Sprzętowo-zorientowane konstrukcje szyfrów strumieniowych

Marcin Rogawski
PROKOM Software SA
rogawskim@prokom.pl

Streszczenie

Szyfry strumieniowe znajdują zastosowanie w różnych dziedzinach telekomunikacji. Wszędzie tam, gdzie krytyczną, oprócz poufności, cechą transmisji jest brak propagacji błędów w komunikacji, stosuje się właśnie ten rodzaj algorytmów kryptograficznych. Rozwiązania i technologie komercyjne, wymagające zastosowania szyfrów strumieniowych, nie korzystają z żadnego zatwierdzonego standardu. Bluetooth, WEP, GSM zapewniają poufność przesyłanych danych za pomocą słabych, jak na dzisiejsze możliwości kryptoanalizy, algorytmów takich jak: E0, RC4, A5. Szyfry strumieniowe mogą zostać zastąpione z powodzeniem rozwiązaniami blokowymi w trybie strumieniowym.

Szyfrowanie blokowe ma wiele zalet - jest uniwersalne: implementacyjnie i funkcjonalnie (różne tryby pracy). Konkursy AES, NESSIE i CRYPTREC, dotyczyły przede wszystkim algorytmów blokowych i pozwoliły na opracowanie nowych technik kryptoanalitycznych, umożliwiających na bardzo wnikliwe oceny każdego nowego szyfru. Poza tym wypracowane zostały metody budowania szyfrów udowadnialnie bezpiecznych. Na dodatek bardzo atrakcyjnym wydaje się stosowanie ogólnie uznanego i przyjętego za bezpieczny standardu FIPS197. Co więc z szyframi strumieniowymi?

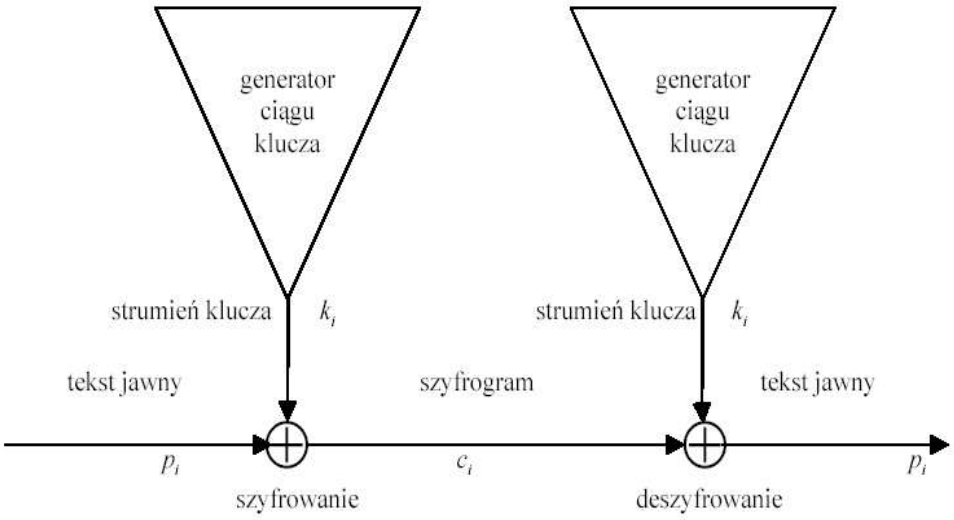
Projekt eSTREAM, rozpoczął się w roku 2004 i będzie trwał do roku 2008. Jego celem jest wybranie algorytmu strumieniowego, który mógłby w przyszłości stać się standardem. Algorytm, który zostanie zwycięzcą konkursu musi spełniać bardzo dużo wymagań (m.in. dotyczące szybkości, efektywności i dużej elastyczności implementacji), ale przede wszystkim musi być dużo lepszy niż AES, aby w ogóle był stosowany. W ramach projektu oceniane są dwa profile algorytmów – rozwiązania zorientowane sprzętowo i programowo. Pojawiło się już wiele komentarzy i publikacji na temat bezpieczeństwa, efektywności, szybkości i elastyczności zgłoszonych do konkursu algorytmów.

Celem autora artykułu jest przedstawienie cech szyfrów strumieniowych, które wpływają na ich możliwości sprzętowych implementacji.

1.1. Szyfry strumieniowe ...

są to algorytmy kryptograficzne, które przekształcają bit po bicie (znak po znaku, słowo po słowie) tekst jawny p_i na szyfrogram c_i przy użyciu klucza k_i. Długość klucza musi być przynajmniej taka jak długość tekstu jawnego. Najprostsza implementacje szyfru strumieniowego przedstawia poniższy rysunek.

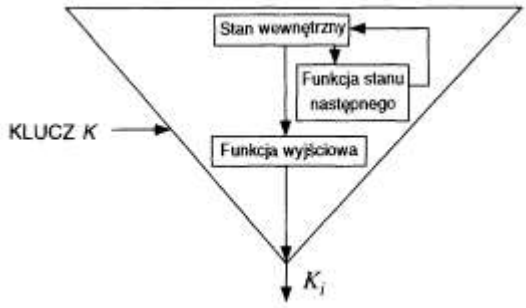
Rys



Rys. 1: sposób działania szyfru strumieniowego

Szyfry strumieniowe zbudowane są z trzech elementów:

- pamięci przechowującej stan wewnętrzny,
- funkcji wyznaczającej następny stan wewnętrzny,
- funkcji filtrujacej stan wewnętrzny i produkującej wyjściowy strumień klucza.



Rys. 2: ogólna budowa szyfru strumieniowego.

1.2. Czego oczekuje się od szyfrów strumieniowych

W dziedzinie zabezpieczania danych za pomocą kryptografii symetrycznej sytuacja jest następująca: istnieje zatwierdzony standard szyfrowania danych za pomocą algorytmu blokowego (FIPS197). Jest on powszechnie znany i stosowany. Ma bardzo elegancką specyfikacje. Bezpieczeństwo pomimo wielu prób kryptoanalitycznych, nadal jest wystarczające do zapewnienia poufności wrażliwych danych. Wyniki implementacji programowych oraz sprzętowych znakomite, a wiele prac wskazuje na jego elastyczność i efektywność. Bardzo dużą zaletą jest możliwość używania go w trybie strumieniowym – zastosowanie w środowiskach z silnymi restrykcjami na propagacje błędów. Pojawia się pytanie: Czy szyfry strumieniowe to jedynie ciekawa ale niepotrzebna alternatywa dla algorytmu AES ? Jakie są oczekiwania podmiotów komercyjnych względem szyfrów ? Czy któreś wymaganie jest trudne do spełnienia dla AES i jednocześnie łatwe dla szyfru strumieniowego ?

W dzisiejszych czasach specyfika zastosowań algorytmów kryptograficznych jest tak bardzo szeroka, że każdy nowy algorytm musi oprócz, bezpieczeństwa posiadać dobre własności implementacyjne.

Możliwości implementacyjne każdego nowego szyfru symetrycznego oceniane są wiec pod względem:

- szybkości (czas inicjalizacji, re-inicjalizacji, przepustowość)
- efektywności (czasami zwanej kosztownością) (wielkość implementacji + pobór mocy)
- elastyczności (różne platformy sprzętowe: 32 lub 64 bitowe procesory, 8 bitowe mikrokontrolery, FPGA, ASIC)

AES w trybie licznikowym i OFB spełnia wszystkie te wymagania.

Szyfry strumieniowe wydają się być niezbędne w środowiskach, które można określić ekstremalnymi z punktu widzenia projektantów systemów kryptograficznych. Chodzi tutaj przede wszystkim o:

- implementacje w środowiskach z bardzo mała bateria zasilająca (np. RFID)
- implementacje przy ograniczonej ilości zasobów (np. Smart Card)
- implementacje w specyficznych architekturach (8-bit)
- implementacje w środowiskach o bardzo dużej przepustowości (wiele Gigabitów na sekundę)

Na podstawie tych wymagań można wiec wyspecyfikować wymagania dla szyfru strumieniowego, który mógłby być alternatywnym standardem szyfrowania danych dla standardu AES. Poza oczywistą własnością, jaką jest odporność na wszelkiego rodzaju ataki kryptoanalityczne, istotnymi cechami są:

- duża elastyczność implementacji: od bardzo małych i wolniejszych, po większe i bardzo szybkie,
- jakość specyfikacji – chodzi o dużą elegancję i prostotę zapisu.

Twierdzenie mówiące o jakości implementacji danego algorytmu brzmiące “szybciej znaczy lepiej” ewoluowało w kierunku “elastyczniej, efektywniej i szybciej znaczy lepiej” ;-)

1.3. eSTREAM ....

Główna motywacją projektu eSTREAM jest identyfikacja szyfrów, które mogą zastąpić AES w zastosowaniach, w których wymagana jest duża przepustowość oraz w środowiskach o małej ilości zasobów. Każdy algorytm oceniany był i jest nadal w następujących kategoriach: kompaktowość (możliwość implementacji o minimalnej ilości zasobów), jakość szyfrowania (przepustowość, częstotliwość taktowania, ilość bitów generowanych w ciągu cyklu zegarowego), pobór mocy (środowiska bateryjno-akumulatorowe wymagają niskiego poboru prądu), elastyczność (skalowalność, potokowanie), prostota zapisu (jasność i zrozumiałość jest oczywista jeżeli weźmiemy pod uwagę szerokie zastosowanie). Kolejność wymienionych kategorii wydaje się być priorytetowa.

W marcu tego roku odbyła się wstępna selekcja algorytmów uczestniczących w projekcie eSTREAM. Poniższa tabela zawiera informacje na temat szyfrów ocenianych w trakcie projektu.

szyfr	*profil*	*Licencja*	*Stan wewnetrzny*	*Dlugosc parametrow*	*s-box*	*Status prac*	*uwagi*
ABC	S	nie	160 + KE(1024)	128/128		Not ok	Złamany – złożoność ataku 2^80, niejasności w specyfikacji dotyczące KE (Key Expansion)
Achterbahn	H	nie	-	-	-	Not ok	Złamany – złożoność ataku 2^73
CryptMT	S	tak	-	-	-	Not ok	Brak licencji “free for all”
Decim	H	tak	-	-	-	Not ok	Złamany, 2^29 IV pozwala odtworzyć klucz
Dicing	S	nie	768	128/256	2k	Not ok	Zbyt duży stan wewnętrzny
Dragon	S	nie	192	128/192	16k	Not ok	Zbyt duże generowane sboxy
Edon80	H	tak	-	-	-	Not ok	Brak licencji “free for all”
F-FCSR	S & H	nie	-	-	-	Not ok	Złamany – mozliwosc odtworzenia klucza
Frogbit	S	tak	-	-	-	Not ok	Brak licencji “free for all”
Grain	H	nie	160	80/64	0	OK	
Hc-256	S	nie	64k	256/256		Not ok	Zbyt duży stan wewnętrzny, operacja odejmowania 1024 bitowych liczb
Hermes8	S & H	nie	224	80/184	2k/logic	OK	Uwaga: używa sbox AESa
LEX	S & H	tak	-	-	-	Not ok	Brak licencji “free for all”
MAG	S & H	nie	-	-	-	Not ok	Atak typu DISTINGUISH
Mickey128	H	nie	-	-	-	OK	
Mir-1	S	nie	2432	128/64	2k/logic	Not ok	Zbyt duży stan wewnętrzny,
Mosquito	S & H	nie	128	96/128		OK	
NLS	S & H	nie	1184	256/256	8k	Not ok	Zbyt duży stan wewnętrzny,
Phelix	S & H	nie	352	256/128		OK	
Polar Bear	S & H	nie	168	128/<248	2k/logic	Not ok	Zbudowany na bazie AES i Rc4 – jedna runda AES wymaga zbyt dużo zasobów
Pomaranch	S & H	nie	184	128/144	4k/logic	Not ok	Niejasny status – wiele specyfikacji
Py	S	nie	10400	256/128	0	Not ok	Zbyt duży stan wewnętrzny,
Rabbit	S & H	tak	-	-	-	Not ok	Brak licencji “free for all”
Slasa20	S	nie	512	256/64	0	Not ok	Zbyt duży stan wewnętrzny,
Sfinks	H	nie	256	80/80	64k/logic	OK	
Sosemanuk	S	nie	512	128/256			Zbyt duży stan wewnętrzny,
SSS	S & H	nie	-	-	-	Not ok	Złamany przez J.Deamen w sposób praktyczny (10s na PC)
Trbdk3yaea	S & H	tak	-	-	-	Not ok	Brak licencji “free for all”
Trivium	H	nie	288	80/80	0	OK	
Tsc-3	H	nie	-	-	-	Not ok	Atak praktyczny – 4min na PC
Vest	H	tak	-	-	-	Not ok	Brak licencji “free for all”
Wg	H	nie	-	-	-	Not ok	Złamany
Yamb	S & H	nie	>3k	256/128	0	Not ok	Zbyt duży stan wewnętrzny,
Zk_Crypt	H	tak	-	-	-	Not ok	Brak licencji “free for all”

Spośród 34 algorytmów, aż 27 nie będzie rozpatrywanych w dalszej fazie projektu. 10 algorytmów zostało odrzuconych ze względów implementacyjnych. Większość implementacyjnych własności, które były bardzo niepożądanymi, dotyczyło zbyt dużego stanu wewnętrznego. 352 bity na stan wewnętrzny algorytmu Phelix jest wielkością akceptowalna, natomiast 512 bitów Salsa20 i Sosemanuk to już zbyt duży stan wewnętrzny.

Algorytm Hc-256 został odrzucony także z powodu jednej z operacji – odejmowania liczb 1024-bitowych. Algorytm ABC nie będzie dalej rozpatrywany, ponieważ wynikało zbyt dużo niejasności z jego specyfikacji.

2. Implementacje algorytmów strumieniowych

Oczywiście nie jest możliwym omówienie wszystkich operacji składowych występujących w najbardziej znanych i popularnych algorytmach strumieniowych. Dosyć liczną grupę stanowią jednak szyfry, w których składowymi operacji są:

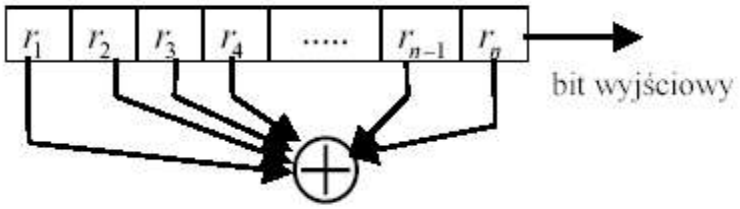
- operacje logiczne (bardzo łatwe w implementacji – bezpośrednia interpretacja fizyczna w układzie),
- operacje arytmetyczne (operacje, które mogą być bardzo skomplikowane i bardzo uciążliwe w realizacji jeżeli parametry tych operacji są z ciał o dużej liczbie elementów. W zasadzie łatwe w implementacjach są dodawania czynników do 8-bitow, mnożenie do 4-bitów),
- podstawienia – sboxy (bezpieczeństwo: “im większe tym lepsze”, implementacje: “im mniejsze tym lepsze”),
- przesunięcia i rotacje (bardzo łatwe w implementacji),

Bardzo popularnym elementem składowym szyfrów strumieniowych jest LFSR.

LFSR

Liniowy rejestr przesuwny ze sprzężeniem zwrotnym (ang. Linear feedback shift register) o n-stanach składa się z rejestru przesuwającego $R = (r_n, r_{n-1}, ..., r_1)$ i rejestru sprzężeń $T = (t_n, t_{n-1}, ..., t_1)$. Każdy element t_i i r_i reprezentuje jeden bit. W każdym kroku bit r_i jest przyłączany do łańcucha klucza, bity $r_n, ... r_2$ są przesuwane w prawo, a nowy bit obliczany z wartości T i R jest wprowadzany z lewej strony rejestru. Następny stan rejestru $R' = (r'_n, r'_{n-1}, ... r'_1)$ jest obliczany ze wzoru

$$r'_1 = r_{i+1} \text{ dla } i = 1, ...,n-1$$



$$r'_n = t_1 * r_{n-1} \text{ xor.... } t_n * r_1$$

Rys. 3: Przykład LFSR

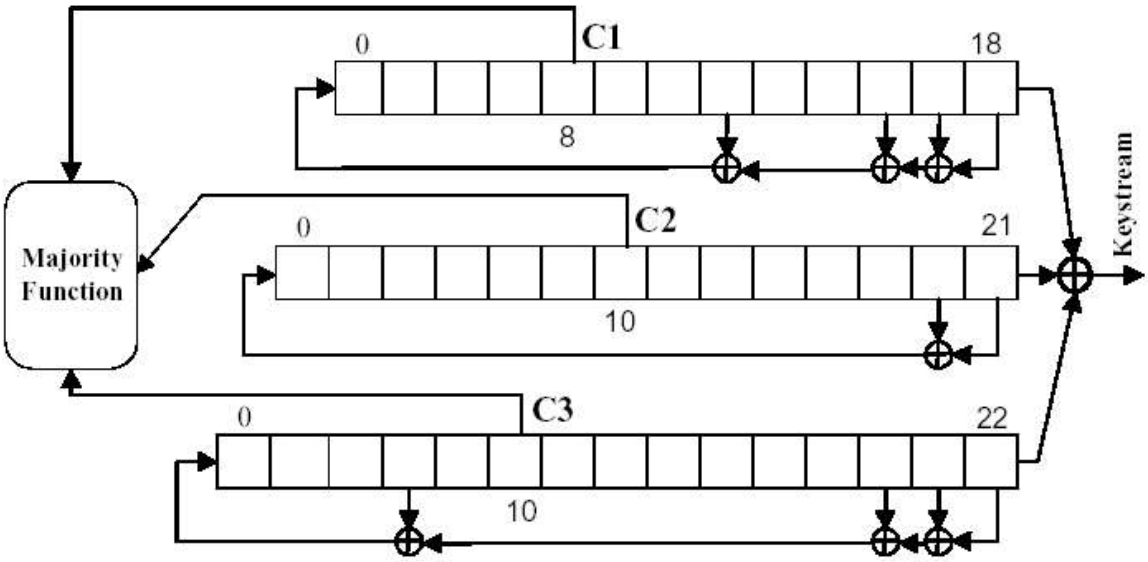
Liniowy rejestr przesuwny ze sprzężeniem zwrotnym w odniesieniu do implementacji sprzętowych charakteryzuje się:

- bardzo efektywną implementacją,
- zmianą tylko jednego bitu w jednej rundzie,
- wadą LFSR jest operowanie na bitach – duże rozdrobnienie i ograniczenia przepustowości,
- dużą efektywnością,
- elastycznością i prostotą,

A5 – LFSR + nieregularne taktowanie ...

Najbardziej znanym, a także najpowszechniej używanym szyfrem strumieniowym, w ostatnich latach jest algorytm A5/1, używany do zabezpieczania telefonii komórkowej GSM.

A5/1 zawiera trzy krótkie rejestry przesuwne (LFSR) o długości 19, 22, 23, oznaczone odpowiednio C1, C2, C3. Wszystkie LFSR-y mają wielomiany pierwotne, które definiują sposób wyznaczania nowej wartości bitu na pozycji zerowej w LFSR. Strumień klucza generowany przez A5/1 jest xor-em wyjść tych trzech wymienionych rejestrów zostało przedstawione na rysunku poniżej. Szyfrowanie polega na dodawaniu modulo 2 strumienia danych jawnych do strumienia klucza wychodzącego z modułu realizującego A5.



Rys. 4: Schemat działania A5/1

Pozycje taktujace (C1, C2, C3)	***Taktowanie w następnym kroku***
(0, 0, 0)	(C1, C2, C3)
(1, 1, 1)	(C1, C2, C3)
(0, 0, 1)	(C1, C2, -)
(1, 1, 0)	(C1, C2, -)
(0, 1, 1)	(-, C2, C3)
(1, 0, 0)	(-, C2, C3)
(1, 0, 1)	(C1, -, C3)
(0, 1, 0)	(C1, -, C3)

Tab. 1: nieregularne taktowanie LFSR

W algorytmie A5, jak w każdym innym strumieniowym szyfrze, wyróżniamy więc:
- stan wewnętrzny (zawartość rejestrów)
- funkcje wyznaczająca następny stan (funkcja większościowa – nieregularne taktowanie, xor odczepów rejestrów)
- funkcja filtrująca (xor wyjść, każdego z rejestrów)

Podstawowa implementacja sprzętowa modułu kryptograficznego wspierającego szyfrowanie algorytmem A5 umożliwia wygenerowanie kolejnego stanu wewnętrznego oraz pojedynczego bitu wyjściowego w ciągu jednego cyklu zegarowego. Po zapisie wartości do rejestrów wewnętrznych realizowane są równolegle funkcje kombinacyjne. Funkcja większościowa, na podstawie wartości na bitach odpowiednio 8, 10, 10 w rejestrach C1, C2, C3 – wyznacza sposób taktowania rejestrów w następnym kroku działania algorytmu. Suma modulo 2 wybranych połączeń (odczepów) rejestru wylicza wartość pierwszego bitu w każdym składowym rejestrze. Rejestry składowe realizują dodatkowo operacje przesuwania, dlatego też przerzutniki, z których składa się rejestr przesuwny musza być połączone ze sobą. Ostatnia funkcja kombinacyjna wykonywana równolegle podczas tego pojedynczego cyklu to operacja wyznaczenia bitu wyjściowego – czyli w tym wypadku operacja xor na wyjściach wszystkich rejestrów. Cały cykl kończy zapis do elementów pamięciowych, czyli w tym przypadku do rejestrów. Zapis oczywiście odbywa się zgodnie z tabelka, według której nie zawsze pomimo wyznaczenia następnej wartości pojedynczego LFSR ta wartość jest zapisywana – idea nieregularnego taktowania.

Aby operacja była oczywiście poprawna na wyjściach każdej z funkcji boolowskiej wykonywanej równolegle muszą być ustabilizowane dane wyjściowe. Wyznaczenie okresu, po jakim to następuje, pozawala określić maksymalną częstotliwość taktowania wytworzonego modułu kryptograficznego.

Co charakteryzuje taka podstawową implementację ?

- duża przepustowość,
- duża maksymalna częstotliwość taktowania,
- kompaktowość implementacji,

układ	*zajętość*	*Częstotliwość*	*przepustowość*
Virtex – II	32 LC	188 Mhz	188 Mb/s
Flex 10KE	157 LC	158,7 Mhz	39,7 Mb/s
Cyclone	145 LC	267 Mhz	267 Mb/s

Różne wyniki implementacji wynikają przede wszystkim z wyboru różnych układów. Biorąc pod uwagę tylko wyniki implementacyjne można dojść do wniosku, że algorytm A5/1 wydaje się bardzo szybkim i bardzo tanim algorytmem. Przepustowość rzędu kilkudziesięciu Mb/s jest aż nadto wystarczająca do zastosowań w telefonii GSM. Największym problemem algorytmu A5/1 jest jednak jego skalowalność. Aby osiągnąć wyższe przepustowości należałoby albo zwiększać częstotliwość sygnału zegarowego lub zrównoleglać działanie algorytmu. Moduł A5\1 generowałby wówczas w ciągu jednego taktu wytwarzany by strumień 2, 3, 4 ... lub większej ilości bitów. Zwiększenie częstotliwości to problem wyboru lepszego układu – szybszego, opartego o nowszą technologie. Zwiększanie przepustowości w ten sposób wedle "prawa Moore'a" jest dosyć ograniczone, a jego zwiększanie nie jest tak duże jak zwiększanie możliwości układów scalonych poprzez zmiany architektoniczne.

Prześledzimy możliwości zmian architektonicznych w algorytmie A5/1. Zmiany będą polegały, tak jak wcześniej już zaznaczyliśmy, na zrównolegleniu niezbędnych operacji – tak żeby na wyjściu po każdym cyklu były 2 bity zamiast jednego. W ciągu jednego cyklu każdy z rejestrów będzie musiał więc wykonać przesunięcie o 0, 1, 2 pozycji. Muszą wiec powstać odpowiednie połączenia. Ilość przesunięć będzie zależna od funkcji większościowej. Na tym etapie nie zajmujemy się ta funkcja, przyjmiemy tylko, że na wejściu logiki obsługującej pojawi się szyna konfigurująca zachowanie rejestru, zamiast wcześniej tylko jednej linii. Najlepiej zobrazuje to pseudokod.
{znak $ oznacza operacje xor}

Podstawowa implementacja

```
{r0, r1, ... r17, r18}                  /* rejestr przesuwny */
maj_func                                /* linia z funkcji większościowej decydująca czy następuje
                                           przesuniecie czy tez nie. */

if (maj_func = 1)
        {r1, r2, ... r18}    <= {r0, ... r17}
        r0 = r18 $ r17 $ r16 $ r13
else
        {r0, ..., r18}       <= {r0,.. r18}
```

Implementacja zrównoleglająca wykonanie 2 rund w ciągu jednego cyklu

```
{r0, r1, ... r17, r18}                  /* rejestr przesuwny */
maj_fun[0..1]                           /* wejście – które jest wyjściem funkcji większościowej
                                           i które decyduje o długości przesunięcia w rejestrze*/

switch ( maj_func[ ])

case "00"                               /* funkcja większościowa przesyła po linii informację, że
                                           rejestr nie będzie taktowany w dwóch kolejnych rundach*/
        {r0, ..., r18}       <= {r0,.. r18}


case "01"                               /* funkcja większościowa przesyła po linii informację, że
                                           rejestr będzie taktowany tylko raz i nie ważne jest czy w pierwszej
                                           z dwóch rund czy w drugiej – ważny jest stan końcowy operacji*/
        {r1, r2, ... r18}    <= {r0, ... r17}
        r0 = r18 $ r17 $ r16 $ r13


case "10"                               /* funkcja większościowa przesyła po linii informacje, że
```

```
                                        rejestr będzie taktowany dwa razy */

{r2, ... r18}          <= {r0, ... r16}
r1 = r18 $ r17 $ r16 $ r13
r2 = r17 $ r16 $ r15 $ r12              /*po pierwszym przesunięciu wartości z kolejnych pozycji
                                        przesuną się o jedną pozycje */
```

Ilość elementów logicznych potrzebnych do realizacji dwóch kolejnych wersji rejestru przesuwnego: pierwszej ,z maksymalnie pojedynczym przesunięciem, drugiej, z maksymalnie dwoma przesunięciami wzrosła ponad dwukrotnie. Możliwe jednak jest dalsze zwiększanie zrównoleglenia polegające na wykonaniu większej ilości rund w ciągu jednego taktu. Oczywiście będzie to bardzo kosztowne w elementy logiczne rozwiązanie – ale ciągle opłacalne.

Największą trudnością w skalowaniu implementacji sprzętowej algorytmu A5 stanowi funkcja większościowa. Zależność taktowania rejestrów, w zależności od zawartości powoduje znaczne skomplikowanie zrównoleglonej funkcji większościowej.

W wersji podstawowej funkcja wyglądać może tak:

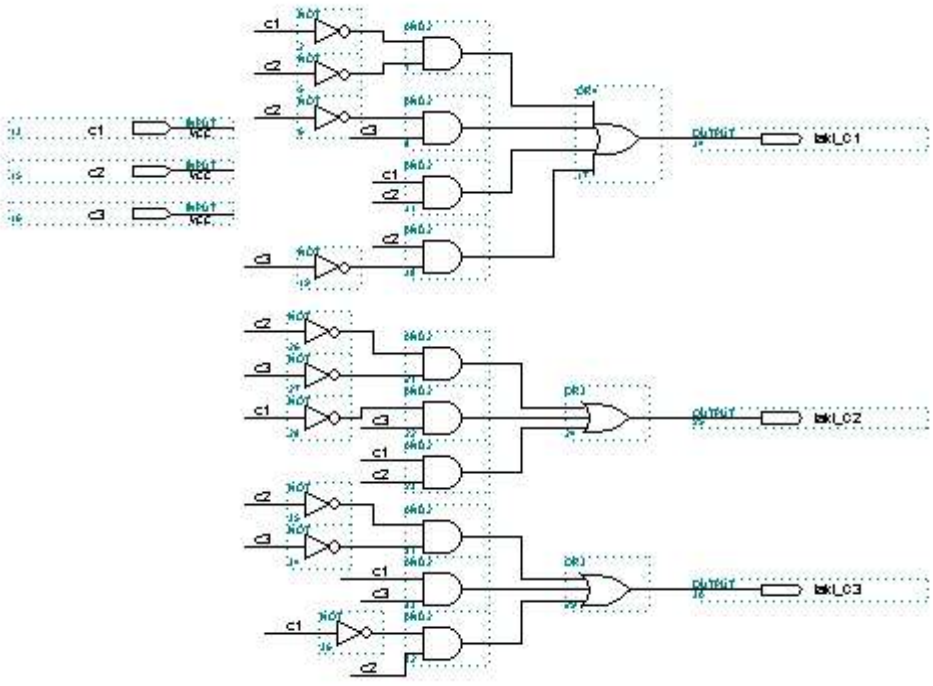
```
state = c1, c2, c3                      /* wartości odpowiednich bitów taktujących z 3 rejestrów*/
switch ( state )
        case 000 => maj_func[] = 111
        case 111 => maj_func[] = 111    /* w zależności od wartości taktowane są odpowiednie rejestry*/

        case 001 => maj_func[] = 110
        case 110 => maj_func[] = 110

        case 011 => maj_func[] = 011
        case 100 => maj_func[] = 011

        case 101 => maj_func[] = 101
        case 010 => maj_func[] = 101
end case;
```

Kod ten zostanie odwzorowany na strukturę logiczna na przykład na poniższą sieć bramek



Rys. 5: Funkcja większościowa w postaci funkcji kombinacyjnej

Zaledwie 21 bramek wystarcza na realizację całej funkcji większościowej.

W przypadku implementacji funkcji większościowej dla wersji zrównoleglonej sytuacja się komplikuje. Z jednej strony z każdego rejestru pobierać trzeba po dwie kolejne wartości z pozycji taktujących, a z drugiej nie wszystkie w danym momencie musza być potrzebne.

Pseudokod tej funkcji wyglądać może następująco

a1, a0 – dwa kolejne bity z pozycji sterujących rejestru 19 bitowego
b1, b0 – dwa kolejne bity z pozycji sterujących rejestru 22 bitowego
c1, c0 – dwa kolejne bity z pozycji sterujących rejestru 23 bitowego

```
state[]  =(a1,b1,c1);          /* rozpatrujemy najpierw potencjalny pierwszy takt*/

if ((state[] = 000) # (state[] = 111)) then
        2state[] = (a0, b0, c0);              /* w zależności czy był wykonany, kolejny bit z pozycji
                                                 sterującej, albo jest brany pod uwagę na aktualnej pozycji
                                                        sterującej albo nie */
        switch ( 2state[] )
                case 000 => takt[] = 101010      /* najbardziej znaczące bity z szyny takt[] oznaczają ilość
                case 111 => takt[] = 101010      taktów rejestru 19 bitowego C1 */
                case 001 => takt[] = 101001      /* środkowe dwa bity z szyny takt oznaczają ilość
                case 110 => takt[] = 101001      taktów rejestru 22 bitowego C2 */
                case 011 => takt[] = 011010      /* najmniej znaczące bity z szyny takt[] oznaczają ilość
                case 100 => takt[] = 011010      taktów rejestru 23 bitowego C3 */
                case 101 => takt[] = 100110
                case 010 => takt[] = 100110
end if;

if ((state[] = 001) # (state[] = 110)) then
        2state[] = (a0, b0, c1);

        switch ( 2state[] )
                case 000 => takt[] = 101001
                case 111 => takt[] = 101001
                case 001 => takt[] = 101000
                case 110 => takt[] = 101000
                case 011 => takt[] = 011001
                case 100 => takt[] = 011001
                case 101 => takt[] = 100101
                case 010 => takt[] = 100101
end if;

if ((state[] = 011) # (state[] = 100)) then
        2state[] = (a1, b0, c0);

        switch ( 2state[] )
                case 000 => takt[] = 011010
                case 111 => takt[] = 011010
                case 001 => takt[] = 011001
                case 110 => takt[] = 011001
                case 011 => takt[] = 001010
                case 100 => takt[] = 001010
                case 101 => takt[] = 010111
                case 010 => takt[] = 010110
end if;

if ((state[] = 101) # (state[] = 010)) then
        2state[] = (a0, b1, c0);

        switch ( 2state[] )
                case 000 => takt[] = 100110
                case 111 => takt[] = 100110
                case 001 => takt[] = 100101
                case 110 => takt[] = 100101
```

```
                case 011 => takt[] = 010110
                case 100 => takt[] = 010110
                case 101 => takt[] = 100010
                case 010 => takt[] = 100010
        end if;
```

Ilość zasobów potrzebnych do zrealizowania funkcji większościowej jest 16 większa od wersji podstawowej i nie będziemy przedstawiać tutaj sieci bramek (w przypadku zastosowania dekompozycji funkcjonalnej i oprogramowania DEMAIN dostępnego na stronie http://wwwzpt.tele.pw.edu.pl/ funkcję większościowa jest 10 razy bardziej zasobo-chłonna). Ścieżka krytyczna, czyli taką, przez którą sygnał cyfrowy najdłużej się propaguje wynosi 9 poziomów bramek i powoduje to znaczne obniżenie maksymalnej częstotliwości taktującej do 145 Mhz (układ Cyclone). Taka sytuacja powoduje, ze implementacja z podwójnym wykonaniem rundy algorytmu A5 powoduje minimalny wzrost przepustowości (290 Mb/s) przy czterokrotnym zwiększeniu ilości wymaganych zasobów.

Najbardziej niezorientowaną sprzętowo operacja algorytmu A5 jest nieregularne taktowanie. Powoduje ono komplikacje funkcji większościowej oraz operacji wyznaczania następnego stanu. Nieregularne taktowanie oczywiście ma znaczenie dla własności kryptograficznych algorytmu, ale czyni go także nieelastycznym.

Inna bardzo istotna sprawa jest dobór wielomianu pierwotnego. Od niego zależy definicja odczepów poszczególnych rejestrów, a w przypadku gdy wielomian ma pierwotny ma niezerowe współczynniki przy najwyższych potęgach wówczas utrudniona jest również skalowalność takiego algorytmu.

Nieregularne taktowanie maja także inne szyfry strumieniowe: Lili – 128 (algorytm, który zgłoszony był do konkursu NESSIE, ale nie zdobył tam uznania) oraz Mickey128, który w tym roku, pomimo podobnych problemów implementacyjnych jak w przypadku A5, został wybrany do grupy czterech algorytmów, które będą rozpatrywane w dalszej części projektu eSTREAM.

LFSR – tylko GF(2) ?

Algorytmy strumieniowe oparte o LFSR binarny są bardzo efektywne, ale ich implementacje charakteryzuje także oprócz dużej częstotliwości stosunkowo mała szybkość przetwarzania. Wśród dokonań najnowszej historii kryptografii strumieniowej pojawiło się wiele konstrukcji, które oparte były o LFSR bajtowy, słowowy (16 bitowe, a nawet 32 bitowe). W trakcie jednego cyklu zegarowego generowany wiec odpowiednio 8, 16 lub 32 bity.

Poniższa tabela zawiera wyniki implementacji przykładowych algorytmów opartych o bajtów i słowowy LFSR.

algorytm	*układ*	*zajętość*	*częstotliwość*	*przepustowość*
W7 (8 bitów)	Virtex	608 LC	96 Mhz	768 Mb/s
W7 (8 bitów)	Cyclone	627 LC	161,3 Mhz	322 Mb/s
Sober (8 bitów)	Cyclone	525 LC	267 Mhz	267 Mb/s
Snow (32 bity)	Cyclone	824 LC / 24k mem	156 Mhz	998 Mb/s
A5 (1 bit)	Cyclone	145 LC	267 Mhz	267 Mb/s

Z tabeli jasno wynika, ze zwiększenie długości wytwarzanego strumienia danych w implementacjach podstawowych powoduje wydatnie podniesienie szybkości przetwarzania.

Skalowalność implementacji tych algorytmów jest jednak jeszcze mniejsza niż w przypadku omówionego algorytmu A5\1.

eSTREAM

Po roku trwania projektu eSTREAM zakończyła się faza pierwsza oceny szyfrów strumieniowych. Wśród algorytmów rozpatrywanych, jako przyszły sprzętowy, strumieniowy standard szyfrowania danych szczególnie cztery konstrukcje zwróciły uwagę. Grain, Trivuim, Mickey128 i Phelix zakwalifikowały się do drugiej fazy projektu. Dwa pierwsze maja szczególne cechy orientacji sprzętowej i jeżeli nie pojawia się żadne, zmniejszające margines bezpieczeństwa, ataki kryptoanalityczne to wydaje się, że któryś z nich może zostać wybrany na nowy standard szyfrowania danych.

GRAIN ...

Twórcy algorytmu Grain, Marti Hell, Tomas Johanson i Wili Meier z Uniwersytetu Lund, oparli działanie swojego algorytmu na 2 rejestrach: liniowym i nie liniowym rejestrze przesuwnym (oba o długości 80 bitów). Funkcje wyznaczające następna wartość stanu wewnętrznego są dobrane w taki sposób, aby co najmniej 16 pierwszych bitów każdego z rejestrów nie uczestniczyło w operacji wyznaczenia kolejnej zawartości stanu wewnętrznego.

W przypadku rejestru LFSR funkcja

$$f(x) = 1 + x^{18} + x^{29} + x^{42} + x^{57} + x^{67} + x^{80}$$

definiuje sposób połączenia

$$s_{i+80} = s_{i+62} + s_{i+51} + s_{i+38} + s_{i+23} + s_{i+13} + s_i.$$

Zauważamy więc, ze w przypadku LFSR bity od s63 do s80 nie biorą udziału w wyznaczaniu kolejnej wartości. Można wiec ta część algorytmu zrealizować w postaci 18 rundowej w jednym cyklu zegarowym.

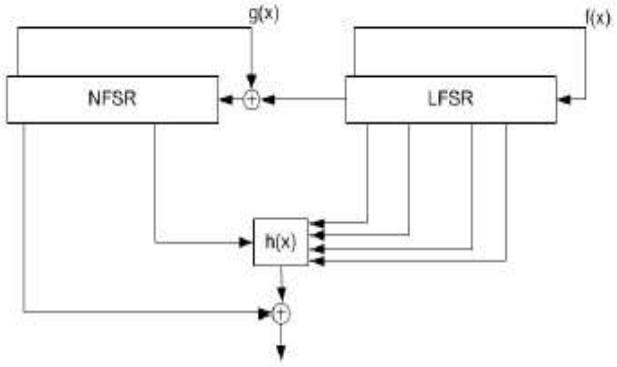
Podobnie jest z rejestrem NLFSR: funkcja g(x) definiuje sposób uaktualnienia stanu wewnętrznego rejestru przesuwnego.

$$\begin{aligned} g(x) = 1 &+ x^{17} + x^{20} + x^{28} + x^{35} + x^{43} + x^{47} + x^{52} + x^{59} + x^{65} + x^{71} + x^{80} + \\ &+ x^{17}x^{20} + x^{43}x^{47} + x^{65}x^{71} + x^{20}x^{28}x^{35} + x^{47}x^{52}x^{59} + x^{17}x^{35}x^{52}x^{71} + \\ &+ x^{20}x^{28}x^{43}x^{47} + x^{17}x^{20}x^{59}x^{65} + x^{17}x^{20}x^{28}x^{35}x^{43} + x^{47}x^{52}x^{59}x^{65}x^{71} + \\ &+ x^{28}x^{35}x^{43}x^{47}x^{52}x^{59}. \end{aligned}$$

Na jej podstawie budujemy sposób połączeń NLFSR.

$$\begin{aligned} b_{i+80} = s_i &+ b_{i+63} + b_{i+60} + b_{i+52} + b_{i+45} + b_{i+37} + b_{i+33} + b_{i+28} + b_{i+21} + \\ &+ b_{i+15} + b_{i+9} + b_i + b_{i+63}b_{i+60} + b_{i+37}b_{i+33} + b_{i+15}b_{i+9} + \\ &+ b_{i+60}b_{i+52}b_{i+45} + b_{i+33}b_{i+28}b_{i+21} + b_{i+63}b_{i+45}b_{i+28}b_{i+9} + \\ &+ b_{i+60}b_{i+52}b_{i+37}b_{i+33} + b_{i+63}b_{i+60}b_{i+21}b_{i+15} + \\ &+ b_{i+63}b_{i+60}b_{i+52}b_{i+45}b_{i+37} + b_{i+33}b_{i+28}b_{i+21}b_{i+15}b_{i+9} + \\ &+ b_{i+52}b_{i+45}b_{i+37}b_{i+33}b_{i+28}b_{i+21}. \end{aligned}$$

W przypadku NLFSR w wyznaczaniu kolejnej kolejnego stanu wewnętrznego rejestru przesuwnego nie bierze udziału dosyć duża grupa najmniej znaczących bitów tego rejestru (od s64 do s80). Możliwe jest wiec wykonanie 16 rund w ciągu jednego cyklu zegarowego.



Rys.6: Schemat działania algorytmu Grain

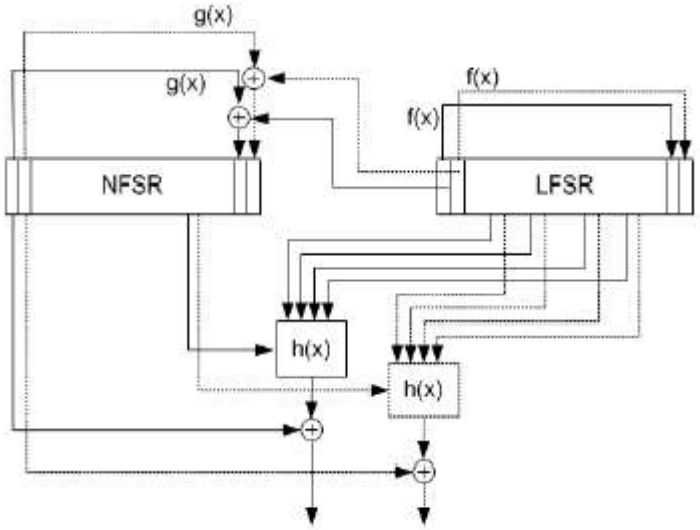
Także funkcja h(x) jest bardzo elastyczna.

$$h(x) = x_1 + x_4 + x_0x_3 + x_2x_3 + x_3x_4 + x_0x_1x_2 + x_0x_2x_3 + x_0x_2x_4 + x_1x_2x_4 + x_2x_3x_4$$

gdzie poszczególne elementy od x0 do x4 to wybrane wartości z obu rejestrów.

Poniższy rysunek pokazuje w sposób poglądowy sposób zwiększania wielkości strumienia wyjściowego, a także wykonania odpowiednio dużej liczby kroków w poszczególnych elementach algorytmu w pojedynczym cyklu

zegarowym.



Rys.7: Zrównoleglenie dwóch kolejnych wykonań algorytmu Grain
w ciągu jednego cyklu zegarowego

Wyniki implementacji poszczególnych wersji algorytmu Grain.

radix	***zajętość***	***częstotliwość***	***przepustowość***
1	203 LC	155 Mhz	155 Mb/s
2	221 LC	155 Mhz	310Mb/s
4	258 LC	154 Mhz	616 Mb/s
8	334 LC	154 Mhz	1,24 Gb/s
16	505 LC	154 Mhz	2,48 Gb/s

Implementacja sprzętowa algorytmu Grain jest wiec bardzo elastyczna, kompaktowa i może podołać wymaganiom postawionym przyszłemu standardowi strumieniowego szyfrowania danych. Wyniki prac badających bezpieczeństwo oraz możliwości praktycznego wykorzystania algorytmu pokazują bardzo duże możliwości algorytmu oraz jego jakość kryptograficzną.

Jego siłą jest jego konstrukcja oparta na:
- LFSR – z odpowiednio dobranymi odczepami,
- NLFSR – z również odpowiednio dobranymi odczepami,
- regularne taktowanie, które nie zaburza możliwości określenia stanu rejestru po czasie n-cykli,
- stosowanie funkcji uaktualniających stan oraz funkcji filtrujących, które są oparte o podstawowe operacje logiczne.

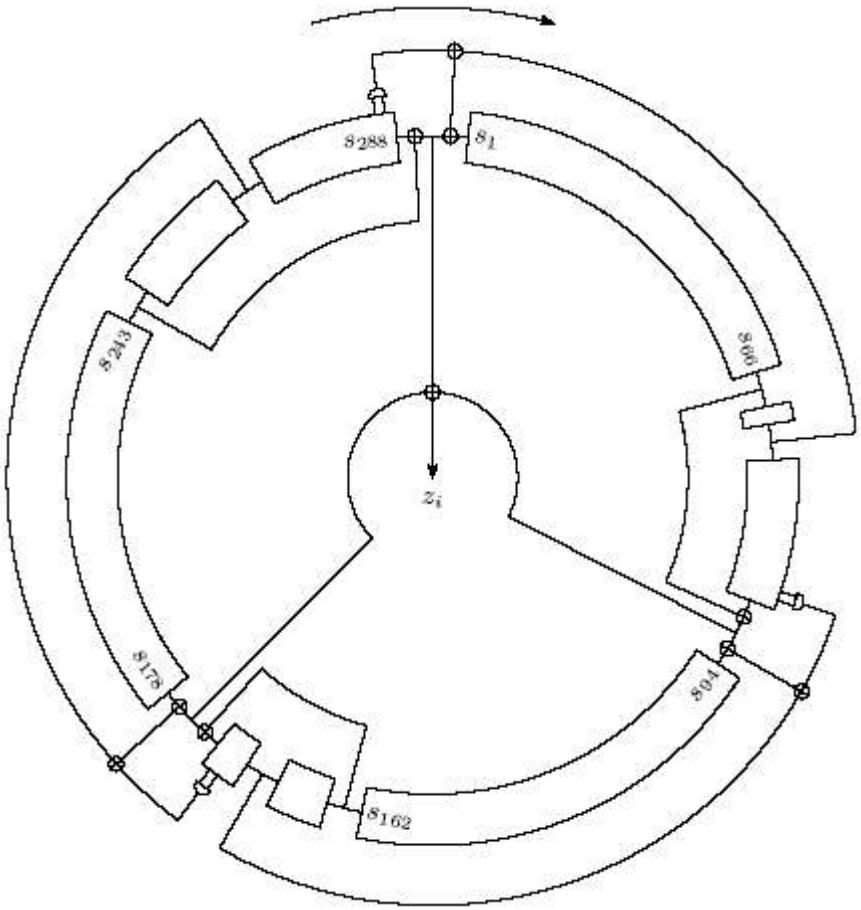
Trivium

Jest algorytmem wyspecyfikowanym przez Christophe De Canniere i Barta Preneel z Katholieke Universiteit Leuven. Myślą przewodnią towarzyszącą twórcom algorytmu było badanie polegające na upraszczaniu algorytmu najbardziej jak to tylko możliwe, bez poświęcania bezpieczeństwa, szybkości i elastyczności na rzecz prostoty algorytmu. Elegancki i prosty zapis powoduje większą uwagę środowisk kryptoanalitycznych, a ten fakt determinuje badanie algorytmu przez wielu ludzi. Stan wewnętrzny algorytmu składa się z tablicy {s1, ... , s288}. Podstawowa runda (funkcja filtrująca + uaktualniająca stan wewnętrzny) Trivium składa się z następujących operacji:

$$
\begin{aligned}
t_1 &\leftarrow s_{66} + s_{93} \\
t_2 &\leftarrow s_{162} + s_{177} \\
t_3 &\leftarrow s_{243} + s_{288} \\
z_i &\leftarrow t_1 + t_2 + t_3 \\
t_1 &\leftarrow t_1 + s_{91} \cdot s_{92} + s_{171} \\
t_2 &\leftarrow t_2 + s_{175} \cdot s_{176} + s_{264} \\
t_3 &\leftarrow t_3 + s_{286} \cdot s_{287} + s_{69} \\
(s_1, s_2, \ldots, s_{93}) &\leftarrow (t_3, s_1, \ldots, s_{92}) \\
(s_{94}, s_{95}, \ldots, s_{177}) &\leftarrow (t_1, s_{94}, \ldots, s_{176}) \\
(s_{178}, s_{279}, \ldots, s_{288}) &\leftarrow (t_2, s_{178}, \ldots, s_{287})
\end{aligned}
$$

*) + oznacza operacje xor

Algorytm składa się więc z trzech rejestrów przesuwnych, których stany wewnętrzne są wynikami działania funkcji uaktualniających. Parametry tych funkcji są tak wybrane, aby nie uczestniczyły w wyznaczaniu kolejnego stanu częściej niż raz na 64 rundy algorytmu.



Rys.8: Algorytm Trivium

Podstawowa implementacja algorytmu Trivium może być więc zapisana następującym pseudokodem.

```
T1 = s66 + s93
T2 = s162 + s177
T3 = s243 + s288

z1 = T1 + T2 + T3                                   /* wyznaczenie wyjścia algorytmu*/

T1 = T1 + s91*s92 + s171
T2 = T2 + s175*s176 + s264
T3 = T3 + s286*s287 + s69

{s1, s2, ... s92, s93} <= {T3, s1, s2, ..., s91, s92}          /*utworzenie nowego stanu*/
{s94, s95, ... s176, s177} <= {T1, s94, s95, ..., s175, s176}
{s178, s179, ... s287, s288} <= {T2, s178, s179, ..., s286, s287}
```

Dwie rundy równolegle

```
T1[1..0] = s[66..65] + s[93..92]                /*zamiast pojedynczych wartości – operacje na */
T2[1..0] = s[162..161] + s[177..176]            /* tablicach dwuwymiarowych*/
T3[1..0] = s[243..242] + s[288..287]

z1[1..0] = T1[1..0] + T2[1..0] + T3[1..0]       /*dwa bity na wyjściu po generacji dwóch taktów*/

T1[1..0] = T1[1..0] + s[91..90]    *s[92..91]    + s[171..170]
T2[1..0] = T2[1..0] + s[175..174] *s[176..175] + s[264..263]
T3[1..0] = T3[1..0] + s[286..285] *s[287..286] + s[69..68]

{s1, s2, ... s92, s93} <= {T3[0..1], s1, s2, ..., s90, s91}
{s94, s95, ... s176, s177} <= {T1[0..1], s94, s95, ..., s174, s175}
{s178, s179, ... s287, s288} <= {T2[0..1], s178, s179, ..., s285, s286}
```

64 rundy równolegle

```
T1[63..0] = s[66..3] + s[93..30]
T2[63..0] = s[162..99] + s[177..114]
T3[63..0] = s[243..180] + s[288..224]

z1[63..0] = T1[63..0] + T2[63..0] + T3[63..0]

T1[63..0] = T1[63..0] + s[91..28]    * s[92..29]    +  s[171..108]
T2[63..0] = T2[63..0] + s[175..112] * s[176..113] +  s[264..201]
T3[63..0] = T3[63..0] + s[286..223] * s[287..224] +  s[69..6]

{s1, s2, ... s92, s93} <= {T3[0..63], s1, s2, ..., s28, s29}
{s94, s95, ... s176, s177} <= {T1[0..63], s94, s95, ..., s112, s113}
{s178, s179, ... s287, s288} <= {T2[0..63], s178, s179, ..., s223, s224}
```

Wyniki implementacji algorytmu Trivium dla różnej długości strumienia wyjściowego w ciągu jednego taktu zawiera poniższa tabelka.

radix	*zajętość*	*Częst otliwość*	*przepustowość*
1	324 LC	160 Mhz	160 Mb/s
2	330 LC	157 Mhz	314Mb/s
4	330 LC	157 Mhz	628 Mb/s
8	330 LC	155 Mhz	1240 Gb/s
16	330 LC	155 Mhz	2,48 Gb/s
32	514 LC	155 Mhz	4,96 Gb/s
64	706 LC	154 Mhz	9,86 Gb/s

Wyniki implementacyjne algorytmu Trivium są fenomenalne. Zupełnie kompaktowy charakter, znakomita skalowalność. Moduł krypograficzny wspierający działanie Trivium może działać bardzo wydajnie zarówno w środowiskach 8, 16, 32, jak 64 bitowych.

Podstawowa runda składająca się z funkcji boolowskiej jest bardzo efektywna. Dokonuje zmiany 3- bitów w każdej rundzie przy wykorzystaniu 15 bitów na wejściu. Podobnie jak w przypadku algorytmu Grain istotnym faktem był dobór odpowiednich odczepów. Każdy bit używany jest zapisywany raz na okres dłuższy niż 64 cykle, odczytywany jest natomiast częściej.

Podsumowanie i wnioski ...

Twierdzenie mówiące o jakości implementacji danego algorytmu brzmiące “szybciej znaczy lepiej” ewoluowało w kierunku “elastyczniej, efektywniej i szybciej znaczy lepiej”. Kryptografia teoretyczna, która zajmuje się tworzeniem i łamaniem szyfrów dotrzymuje kroku zmianom technologicznym i potrzebom rynku. Duża ilość zastosowań mobilnych, środowisk z małym poborem mocy, implementacji na procesory 8-bitowe wymusiła poszukiwania alternatywy dla standardu AES.

Trwający od roku projekt eSTREAM zmienił raz na zawsze sposób postrzegania problemu projektowania algorytmów kryptograficznych przez konstruktorów szyfrów strumieniowych. Przepustowość i wysoka częstotliwość taktowania układu nie jest jedyną ważna cechą implementacji sprzętowych. Równie ważne są kompaktowość implementacji oraz ich elastyczność. Dziesięć spośród dwudziestu siedmiu odrzuconych w projekcie eSTREAM algorytmów, nie będzie podlegało dalszym procesom oceny dlatego, że posiadają cechy, które uniemożliwiają efektywną, elastyczną i szybką implementacje.

Podstawowym elementem wielu algorytmów strumieniowych jest liniowy rejestr przesuwny. Na przykładzie algorytmów A5\1 i Grain widzimy jednak, że wybór LFSR nie daje gwarancji, że algorytm będzie sprzętowo zorientowany. Algorytm A5\1 w wersji podstawowej jest bardzo efektywny i stosunkowo szybki. Jeżeli jednak miałby być zastosowany w środowisku o bardzo dużej przepustowości to wówczas jego szybkość nie będzie jego atutem. Implementacja ze zrównoleglonymi dwiema rundami tego algorytmu była szybszą, ale bardzo nieefektywna. Nie poruszając nawet kwestii bezpieczeństwa tego algorytmu algorytm ten nie spełniałby dzisiejszych wymagań dla szyfrów symetrycznych.

Algorytmy Trivium i Grain oparte są o schemat: niewielki stan wewnętrzny, uaktualniany jest bardzo efektywnie przez małe funkcje boolowskie. Filtrowanie stanu wewnętrznego jest wykonane w taki sposób, ze zdradza ono minimalną ilość wiedzy potrzebną kryptoanalitykowi do ataku na algorytm. Zapewnia to wysoką wydajność, elastyczność i przepustowość obu tym algorytmom.

Bibliografia

[1]. www.altera.com
[2].Bora Piotr – „Metody sprzętowej implementacji algorytmów kryptograficznych" – 2003r.
[3]. Boesgaard M., Pedersen T., Vesterager M., Zenner E. – "The Rabbit stream cipher –Design and Security Analysis" – dostęp ny na stronie http://www.cryptico.com/Files/Filer/SASC_Rabbit.pdf
[3]. Babbage S. - "Stream cipher – what does industry want ?"
[4]. Babbage S. and Matthew Dodd – Mickey128
[5]. Christophe De Cannič re and Bart Preneel – "Trivium"
[6]. Martin Hell, Thomas Johansson and Willi Meier - "Grain"
[7]. Kijowski M. - "Ataki korelacyjne na szyfry strumieniowe" - praca dyplomowa, WAT 2003r.
[8]. Kitsos P. - "Comparison of the hardware Implementation of stream ciphers"
[9]. Łuba Tadeusz – „Synteza Układów Cyfrowych"- 2003r.
[10]. Łuba Tadeusz – Wykłady dotyczące dekompozycji funkcjonalnej (www.zpt.pw.waw.pl)
[11]. Mańk Krzysztof. – cykl wykładów z przedmiotu „Szyfry Strumieniowe"
[12]. Menezes A.– „Handbook of Applied Cryptography"
[13]. Misztal Michał. - cykl wykładów z przedmiotu "Projektowanie szyfrów blokowych"
[14]. Rogawski M. - "Szyfry strumieniowe w strukturach FPGA" - ENIGMA 2005r.,
[15]. Szmidt J., Misztal M. - "Wstęp do kryptologii"-
[16]. Schneier B. - "Kryptografia dla praktyków"
[17]. The ECRYPT Stream Cipher Project – http://www.ecrypt.eu.org/stream/,
[18]. Thomas S., Anthony D., Berson T., Gong G. - "The W7 Stream Cipher Algorithm" - INTERNET DRAFT,
[19]. Tim Good, William Chelton and Mohamed Benaissa, "Review of stream cipher candidates from a low resource hardware perspective"
[20]. Frank K. Gürkaynak, Peter Luethi, Nico Bernold, René Blattmann, Victoria Goode, Marcel Marghitola, Hubert Kaeslin, Norbert Felber and Wolfgang Fichtner, "Hardware Evaluation of eSTREAM Candidates: Achterbahn, Grain, MICKEY, MOSQUITO, SFINKS, Trivium, VEST, ZK-Crypt"
[21]. Lejla Batina, Sandeep Kumar, Joseph Lano, Kirstin Lemke, Nele Mentens, Christof Paar, Bart Preneel, Kazuo Sakiyama and Ingrid Verbauwhede, "Testing Framework for eSTREAM Profile II Candidates"
[22]. Wobst Reinhardt. – „Kryptologia. Budowanie i Łamanie zabezpieczeń"
[23]. Żółtak Bartosz - "VMPC One-way function and Stream Cipher" - 2004r.
[24].http://wwwzpt.tele.pw.edu.pl/