są obliczane przy wykorzystaniu Serpent1 na danych (f_4, f_3, f_2, f_1), jego wyjście jest poddawane operacji XOR z (s_4, s_3, s_2, s_1).

2.6. Algorytm generowania podkluczy

Algorytm generowania podkluczy odpowiada stosowanemu w Serpent24, który produkuje 25 128-bitowych podkluczy jako 100 32-bitowych słów. Te 25 kluczy jest identyczne jak pierwsze 25 podkluczy generowanych przez oryginalny algorytm Serpent.

Serpent umożliwia pracę z kluczami o długości od 1 do 256 bitów, więc SOSEMANUK może pracować z dokładnie takimi samymi kluczami. Jednakże SOSEMANUK zapewnia bezpieczeństwo na poziomie 128 bitów, stąd długość klucza nie może być mniejsza niż 128 bitów. Wartość 128 bitów dla długości klucza jest standardem dla algorytmu, jednakże może on obsłużyć klucze o dowolnej długości w zakresie 128-256 bitów, ale poziom bezpieczeństwa nawet dla klucza 256-bitowego pozostaje na poziomie 128 bitów. Innymi słowy, dłuższy klucz nie gwarantuje zwiększenia poziomu bezpieczeństwa, jakiego można by się spodziewać po kluczu o takiej długości.

2.7. Ustawianie wartości początkowych

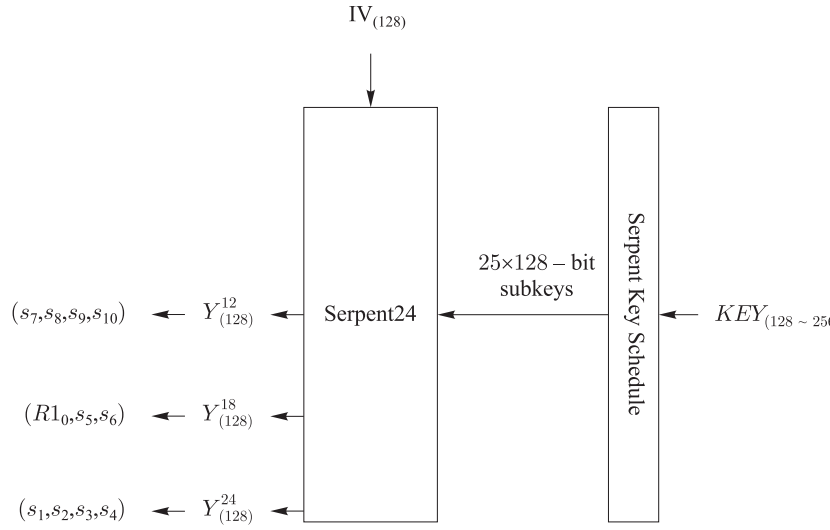
Wartość początkowa składa się ze 128 bitów. Jest używana jako wejście do algorytmu generowania podkluczy Serpent24. Serpent24 składa się z 24 rund, ale używane są tylko produkty 12., 18. i 24. rundy. Opisane są one następująco:

- $(Y_3^{12}, Y_2^{12}, Y_1^{12}, Y_0^{12})$: produkt 12. rundy;
- $(Y_3^{18}, Y_2^{18}, Y_1^{18}, Y_0^{18})$: produkt 18. rundy;
- $(Y_3^{24}, Y_2^{24}, Y_1^{24}, Y_0^{24})$: produkt 24. rundy;

Produkt każdej rundy składa się z czterech 32-bitowych wartości, wziętych tuż po przekształceniu liniowym. Wyjątkiem jest tu produkt 24. rundy

– jest on brany po dodaniu 25. podklucza. Wartości te są używane do zainicjowania SOSEMANUK w następujący sposób:

$$
\begin{aligned}
(s_7, s_8, s_9, s_{10}) &= (Y_3^{12}, Y_2^{12}, Y_1^{12}, Y_0^{12}) \\
(s_5, s_6) &= (Y_1^{18}, Y_3^{18}) \\
(s_1, s_2, s_3, s_4) &= (Y_3^{24}, Y_2^{24}, Y_1^{24}, Y_0^{24}) \\
R1_0 &= Y_0^{18} \\
R2_0 &= Y_2^{18}
\end{aligned}
$$



Rys. 4. Inicjalizacja algorytmu SOSEMANUK [4]

3. Charakterystyka bezpieczeństwa algorytmu SOSEMANUK

3.1. Ataki typu *time-memory-data tradeoff*

Dzięki wyborowi odpowiedniej długości LFSR (ponad dwa razy dłuższy niż klucz) większość tego typu ataków jest niemożliwa do zrealizowania w praktyce. Dodatkowo celem tych ataków jest odtworzenie stanu wewnętrznego szyfru, więc kompromitacja klucza wymaga dodatkowych obliczeń związanych z atakiem przeciwko Serpent24. Najlepszy atak tego typu został zaproponowany przez Hellmana. Jego celem jest odzyskanie

pary (K,IV). Dla 128-bitowego klucza i 128-bitowego IV jego złożoność obliczeniowa jest równa 2^{128} [6].

3.2. Ataki typu *guess and determine*

Główną słabością poprzednika szyfru SOSEMANUK – SNOW 1.0 jest podatność na tego typu ataki. Wykorzystują one pewną słabość w równaniu rekurencji liniowej. Jednak słabość ta nie występuje już w wielomianie wykorzystanym w SNOW 2.0, a co za tym idzie także w SOSEMANUK. Wplątane są tam niebinarne mnożenia przez dwie różne stałe. Najlepszy atak *guess and determine* znaleziony przez twórców algorytmu SNOW 2.0 [7] jest następujący:

- W czasie t zgadujemy: $s_t, s_{t+1}, s_{t+2}, s_{t+3}, R1_{t-1}$ i $R2_{t-1}$ (6 słów).
- Obliczamy odpowiednie wyjścia FSM$(f_t, f_{t+1}, f_{t+2}, f_{t+3})$.
- Obliczamy $R2_t = Trans(R1_{t-1})$ i $R1_t$, jeżeli $\mathrm{lsb}(R1_{t-1}) = 1$.
- Z $f_t = (s_{t+9} + R1_t \bmod 2^{32}) \oplus R2_t$, obliczamy s_{t+9}.
- Obliczamy $R1_{t+1}$ na podstawie wiedzy o s_{t+2} i s_{t+9}; obliczamy $R2_{t+1}$, następnie wyliczamy s_{t+10} z f_{t+1}, $R1_{t+1}$ i $R2_{t+1}$.
- Obliczamy $R1_{t+2}$ na podstawie s_{t+2} oraz s_{t+10}; obliczamy $R2_{t+2}$ oraz s_{t+11} z f_{t+2}, $R1_{t+2}$ oraz $R2_{t+2}$. Teraz s_{t+4} może być odzyskane z następującej relacji w czasie $t+1$:

$$\alpha^{-1} s_{t+4} = s_{t+11} \oplus s_{t+10} \oplus \alpha s_{t+1}.$$

- Obliczamy $R1_{t+3}$, s_{t+4} i s_{t+11}; obliczamy $R2_{t+3}$, następnie wyliczamy s_{t+12} z f_{t+3}, $R1_{t+3}$ i $R2_{t+3}$. Następnie możemy odzyskać s_{t+5} z następującej relacji w czasie $t+2$:

$$\alpha^{-1} s_{t+5} = s_{t+12} \oplus s_{t+11} \oplus \alpha s_{t+2}.$$

W tym miejscu słowa LFSR: s_t, s_{t+1}, s_{t+2}, s_{t+3}, s_{t+4}, s_{t+5}, s_{t+9}, są znane. Trzy elementy $(s_{t+6}, s_{t+7}, s_{t+8})$ pozostają nieznane. Żeby skompletować wszystkie 10 słów stanu LFSR, musimy odgadnąć jeszcze dwa słowa s_{t+6} i s_{t+7} od każdego $f_{t+i} 4 \leqslant i \leqslant 7$, zależne od wszystkich czterech słów: $s_{t+4}, s_{t+5}, s_{t+6}$ i s_{t+7}. Stąd atak ten wymaga odgadnięcia ośmiu 32-bitowych słów, co daje złożoność równą 2^{256}.

3.3. Ataki korelacyjne

W czasie poszukiwania właściwego ataku korelacyjnego na SOSEMANUK nasuwają się następujące pytania:

- Czy istnieje liniowa relacja na poziomie bitowym pomiędzy pewnymi bitami wejściowymi i wyjściowymi?
- Czy istnieje pewna szczególna relacja pomiędzy pewnymi wektorami wejściowymi i wyjściowymi?

W pierwszym przypadku możemy wyróżnić dwie liniowe relacje. Pierwsza z nich – najmniej znaczący bit s_{t+9} jest zachowywany do czasu dodawania modularnego nad $Z_{2^{32}}$, jest operacją liniową na najmniej znaczącym bicie. Druga relacja dotyczy najmniej znaczącego bitu s_{t+1} lub $s_{t+1} \oplus s_{t+8}$ (używane do wyliczenia $R1_t$) lub siódmego bitu $R2_t$ obliczanego z s_t lub $s_t \oplus s_{t+7}$. Przyjęte zostało, że $R2_t = Trans(R1_{t-1})$ oraz $R1_{t-1} = R2_{t-2} + (s_t \text{ lub } (s_t \oplus s_{t+7})) \bmod 2^{32}$.

Żadna liniowa relacja nie może zostać odtworzona po wykonaniu Serpent1. Co więcej, szybki atak korelacyjny jest niewykonalny w praktyce ze względu na operację mux zamazującą powiązania stanu wewnętrznego LFSR i obserwowanego szyfrogramu.

3.4. Ataki rozróżniające

Atak rozróżniający opracowany przeciwko pierwszej wersji algorytmu SNOW wykorzystywał słabość wielomianu sprzężonego zbudowanego na pojedynczym mnożeniu przez α. Atak ten nie ma zastosowania dla algorytmu SNOW 2.0, a co za tym idzie także dla SOSEMANUK, ponieważ nowy wielomian nie posiada ww. słabości i ma zawarte także mnożenie przez α^{-1}.

D. Watanabe, A. Biryukov i C. De Canniere [8] opracowali nowy atak rozróżniający przeciwko SNOW 2.0 ze złożonością około 2^{225} operacji wykorzystujący metodę wielokrotnego liniowego maskowania. Skonstruowali trzy różne maski: $\Gamma_1 = \Gamma,\ \Gamma_2 = \Gamma \cdot \alpha,\ \Gamma_3 = \Gamma \cdot \alpha^{-1}$ bazujące na tej samej relacji liniowej Γ.

Właściwości liniowe wydedukowane na podstawie maski Γ_i $(i = 1, 2, 3)$ muszą iść w parze z wysokim prawdopodobieństwem następujących zdarzeń: $\Gamma_i \cdot S'(x) = \Gamma_i \cdot x$ oraz $\Gamma_i \cdot z \oplus \Gamma_i \cdot t = \Gamma_i \cdot (z \oplus t)$ dla $i = 1, 2, 3$, gdzie S' oznacza funkcję przejściową FSM w SNOW 2.0. W przypadku tego algorytmu najtrudniejszą hipotezą do spełnienia jest pierwsza zdefiniowana $y = S'(x)$. W przypadku algorytmu SOSEMANUK wymagane jest, aby prawdopodobieństwo $\Pr(\Gamma_i \cdot Trans(x) = \Gamma_i \cdot x)$ było wysokie. Ale także wymagamy, aby $\forall i = 1, 2, 3$ relacja:

$$(\Gamma_i', \Gamma_i', \Gamma_i', \Gamma_i') \cdot (x_1, x_2, x_3, x_4) = \text{Serpent1}((\Gamma_i, \Gamma_i, \Gamma_i, \Gamma_i) \cdot (x_1, x_2, x_3, x_4))$$

dla pewnych $\Gamma_i' \in GF(2^{32})$ zachodziła z wysokim prawdopodobieństwem.

Ze względu na właściwości projektowe Serpent1 (bitslice), znalezienie odpowiedniej maski wydaje się być skrajnie trudnym zadaniem. Stąd też atak ten nie może być bezpośrednio wykorzystany do ataku w przypadku algorytmu SOSEMANUK.

3.5. Ataki algebraiczne

Rozważmy stan wewnętrzny LFSR na poziomie bitowym:

$$(s_{10}, \ldots, s_1) = (s_{10}^{31}, \ldots, s_{10}^{0}, \ldots, s_1^{31}, \ldots, s_1^{0}).$$

Wtedy stan wyjść algorytmu w czasie t możemy zapisać:

$$F^t((s_{31}^{10}, \ldots, s_0^1)) = (z_t, z_{t+1}, z_{t+2}, z_{t+3}),$$

gdzie F jest wektorową funkcją boolowską $F_2^{320} \rightarrow F_2^{128}$, która może być rozpatrywana jako 128 funkcji F_j, $\forall j \in [0..127]$ przekształcająca $F_2^{320} \rightarrow F_2$.

Będziemy rozpatrywać stopień funkcji F_j zależący od poszczególnych bitów wyjścia lub ich liniowej kombinacji, ponieważ nie jest możliwe bezpośrednie obliczenie algebraicznej odporności każdej z funkcji F_j, gdyż liczba zmiennych jest zbyt duża (320 bitów). Następujące stwierdzenia wykluczają istnienie relacji niskiego stopnia pomiędzy wejściami i wyjściami F_j.

- Wyjściowy bit i po wykonaniu operacji dodawania modularnego w $Z_{2^{32}}$ jest na pozycji $i + 1$.
- Wyjściowy bit i po wykonaniu $Trans$ jest stopnia $i + 1 - 7 \bmod 32$ $\forall i \neq 6$ i równe 32 dla $i = 6$.
- Operacja mux uniemożliwia przewidzenie z prawdopodobieństwem bliskim 1 dokładnej liczby stanów wewnętrznych zaangażowanych w relację algebraiczną.
- Odporność algebraiczna skrzynki podstawieniowej S_2 algorytmu Serpent dla 4-bitowego słowa jest równa 2.

Z powyższego można wywnioskować, że atak algebraiczny na SOSEMANUK jest niewykonalny.

3.6. Najlepszy znany atak typu *guess-and-determine*

W punkcie tym przedstawiony zostanie najlepszy obecnie znany atak na algorytm SOSEMANUK z 256-bitowym kluczem. Przyjęto następujące założenia:

- Stała wartość klucza w czasie przeprowadzania ataku.
- Atakujący jest w stanie przejąć część szyfrogramu.

Założenie: $\mathrm{lsb}(R1_{t-1}) = 0$
Kiedy powyższe założenie jest spełnione, rejestr $R1_t$ jest uaktualniany zgodnie z poniższym wzorem:

$$R1_t = R2_{t-1} + s_{t+1}.$$

Jak widać, gdy założenie jest spełnione, wartość $R1_t$ jest niezależna od s_{t+8}, co może zostać wykorzystane podczas kryptoanalizy. Zakładając, że $t = 1$ spełnia warunki założenia, można odgadywać stan wewnętrzny algorytmu tuż po jego inicjalizacji.

Przypuszczenie 1: $s_1, s_2, s_3, s_4, R1_0, R2_0$
Jeżeli $\mathrm{lsb}(R1_0) = 0$, to pozostaje 191 bitów do odgadnięcia. Równanie (1) i (2) uaktualniają wartości rejestrów $R1_1$, $R2_1$, odpowiednio gdy $t = 1$. Równania (3) i (4) używane są do uaktualnienia wartości rejestru s_{11}, a także do wygenerowania wartości wyjścia f_1 FSM.

$$R1_1 = (R2_0 + s2), \tag{1}$$
$$R2_1 = Trans(R1_0), \tag{2}$$
$$s_{11} = s_{10} \oplus \alpha^{-1}(s_4) \oplus \alpha(s_1), \tag{3}$$
$$f_1 = (s_{10} + R1_1) \oplus R2_1. \tag{4}$$

Bazując na przypuszczeniu 1, wartości rejestrów $R1_1$ oraz $R2_1$ mogą zostać wyliczone. (f_1, f_2, f_3, f_4), wyjścia FSM dla kolejnych czterech kroków również można uzyskać, używając równania transformacji wyjściowej, gdzie odwrotność funkcji Serpent1 jest oznaczana jako $\mathrm{Serpent1}^{-1}$.

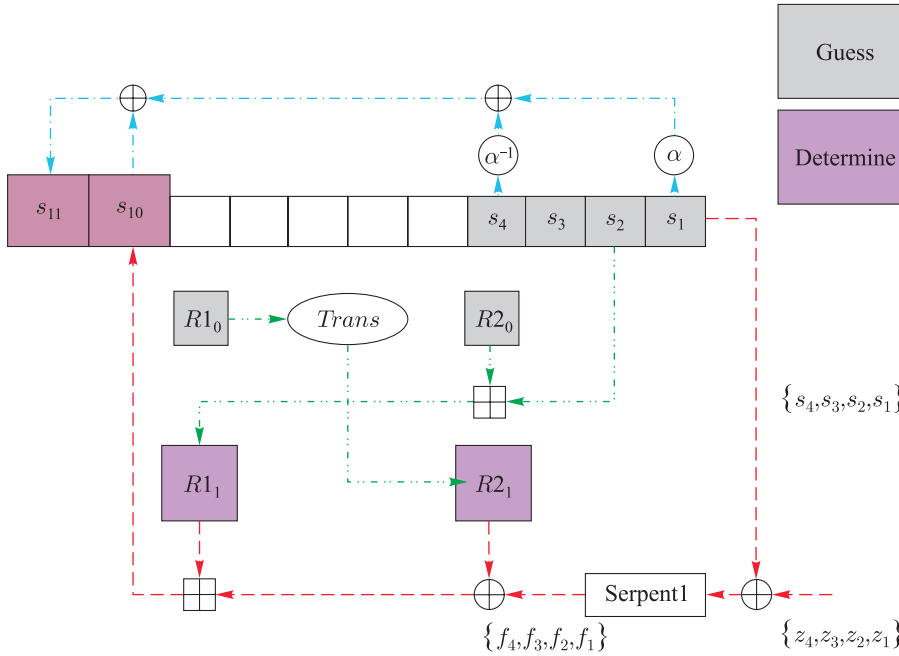
$$(f_4, f_3, f_2, f_1) = \mathrm{Serpent1}^{-1}(z_4 \oplus s_4,\ z_3 \oplus s_3,\ z_2 \oplus s_2,\ z_1 \oplus s_1)$$

Stąd, s_{10} i s_{11} mogą być określone, używając równań (4) i (3). W ten sposób odgadywanie części stanu wewnętrznego umożliwia określenie nieznanej jego części. Poniżej przedstawiony schemat obrazuje, jak wykonywana jest kryptoanaliza w czasie $t = 1$.

Następnie dla t=2 należy obliczyć parę $(R1_2, R2_2)$, używając w tym celu równania (5) uaktualniającego FSM.

$$f_2 = (s_{11} + R1_2) \oplus R2_2 \tag{5}$$

Ponieważ $s_{11}, f_2, R1_2, R2_2$ są określone na podstawie przypuszczenia 1, więc jeżeli jakikolwiek błąd był w nim zawarty, równanie (5) będzie



Rys. 5. Krok kryptoanalizy dla $t = 1$ [4]

sprzeczne. W tym przypadku atakujący może odrzucić przypuszczenie 1 i spróbować sformułować następne. Te działania powodują zawężenie ogólnej liczby kandydatów na przypuszczenie 1 do 2^{32} wszystkich możliwych przypuszczeń.

Podobnie jak w poprzednich krokach, uzyskanie danych dla $t = 3$ jest możliwe przy użyciu równania uaktualniającego FSM, zawartości rejestrów $R1_3$ oraz $R2_3$. Wykorzystując równanie generujące wyjście f_3 FSM, można obliczyć s_{12}. Teraz podstawiając s_{12} do równania uaktualniającego LFSR, w momencie $t = 2$ otrzymujemy:

$$s_5 = \alpha(s_{12} \oplus s_{11} \oplus \alpha(s_2))$$

Stąd s_5 również może być określone. Wykonując podobne kroki, można określić stan wewnętrzny dla $t = 4$, a następnie $R1_4, R2_4, s_6$ i s_{13} może być obliczone. Jak dotąd 191 bitów musiało zostać odgadnięte, ale liczba możliwych kombinacji została zawężona do 2^{159}.

Podobnie określamy stan wewnętrzny dla $t = 5$ do $t = 8$. Znając s_5 i s_6 dla $t = 1 \ldots 4$, należy dokonać przypuszczenia 2.

Przypuszczenie 2: s_7, s_8
Poza bitami określonymi do tej pory, pozostają do odgadnięcia jeszcze 64

bity. Przy wykorzystaniu przypuszczenia 2 możemy określić (f_5, f_6, f_7, f_8), używając równania transformacji wyjściowej. W trakcie przewidywania stanu wewnętrznego (podobnie jak uprzednio) atakujący może sprawdzić, czy nie zachodzi sprzeczność w przypuszczeniu 1 i/lub przypuszczeniu 2. Może to być wykonane trzykrotnie przy użyciu równania generującego wyjścia f_t FSM. Ten krok zawęża liczbę kandydatów do 2^{96} ich całkowitej liczby. Do tej pory atakujący odgadnął 223 bity, a liczba ich różnych kombinacji jest równa 2^{127}. Kolejne rejestry stanu wewnętrznego, to jest $R1_t, R2_t$ $(t = 1 \ldots 8)$ i s_t $(t = 1 \ldots 18)$ także mogą być określone.

Podejmując podobne kroki możliwe staje się określenie stanu wewnętrznego dla $t = 9$ do $t = 12$. Ponieważ s_9, s_{10}, s_{11} i s_{12} są znane dla $t = 1$ do $t = 8$, nie ma potrzeby, by formułować nowe przypuszczenia. Atakujący ma cztery możliwości sprawdzenia, czy nie występuje sprzeczność w równaniach generujących wyjście f_t FSM. W trakcie ataku możemy zawęzić liczbę kandydatów na przypuszczenie 1 i przypuszczenie 2 do 2^{125} ich całkowitej liczby. Oznacza to, że teoretycznie wszystkie 384 bity stanu po inicjalizacji mogą być unikalnie określone.

Tabela 1

Rejestry określane w momencie t i kandydaci do przypuszczenia 1 oraz przypuszczenia 2 [4]

T	Rejestry do odgadnięcia	Określane rejestry	Możliwość wykrycia sprzeczności	Liczba kandydatów
1	$s_1, s_2, s_3, s_4, R1_0, R2_0$	$s_{10}, s_{11}, R1_1, R2_1$		2^{191}
2		$R1_2, R2_2$	✓	2^{159}
3		$s_5, s_{12}, R1_3, R2_3$		
4		$s_6, s_{13}, R1_4, R2_4$		
5	s_7, s_8	$s_{14}, s_{15}, R1_5, R2_5$	✓	2^{191}
6		$R1_6, R2_6$	✓	2^{159}
7		$s_9, s_{16}, s_{17}, R1_7, R2_7$		
8		$s_{18}, R1_8, R2_8$	✓	2^{127}
9		$s_{19}, R1_9, R2_9$	✓	2^{95}
10		$s_{20}, R1_{10}, R2_{10}$	✓	2^{63}
11		$s_{21}, R1_{11}, R2_{11}$	✓	2^{31}
12		$s_{22}, R1_{12}, R2_{12}$	✓	1

Powyższa tabela przedstawia zestawienie określanych w momencie t rejestrów i liczbę kandydatów na przypuszczenie 1 i przypuszczenie 2.

Teraz możliwe jest określenie całkowitej ilości obliczeń, które muszą być wykonane podczas tego ataku. Przypuszczenie 1 i 2 wymagają odgadnięcia 2^{223} bitów. Prawdopodobieństwo, że założenie $(\mathrm{lsb}(R1_{t-1}) = 0)$

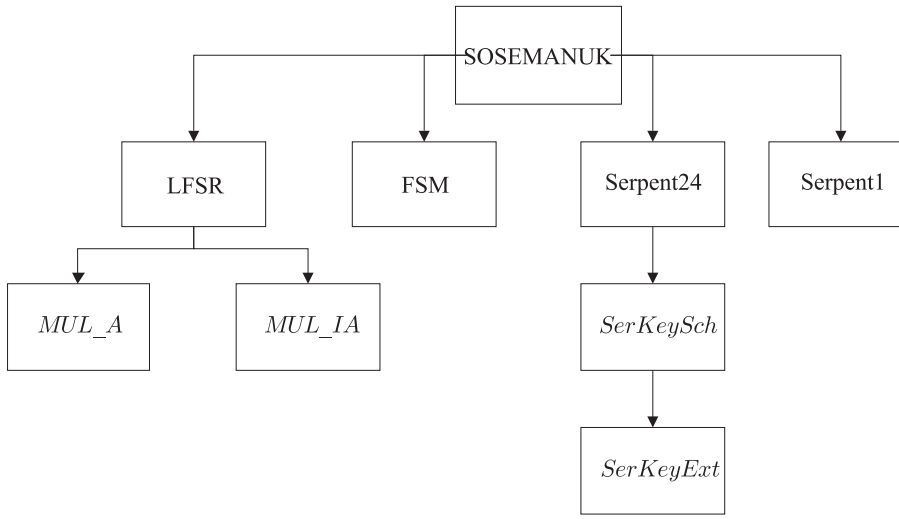
jest prawdziwe, wynosi 1/2. Całkowita liczba obliczeń jest więc równa $2^{223} \cdot 2^1 = 2^{224}$. W tym miejscu warto również zaznaczyć, że do wykonania tego ataku taka liczba nie jest potrzebna, ponieważ kandydaci na odpowiednie przypuszczenie są sprawdzani jeden po drugim.

Ten typ ataków wymaga porównania szyfrogramu, przechwyconego przez atakującego z dwunastoma szyfrogramami otrzymanymi poprzez wykonanie algorytmu. Stąd, do poprawnego przeprowadzenia ataku wystarcza przechwycenie zaledwie 24 słów szyfrogramu.

Warto zaznaczyć, że bezpieczeństwo szyfru deklarowane przez jego twórców jest niezależne od długości klucza i jest na poziomie 128 bitów. Ponieważ najlepszy znany atak wymaga wykonania 2^{224} obliczeń, algorytm SOSEMANUK może zostać uznany za bezpieczny.

4. Opis implementacji

Implementacja algorytmu SOSEMANUK została wykonana z wykorzystaniem języka AHDL oraz środowiska projektowego Altera Quartus II 9.1 sp1. Konstrukcje języka AHDL umożliwiają rozdzielenie implementacji na wiele modułów, co znacząco upraszcza analizę i testowanie wytworzonego kodu źródłowego. Właściwość ta została wykorzystana podczas implementacji szyfru SOSEMANUK.



Rys. 6. Schemat blokowy modułów implementacji algorytmu SOSEMANUK [opracowanie własne]

4.1. Główny moduł implementacji

Najważniejszym modułem szyfru jest moduł SOSEMANUK. Odpowiada on za generowanie strumienia klucza oraz synchronizację pracy innych modułów. Na potrzeby implementacji przyjęta została długość klucza równa 128 bitów oraz ze względu na dostępne wektory testowe długość 40 bitów. Wyjście modułu posiada 64-bitową długość i służy do wyprowadzenia wygenerowanego strumienia klucza. Rozwiązanie to zostało wymuszone przez niewystarczającą ilość wyprowadzeń układu – algorytm SOSEMANUK wytwarza 128-bitowe strumienie klucza, układ wystawia na wyjściu w pierwszym takcie dwa najstarsze słowa szyfrogramu, a w drugim takcie wystawiane są dwa najmłodsze słowa.

Moduł do swojej pracy wykorzystuje 6 przerzutników typu D i jeden 128-bitowy węzeł. Umożliwia on wykonanie operacji XOR na słowach dostarczonych przez Serpent1 i zapamiętanych wcześniej w rejestrze u słów wyrzucanych przez LFSR.

Moduł SOSEMANUK realizuje również inicjalizację rejestru LFSR oraz rejestrów Skończonej Maszyny Stanów.

4.2. Skończona Maszyna Stanów (FSM)

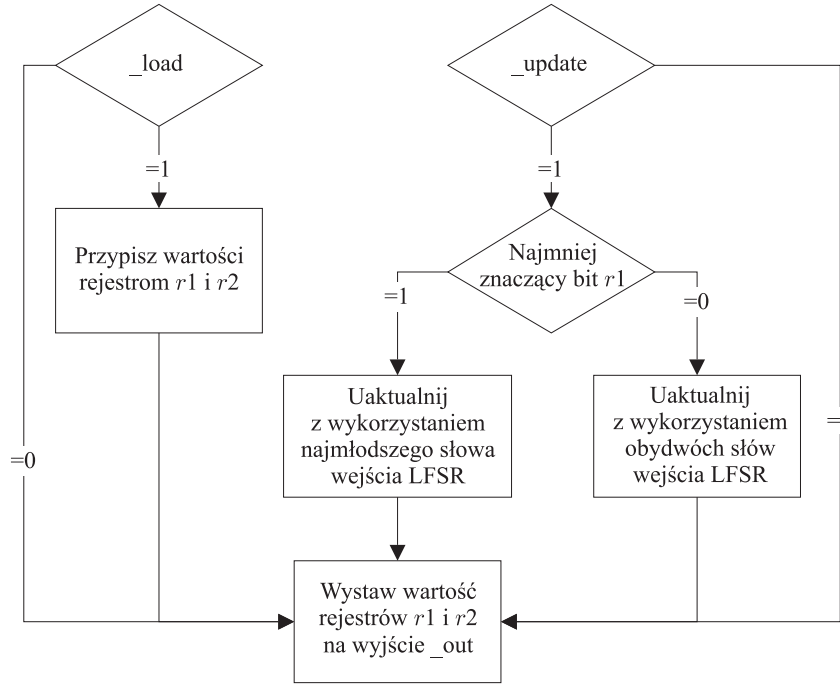
Moduł FSM odpowiada za obsługę Skończonej Maszyny Stanów algorytmu SOSEMANUK. W trybie ciągłym wystawia na wyjściu zawartości rejestrów $r1$ i $r2$. Najstarszym słowem wyjścia jest zawartość rejestru $r2$, najmłodszym zawartość rejestru $r1$.

Moduł wykorzystuje dwa 32-bitowe rejestry $r1$ i $r2$, są one odpowiednikami rejestrów zawartych w specyfikacji algorytmu. Dodatkowo w module zawarty jest jeden 32-bitowy węzeł umożliwiający wykonanie rotacji w lewo o 7 bitów. Multiplekser zawarty w specyfikacji algorytmu został zrealizowany przy wykorzystaniu instrukcji warunkowych if.

4.3. Rejestr Przesuwny z Liniowym Sprzężeniem Zwrotnym (LFSR)

Zadaniem modułu $LFSR$ jest ładowanie i aktualizacja rejestru LFSR. Moduł do swojej pracy potrzebuje dwóch dodatkowych modułów – MUL_A oraz MUL_IA. Moduły te są zrealizowane jako tablice prawdy i umożliwiają wykonanie operacji mnożenia i dzielenia przez α jako operacji XOR z 32-bitową maską.

Moduł posiada jeden 320-bitowy rejestr zbudowany na przerzutnikach typu D. Dodatkowo zostały zaimplementowane dwa 32-bitowe węzły umożliwiające wykonanie operacji przesunięcia bitowego.



Rys. 7. Schemat blokowy modułu FSM [opracowanie własne]

4.4. Serpent1

Moduł *Serpent*1 jest implementacją skrzynki numer 2 algorytmu Serpent w trybie „bitslice". Oznacza to, że operacje wykonywane na tej skrzynce mogą być opisane jako logiczne operacje na bitach.

4.5. Serpent24

*Serpent*24 jest najbardziej rozbudowanym modułem algorytmu SOSEMANUK. Jego zadaniem jest wygenerowanie wartości inicjalizacyjnych dla rejestru LFSR oraz Skończonej Maszyny Stanów. Do swojego działania potrzebuje modułu *SerKeySch* generującego 25 podkluczy.

Moduł wykonuje 24 rundy algorytmu Serpent. Każda z rund składa się z trzech operacji:

- Dodanie klucza – wykonuje dodawanie podklucza rundy.
- Podstawienie – przekształcenie wyników przy wykorzystaniu skrzynek podstawieniowych.
- Przekształcenie liniowe – wykonuje operacje przekształcenia liniowego algorytmu Serpent.

Wyjątkiem jest ostatnia runda algorytmu, podczas której po operacji przekształcenia liniowego dodawany jest ostatni 25. podklucz. Moduł potrzebuje 341 cykli zegara, żeby wykonać wszystkie 24 rundy algorytmu Serpent.

4.6. Algorytm generowania podkluczy (*SerKeySch*)

Moduł generujący podklucze jest integralną częścią modułu *Serpent*24, do swojej pracy potrzebuje modułu *SerKeyExt* odpowiedzialnego za dopełnianie klucza wprowadzonego do układu. Potrzebuje 217 cykli zegara, aby wygenerować wszystkie 25 podkluczy.

4.7. Dopełnianie klucza (*SerKeyExt*)

Moduł *SerKeyExt* jest najniżej położony w hierarchii projektu. Jego zadaniem jest wykonanie operacji dopełniania klucza oraz zamiana zapisu na konwencję *little-endian*. Moduł obsługuje klucze o długości 40 i 128 bitów, przy czym klucze o długości innej niż 40 bitów są traktowane jako klucze 128-bitowe z prowadzącymi zerami. Całość operacji jest wykonywana w czasie jednego cyklu zegara.

5. Omówienie osiągniętych wyników

SOSEMANUK jest algorytmem umieszonym w profilu programowym portfolio konkursu eSTREAM. Wydajność implementacji programowej tego szyfru została sprawdzona na specjalnym zestawie testów konkursowych. Szybkość generacji strumienia klucza na procesorze Pentium 4 2,8 GHz wynosi 3847 Mb/s. Czyni to szyfr wiceliderem wydajności portfolio konkursu.

Tabela 2
Wydajność algorytmów programowych portfolio konkursu eSTREAM [9]

Algorytm	Przepustowość [Mb/s]
HC-128	5951,04
SOSEMANUK	3847,00
Rabbit	2361,81
Salsa20/12	2105,47

Do implementacji sprzętowej został wykorzystany jeden z najmniejszych układów rodziny Altera Stratix II – EP2S15F672C3. Logika układu jest wykorzystana w 88% – implementacja wykorzystuje 8293 ALUT, 5414 ALM oraz 2743 dedykowane rejestry logiczne. Wykorzystane zostało także 87% (321 z 367) dostępnych wyprowadzeń.

Narzędzie analizy czasowej środowiska projektowego Altera Quatus II 9.1 sp1 określiło maksymalną wartość częstotliwości pracy układu na 121,65 MHz, co odpowiada okresowi zegara równemu 8,22 ns. Dla dostarczonych wektorów testowych układ dawał poprawne wyniki dla częstotliwości pracy równej 125 MHz (okres zegara równy 8 ns), co umożliwiło osiągnięcie wydajności na poziomie 3814,67 Mb/s.

5.1. Omówienie wydajności

SOSEMANUK został umieszczony w portfolio konkursu eSTREAM nie tylko ze względu na wysoki poziom bezpieczeństwa, ale także ze względu na wysoką wydajność implementacji programowej. Zrealizowana implementacja sprzętowa jest równie wydajna jak wersja programowa. Dzięki temu możliwe jest porównanie wydajności omawianej implementacji z wydajnością algorytmów zawartych w profilu sprzętowym.

Wykorzystane omówienie implementacji algorytmów profilu sprzętowego zostało zrealizowane przy wykorzystaniu układów FPGA z rodziny Xilinx Spartan 3. Do porównań wykorzystano implementację algorytmów zawartych w portfolio: Grain-16, Mickey-128 i Trivium-64 oraz algorytmu F-FCSR-16.

Tabela 3
Porównanie wydajności algorytmów profilu II konkursu eSTREAM i algorytmu SOSEMANUK [10]

Algorytm	Częstotliwość pracy (MHz)	Wydajność (Mb/s)
SOSEMANUK	121,65	3712,46
	125,00	3814,67
Trivium (x64)	211,00	13504,00
F-FCSR-16	134,00	2144,00
Grain-16	130,00	2080,00
Mickey-128	223,00	223,00

Powyższe porównanie pokazuje, że wykonana implementacja jest szybsza niż dwa z trzech algorytmów znajdujących się w portfolio konkursu. SOSEMANUK jest blisko dwukrotnie szybszy od algorytmów Grain-16 i F-FCSR-16 oraz blisko 19-krotnie szybszy od algorytmu Mickey-128. Jedynie 64-bitowa wersja algorytmu Trivium jest prawie 3,5-krotnie szybsza od wykonanej implementacji.

Wykonana implementacja potrzebuje 341 cykli zegara (2803 ns), by zainicjować stan wewnętrzny. Czas ten można by znacząco skrócić poprzez zmianę sposobu implementacji Serpent24. W tym celu należy usunąć rejestry pomocnicze dla obliczenia skrzynek podstawieniowych, zastąpić je

poprzez oddzielne moduły obliczające wszystkie wyrazy w jednym takcie zegara. Zmianie ulec powinien także sposób aktualizowania słowa klucza – w tym celu należy wykorzystać węzły i skrócić czas aktualizacji do 4 cykli zegarowych. Podobna zmiana powinna być zastosowana przy wykonaniu przekształceń liniowych algorytmu Serpent – ich czas wykonania powinien zostać skrócony do 1 cyklu zegara. Zmiana ta spowodowałaby skrócenie procesu generowania podkluczy do 100 cykli zegarowych, a procesu szyfrowania do 25 cykli, jednakże całość procesu nadal nie może być krótsza niż 100 cykli. Niestety, takie rozwiązania z pewnością spowodują wydłużenie okresu zegara, co będzie skutkowało obniżeniem wydajności implementacji sprzętowej.

5.2. Omówienie zajętości zasobów układu

Implementacja algorytmu SOSEMANUK wykorzystuje 8293 układy ALUT (5414 ALM, 2743 rejestry) oraz 321 z 367 dostępnych wyprowadzeń układu. Główny moduł implementacji – SOSEMANUK – zajmuje niewielką część przyporządkowanych zasobów układu. Jest to 381 układów ALUT, 267 ALM oraz 292 rejestry. Dosyć duże wykorzystanie rejestrów jest spowodowane wykonywaniem przez główny moduł implementacji przekształceń na słowach uzyskiwanych z LFSR oraz FSM i konieczność ich zapamiętywania w trakcie kolejnych rund algorytmu. Zapotrzebowanie na rejestry zwiększone jest także poprzez wykorzystanie dodatkowego rejestru umożliwiającego wyprowadzenie wygenerowanego strumienia klucza w dwóch taktach zegara.

Najbardziej zasobochłonnym modułem implementacji jest Sepent24 odpowiedzialny za wygenerowanie wartości inicjalizacyjnych algorytmu. Wraz z wykorzystywanymi modułami SerKeySch i SerKeyExt potrzebuje 7322 układy ALUT, 4908 ALM oraz 2062 rejestry. Tak wysokie zapotrzebowanie na zasoby jest podyktowane wykorzystywaniem rejestrów do wykonywania większości operacji algorytmu Serpent.

Następny ważny moduł implementacji – LFSR – wykorzystuje 393 układy ALUT, 224 ALM i 320 rejestrów. Liczba wykorzystywanych rejestrów jest równa długości zaimplementowanego LFSR. Moduł do swojej pracy wykorzystuje dwa moduły MUL_A oraz MUL_IA. Są to tablice prawdy pozwalające na wykonanie operacji mnożenia i dzielenia przez α. Moduł MUL_IA wykorzystuje 9 układów ALUT i 8 ALM, natomiast moduł MUL_A potrzebuje 13 układów ALUT i 12 ALM.

Kolejny moduł – FSM – odpowiedzialny za implementację Skończonej Maszyny Stanów zużywa 153 układy ALUT, 94 ALM oraz 69 rejestrów. Tak niewielkie wymagania co do zajętości zasobów układu są efektem prostoty

wykonywanych przez moduł funkcji. Jest to przede wszystkim realizacja multipleksera oraz uaktualnianie wartości rejestrów wewnętrznych.

Ostatni moduł – *Serpent*1 – to implementacja skrzynki nr 2 algorytmu Serpent. Wykorzystuje ona 22 układy ALUT i 22 ALM. Do swojego działania nie potrzebuje żadnych rejestrów.

Tabela 4
Wykorzystanie zasobów układu przez poszczególne moduły implementacji [opracowanie własne]

Moduł	ALUT	ALM	Rejestry
SOSEMANUK	381	267	292
LFSR	393	224	320
MUL_IA	13	12	0
MUL_A	9	8	0
*Serpent*24	4426	3026	1606
SerKeySch	2877	1950	456
SerKeyExt	19	19	0
FSM	153	94	69
*Serpent*1	22	22	0

6. Wnioski

SOSEMANUK to algorytm umieszony w portfolio konkursu eSTREAM w profilu I – algorytmy do zastosowań programowych. Opisana w tym dokumencie praca jest pierwszą próbą implementacji tego algorytmu w strukturze programowalnej. Głównym celem było uzyskanie jak największej wydajności generowania strumienia klucza. SOSEMANUK pracujący w układzie Stratix II jest równie szybki jak jego wersja pracująca na sprzęcie wykorzystującym procesor Pentium IV 2,8 GHz.

Czas inicjalizacji algorytmu jest wartością drugorzędną dla każdego algorytmu strumieniowego. Również omawiana implementacja skupiła się głównie na uzyskaniu maksymalnej wydajności, co zostało okupione czasem inicjalizacji równym 341 cyklom zegara, co przy okresie zegara równym 8 ns daje 2728 ns czasu inicjalizacji.

Uzyskana wydajność przy częstotliwości zegara równej 125 MHz jest równa 3814,67 Mb/s. Jest to wynik lepszy od dwóch z trzech algorytmów obecnych w portoflio profilu 2 konkursu eSTREAM – Grain v1 i Mickey v2.

Przeprowadzona implementacja nie ma dużych wymagań co do wielkości docelowego układu. Całość wykorzystuje 8293 układy logiczne oraz 321 wyprowadzeń, co pozwala na implementację algorytmu w najmniejszym układzie rodziny Stratix II. Wykorzystanie wyprowadzeń układu

może zostać zmniejszone o 128, kiedy przyjmiemy, że wartość IV zostanie podana na to samo wejście układu co wartość klucza, jednakże wartość klucza powinna obowiązywać w trakcie pierwszego taktu zegara, a wartość IV w ciągu dziewiętnastego cyklu zegarowego. Jednakże ze względu na wystarczającą ilość wyjść układu własność ta nie została wykorzystana w przeprowadzanej implementacji.

Aby zmniejszyć czas inicjalizacji algorytmu, można zoptymalizować proces generowania wartości inicjalizujących stan wewnętrzny szyfru. Przedstawiony w pracy sposób pozwala zmniejszyć czas potrzebny do inicjalizacji algorytmu do 100 taktów zegara, jednakże taka zmiana spowoduje wydłużenie okresu zegara, co będzie skutkowało obniżeniem wydajności przeprowadzonej implementacji.

Przeprowadzona implementacja algorytmu znajdującego się w portfolio profilu programowego konkursu eSTREAM pokazała, że nie tylko algorytmy projektowane do wykorzystania w realizacjach sprzętowych mogą uzyskiwać wysoką wydajność generowania strumienia klucza. Warto jednak pamiętać, że algorytmy profilu sprzętowego musiały spełniać także kryteria niskiego zapotrzebowania nie tylko na elementy logiczne, ale także powinny umożliwiać wydajną pracę w układach o niewielkim zapotrzebowaniu na energię elektryczną.

Praca naukowa finansowana ze środków na naukę w latach 2009-2011 jako projekt rozwojowy Nr O R00 0043 07.

Artykuł wpłynął do redakcji w dniu 07.02.2011 r. Zweryfikowaną wersję po recenzji otrzymano w kwietniu 2011 r.

LITERATURA

[1] eSTREAM Phase 3 Candidates http://www.ecrypt.eu.org/stream/phase3list.html

[2] The eSTREAM portfolio rev.1 http://www.ecrypt.eu.org/stream/portfolio_revision1.pdf

[3] C. Berbain, O. Billet, A. Canteaut, N. Courtois, H. Gilbert, L. Goubin, A. Gouget, L. Granboulan, C. Lauradoux, M. Minier, T. Pornin, H. Sibert, *SOSEMANUK, a fast software-oriented stream cipher*, http://www.ecrypt.eu.org/stream/p3ciphers/sosemanuk/sosemanuk_ p3.pdf

[4] Y. Tsunoo, T. Saito, M. Shigeri, T. Suzaki, H. Ahmadi, T. Eghlidos, S. Khazaei, *Evaluation of SOSEMANUK With Regard to Guess-and-Determine Attacks*, 2006-01-02, http://www.ecrypt.eu.org/stream/papersdir/2006/009.pdf

[5] M.E. Hellman, *A cryptanalytic time-memory trade-off*, „IEEE Transactions on Information Theory", 26(4), 1980, 401–406.

[6] P. Hawkes, G. Rose, *Selected Areas in Cryptography – SAC 2002*, tom 2595, *Lecture Notes in Computer Science*, Springer-Verlag, 2002, 37–46.

[7] D. Watanabe, A. Biryukov, C. De Canni'ere, *A distinguishing attack of SNOW 2.0 with linear masking method. Selected Areas in Cryptography 2003*, tom 3006 *Lecture Notes in Computer Science*, Springer-Verlag, 2003, 222–233.

[8] K. Skahill, *Język VHDL*, wyd. 2, WNT Warszawa 2004.

[9] http://www.ecrypt.eu.org/stream/phase3perf/2007a/pentium-4-a/

[10] D. Hwang, M. Chaney, S. Karanam, N. Ton, K. Gaj, *Comparison of FPGA-Targeted Hardware Implementations of eSTREAM Stream Cipher Candidates*, luty 2008,
http://volgenau.gmu.edu/ kgaj/publications/conferences/GMU_SASC_2008.pdf

K. KACZYŃSKI

Hardware implementation of SOSEMANUK stream cipher

Abstract. In the paper, implementation of SOSEMANUK stream cipher in FPGA structure Altera Stratix II was described. Specification and security of algorithm was also presented. Analysis of implementation possibility, resources usage and efficiency of SOSEMANUK FPGA implementation was made. Paper contains comparison of obtained results with other algorithms' implementations of eSTREAM contest hardware profile and with software implementation made by authors of SOSEMANUK.

Keywords: stream cipher, hardware implementation, cryptoanalysis of stream algorithms