Łukasz MATUSZEWSKI, Szymon ŁOZA, Mieczysław JESSA
POLITECHNIKA POZNAŃSKA, WYDZIAŁ ELEKTRONIKI I TELEKOMUNIKACJI,
ul. Polanka 3, 60-965 Poznań

Wykorzystanie funkcji skrótu do poprawy właściwości statystycznych ciągu liczb losowych

Mgr inż. Łukasz MATUSZEWSKI

Asystent na Wydziale Elektroniki i Telekomunikacji Politechniki Poznańskiej. Ukończył studia na tym samym wydziale w roku 2010. Jego zainteresowania to projektowanie urządzeń z wykorzystaniem układów reprogramowalnych w szczególności generatorów liczb losowych i układów synchronizacji.

e-mail: lukasz.matuszewski@et.put.poznan.pl

Mgr inż. Szymon ŁOZA

Absolwent Politechniki Poznańskiej. Studia inżynierskie ukończył na Wydziale Elektrycznym na kierunku Informatyka, specjalizacja Bezpieczeństwo Systemów Komputerowych. Studia magisterskie ukończył na Wydziale Informatyki na kierunku Informatyka, specjalizacja Systemy Wbudowane i Mobilne. Aktualnie doktorant na Wydziale Telekomunikacji Politechniki Poznańskiej. W obszarze jego zainteresowań znajduje się szeroko rozumiana kryptografia oraz bezpieczeństwo systemów wbudowanych.

e-mail: szymon.piotr.loza@gmail.com

Streszczenie

W pracy przedstawiono sposób wykorzystania funkcji skrótu, na przykładzie funkcji SHA-256 (ang. *Secure Hash Algorithm*), do poprawy właściwości statystycznych ciągów liczb losowych. W badaniach wykorzystano pakiet testów statystycznych NIST 800-22 do oceny właściwości wytwarzanego ciągu metodę restartów i test chi kwadrat, dzięki którym możliwe jest wykazanie, czy dany generator produkuje ciąg z przeważającymi elementami deterministycznymi czy niedeterministycznymi. Proponowany układ może być z powodzeniem zaimplementowany w każdym układzie FPGA (ang. *Field Programmable Gate Array*).

Słowa kluczowe: generator liczb prawdziwie losowych, generator pierścieniowy, kryptografia, funkcja skrótu, losowość i pseudolosowość.

Using a hash function to improve statistical properties of true random number sequences

Abstract

Random sequences play a key role in many contemporary cryptographic systems. To increase the efficiency and robustness to attacks, it is recommended to integrate a source of random numbers with a cryptographic system using these numbers. Unfortunately, the list of non-deterministic physical phenomena available in digital circuits is rather short and practically includes jitter and metastable states. It is expected that the generator produces sequences that pass all known statistical tests and that the sequences are unpredictable and attack resistant. A generator that satisfies these expectations is named a true random number generator (TRNG). This paper presents a novel method for producing random bits with the use of jitter observed in ring oscillators. The method uses a Galois ring oscillator introduced recently and the hash function. To assess the quality of output sequences, the statistical test suite prepared by National Institute of Standards and Technology (NIST) and the restart mechanism were used. The proposed system can be implemented in any Field Programmable Gate Array (FPGA).

Keywords: true random number generator, ring oscillator, cryptography, hash function, randomness and pseudo-randomness.

Dr hab. inż. Mieczysław JESSA

Adiunkt na Wydziale Elektroniki i Telekomunikacji Politechniki Poznańskiej, Katedra Systemów Telekomunikacyjnych i Optoelektroniki. Autor lub współautor ponad 120 publikacji, 15 patentów oraz kilkunastu rozwiązań konstrukcyjnych wdrożonych w krajowej sieci telekomunikacyjnej. Kierował ponad trzydziestoma projektami wykonanymi na rzecz podmiotów gospodarczych. Najważniejsze prace dotyczą synchronizacji sieci telekomunikacyjnej, zastosowań zjawiska chaosu oraz losowości i pseudolosowości.

e-mail: mjessa@et.put.poznan.pl

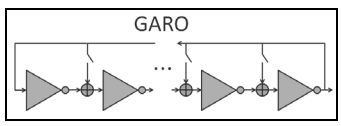
1. Wstęp

Liczby losowe są używane w wielu dziedzinach nauki. Są obecne w systemach identyfikacji, jako ziarno generatorów pseudolosowych i w symulacjach. Dotychczasowe źródła liczb losowych wykorzystują różne zjawiska fizyczne, przede wszystkim szumy występujące w układach elektronicznych, takich jak na przykład diody lawinowe [1]. Oferują one relatywnie małe przepływności (rzędu pojedynczych Mbit/s) i są mało odporne na ataki zewnętrze, co pociąga za sobą potrzebę instalowania rozbudowanych systemów detekcji i ochrony przed atakiem. Nie bez znaczenia jest też ciągły charakter pracy takich źródeł oraz brak możliwości integracji w jednym układzie scalonym, na przykład z systemem szyfrującym. Dlatego obecnie poszukuje się w układach cyfrowych takich zjawisk oraz takich metod, które pozwolą na wytworzenie ciągu losowego „na żądanie", o dużej przepływności, bez możliwości dostępu do elementów ciągu, to jest w tym samym układzie, w którym ciąg jest używany. Proponuje się wykorzystanie szybkozmiennych fluktuacji fazy (ang. *jitter*) generatorów konstruowanych z elementów układu reprogramowalnego albo stanów metastabilnych [2, 3]. Obecnie największe znaczenie praktyczne mają koncepcje wykorzystujące generatory pierścieniowe (ang. *Ring Oscillators*) oraz generatory pierścieniowe Galois (ang. *Galois Ring Oscillators*). W obu przypadkach źródłem losowości są szybkozmienne fluktuacje fazy wytwarzanych przebiegów [2, 3]. Binarny sygnał losowy jest otrzymywany w wyniku próbkowania sygnału generatora pierścieniowego o dużej częstotliwości przez generator o znacznie mniejszej częstotliwości. Oczekuje się, że ciąg wyjściowy nie tylko spełni wszystkie testy statystyczne ale także będzie wystarczająco losowy w tym sensie, że dla każdej pozycji ciągu, przy wielokrotnym użyciu tego samego generatora, otrzymamy zbliżoną liczbę zer i jedynek logicznych na tej pozycji. Ten parametr jest szacowany za pomocą mechanizmu restartów i testu chi-kwadrat [2, 4]. W artykule do wytworzenia ciągu losowego proponuje się użycie generatora pierścieniowego Galois (GARO) opisanego w pracy [3] oraz funkcji skrótu SHA-256. Generator GARO oraz funkcję SHA-256 zaimplementowano w tym samym układzie cyfrowym Virtex-5 (XL5VLX50T) [5]. W pracy pokazano, że otrzymany ciąg spełnia wszystkie testy statystyczne i przechodzi test chi-kwadrat dla mechanizmu restartów już od pierwszego wytworzonego bitu. W rezultacie użytkownik może wykorzystać każdy bit wytworzony przez generator, a nie co któryś jak w innych propozycjach [2]. Wynikiem jest wyższa przepływność otrzymanego strumienia bitów, która wyniosła średnio 26 Mbit/s. Ceną jest większa złożoność układu.

2. Opis układu

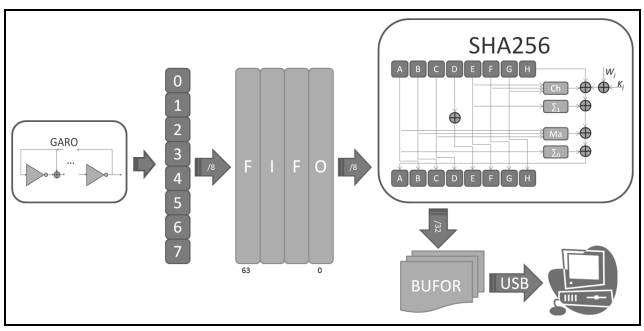
Do wytworzenia losowego ciągu binarnego wykorzystano oscylator pierścieniowy typu Galois [3] przedstawiony na rysunku 1.

Sygnał wyjściowy takiego generatora cechuje duża zawartość składników niedeterministycznych, jednak nie spełnia on większości testów statystycznych [4]. Generator GARO, podobnie jak jego odpowiednik LFSR (ang. *Linear Feedback Shift Register*) jest opisany za pomocą wielomianu charakterystycznego, który powinien być wielomianem pierwotnym nieparzystego stopnia. W generatorze GARO zamiast elementów pamiętających zostały zastosowane inwertery.



Rys. 1. Oscylator pierścieniowy typu Galois (GARO)
Fig. 1. Galois ring oscillator (GARO)

Do poprawy właściwości statystycznych pojedynczego źródła losowości proponuje się zastosowanie funkcji skrótu na przykładzie funkcji SHA-256. Schemat blokowy proponowanego rozwiązania przedstawiono na rysunku 2.



Rys. 2. Schemat blokowy generatora TRNG
Fig. 2. Block diagram of the TRNG

Generator zaimplementowano w układzie Virtex-5 (XL5VLX50T) [5]. W implementacji oscylatora pierścieniowego Galois wykorzystano wielomian:

$$f(x) = x^{31} + x^{27} + x^{23} + x^{21} + x^{20} + x^{17} + x^{16} + x^{15} + x^{13} + x^{10} + x^{9} + x^{8} + x^{6} + x^{5} + x^{4} + x^{3} + x + 1 . \quad (1)$$

Sygnał wyjściowy z generatora GARO jest próbkowany sygnałem o częstotliwości f_L równej 100MHz, który również taktuje pozostałe bloki układu. Tak powstały ciąg losowy jest formowany w bloki po 8 bitów. Następnie bajty są zapisywane w kolejce FIFO (ang. *First In First Out*) o długości 64 słów. Przygotowane w ten sposób losowe 512 bitów stanowi wejście funkcji SHA-256. Funkcja skrótu SHA-256 wraz z funkcjami SHA-384 i SHA-512 tworzy rodzinę funkcji SHA-2 [6].

Do implementacji funkcji SHA-256 w strukturach programowalnych posłużono się algorytmem [6]. W pierwszym kroku następuje dopełnienie wiadomości poprzez dodanie bitu 1 oraz dowolnej liczby bitów 0 doprowadzając do stanu $L \oplus 512 = 488$, gdzie L jest długością wiadomości, oraz dodanie L jako 64-bitowej liczby big-endian[1]. Później następuje podział 512-bitowych bloków na mniejsze 32-bitowe bloki oznaczane jako $M^{(i)}_j$ gdzie $j = 0, 1, \ldots, 63$, a $i = 0, 1, \ldots, N$, N jest numerem bloku w podzielonej wiadomości. Kolejnym krokiem jest wpisanie wartości początkowych $H^{(0)}$ - sekwencja 32-bitowych słów, otrzymanych jako części ułamkowe pierwiastka kwadratowego pierwszych ośmiu liczb pierwszych. Wartości $H^{(0)}$ są następujące:
$H^{(0)}_1$ = 6a09e667,
$H^{(0)}_2$ = bb67ae85,
$H^{(0)}_3$ = 3c6ef372,
$H^{(0)}_4$ = a54ff53a,
$H^{(0)}_5$ = 510e527f,
$H^{(0)}_6$ = 9b05688c,
$H^{(0)}_7$ = 1f83d9ab,
$H^{(0)}_8$ = 5be0cd19.

Następnym krokiem jest zastosowanie funkcji kompresującej do zaktualizowania rejestrów: *a, b, c, d, e, f, g, h*. Aktualizacja ta jest wyznaczana w 64 krokach od $j = 0$ do $j = 63$ i przebiega w następujący sposób:

$$\begin{aligned} T_1 &\leftarrow h + \textstyle\sum_1(e) + Ch(e, f, g) + K_j + W_j \\ T_2 &\leftarrow \textstyle\sum_0(a) + Maj(a, b, c) \\ h &\leftarrow g \\ g &\leftarrow f \\ f &\leftarrow e \\ e &\leftarrow d + T_1 \\ d &\leftarrow c \\ c &\leftarrow b \\ b &\leftarrow a \\ a &\leftarrow T_1 + T_2 \end{aligned} \quad (2)$$

gdzie:

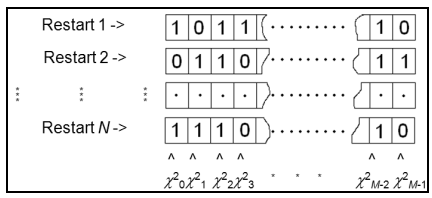
$$\begin{aligned} Ch(x, y, z) &= (x \wedge y) \oplus (\neg x \wedge z) \\ Maj(x, y, z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \\ \textstyle\sum_0(x) &= S2(x) \oplus S13(x) \oplus S22(x) \\ \textstyle\sum_1(x) &= S6(x) \oplus S11(x) \oplus S25(x) \\ \sigma_0(x) &= S7(x) \oplus S18(x) \oplus R3(x) \\ \sigma_1(x) &= S17(x) \oplus S19(x) \oplus R10(x) \end{aligned} \quad (3)$$

Sn – jest przesunięciem *n* bitów w prawo.
Rn – jest obrotem o *n* bitów w prawo.
W_j – bloki wiadomości wyznaczane są w następujący sposób:
1. dla pierwszych 16 bloków: $W_j = M_j^{(i)}$
2. dla pozostałych bloków: $W_j = \sigma_1(W_{j-2}) + W_{j-7} + \sigma_0(W_{j-15}) + W_{j-16}$
K_j - 32-bitowe słowa wyznaczone jako części ułamkowe pierwiastka trzeciego stopnia pierwszych 64 liczb pierwszych.
Następnie oblicza się *i*-tą pośrednią wartość $H^{(i)}$:

$$\begin{aligned} H^{(i)}_1 &\leftarrow a + H_1^{(i-1)} \\ H^{(i)}_2 &\leftarrow a + H_2^{(i-1)} \\ &\vdots \\ H^{(i)}_8 &\leftarrow a + H_8^{(i-1)} \end{aligned} . \quad (4)$$

Wynikiem działania algorytmu [6] jest skrót $H^{(N)}$ wiadomości M taki, że: $H^{(N)} = \{H^{(N)}_1, H^{(N)}_2, H^{(N)}_3, H^{(N)}_4, H^{(N)}_5, H^{(N)}_6, H^{(N)}_7, H^{(N)}_8\}$.

Po wygenerowaniu przez funkcję SHA-256 skrótu wiadomości złożonego z ośmiu 32-bitowych słów, wartości te zapisywane są w buforze i transmitowane poprzez USB (ang. *Universal Serial Bus*) do komputera klasy PC (ang. *Personal Computer*). W komputerze dokonuje się sprawdzenia właściwości statystycznych oraz przy wykorzystaniu metody restartów testuje się losowość wytworzonego ciągu.



Rys. 3. Schemat przedstawiający metodę restartów
Fig. 3. The restarts method

Mechanizm restartów został szerzej opisany w pracach [2], [4] oraz [7]. Polega on na wielokrotnym włączaniu generatora losowego z tymi samymi warunkami początkowymi i rejestrowaniu wynikowych bitów. Jego zastosowanie jest konieczne do prawidłowej oceny czy ciąg bitów wytwarzany przez generator TRNG, a spełniający wszystkie testy statystyczne, jest losowy, czy też spełnienie testów statystycznych zawdzięczamy zjawiskom, w których przeważają procesy deterministyczne. Jeżeli spełnienie testów zawdzięczamy zjawiskom z dużym udziałem czynników deterministycznych, to produkowane ciągi w każdym restarcie

[1] Forma zapisu danych, w której najbardziej znaczący bajt umieszczony jest jako pierwszy

będą zbliżone lub nawet takie same. W przeciwnym wypadku ciągi będą się różnić od siebie. Idea metody została przedstawiona na rysunku 3. W eksperymencie wykonano $N = 2048$ restartów, po $M = 19968$ bitów. Przeprowadzono M testów zgodności χ^2 dla ciągów o długości N bitów.

Jeżeli w danej chwili $m = 1, 2, \ldots, M$ wartość statystyki χ^2 jest mniejsza od wartości krytycznej to na danym poziomie istotności nie ma podstaw do odrzucenia hipotezy, że dany bit powstał w sposób niedeterministyczny, a więc losowy. Wartość statystyki χ^2 obliczono na podstawie wzoru [2, 4]:

$$\chi^2 = \frac{(N_0 - E_0)^2}{E_0} + \frac{(N_1 - E_1)^2}{E_1}, \qquad (5)$$

gdzie N_0 jest liczbą zer, E_0 liczbą oczekiwaną zer w ciągu o długości N bitów. Odpowiednio wartości N_1 i E_1 są liczbą jedynek i oczekiwaną liczbą jedynek w tym ciągu. W wynikach metody restartów poszukiwano największej wartości m, dla której wartość statystyki χ^2 jest większa od wartości krytycznej dla m, m-1 i m-2. Jeżeli zajdzie taki przypadek, liczbę $m + 1$ oznaczamy jako m_{min}. Zatem dla wartości $m \geq m_{min}$ test może nie być spełniony co najwyżej dla dwóch kolejnych bitów, co jest dopuszczalne dla ciągu losowego. Jednak jeżeli test chi-kwadrat nie jest spełniony dla trzech lub więcej kolejnych m, to jest to skutek dominacji pseudolosowości nad losowością [2, 4]. Wyniki eksperymentu przedstawia tabela 1.

Tab. 1. Wyniki metody restartów
Tab. 1. The results of the restarts metod

	GARO	***GARO + SHA256***
m_{min}	3	1

W trakcie badań przeprowadzono testy statystyczne na ciągu o długości 1 Gbit zarówno dla generatora GARO z użyciem funkcji SHA-256 jak i bez. Wyniki testów statystycznych z pakietu testów NIST 800-22 [8] zamieszczono w tabeli 2.

Tab. 2. Wyniki testów NIST 800-22 dla generatora GARO bez zastosowania funkcji skrótu i z funkcją skrótu
Tab. 2. The results of the NIST 800-22 tests for the GARO without and with a hash function

Rodzaj testu	***GARO***		***GARO + SHA256***	
	R_β	P_T	R_β	P_T
Test częstości	**0,0000**	**0,00000**	0,9856	0,36877
Blokowy test częstości	**0,0560**	**0,00000**	0,9867	0,18826
Test skumulowanych sum*	**0,0000**	**0,00000**	0,9878	0,28185
Test ciągów	**0,0000**	**0,00000**	0,9911	0,43932
Test na najdłuższy ciąg jedynak	**0,4370**	**0,00000**	0,9922	0,99890
Test stopnia macierzy binarnej	0,9870	0,23558	0,9856	0,49656
Test spektralny DFT	0,9880	**0,00003**	0,9900	0,29603
Test dopasowania nienakładających się wzorców *	**0,0000**	**0,00000**	0,9911	0,02860
Test dopasowania nakładających się wzorców	**0,0000**	**0,00000**	0,9900	0,02183
Test uniwersalny Mauera	**0,8560**	**0,00000**	0,9911	0,59554
Test przybliżonej entropii	**0,0000**	**0,00000**	0,9889	0,57033
Test błądzenia losowego*	------	------	0,9874	0,37148
Test wariancji błądzenia losowego**	------	------	0,9819	0,30815
Test serii*	**0,1890**	**0,00000**	0,9834	0,31908
Test złożoności liniowej	0,9910	0,43543	0,9889	0,85872

Po wykonaniu programu zostaje utworzony plik wynikowy zawierający statystyki pośrednie i tak zwane P-wartości (ang. *P-value*). Każda P-wartość jest prawdopodobieństwem, że statystyka wybranego testu przybierze wartości równe wartościom statystyki testu obserwowanego lub mniejsze. Jeżeli P-wartość jest równa poziomowi istotności testu β lub większa, to test jest spełniony. Jeżeli jest mniejsza, to test nie jest spełniony. W pakiecie NIST 800-22 standardowy poziom istotności wynosi $\beta = 0,01$. W pracy skorzystano z interpretacji zaproponowanej przez autorów pakietu NIST 800-22. Pierwsza z nich polega na określeniu proporcji R_β ciągów, które przechodzą dany test.

Drugie podejście polega na sprawdzeniu, czy rozkład P-wartości otrzymanych dla różnych ciągów jest rozkładem równomiernym w przedziale [0, 1]. W tym celu przedział jest dzielony na dziesięć podprzedziałów. Otrzymane P-wartości są zaliczane do odpowiedniego podprzedziału. Następnie jest wykonywany test χ^2 Pearsona. Dla testu jest obliczana P-wartość, oznaczana jako P_T. Jeżeli $P_r \geq 0,0001$, to uznajemy, że dany test jest zaliczony. W pracy, podobnie jak w pakiecie NIST 800-22, zakładamy, że ciąg empiryczny, tutaj o długości 10^9 bitów, podzielono na 1000 podciągów, każdy o długości 10^6, spełnia dany test statystyczny na poziomie istotności $\beta = 0,01$ wtedy i tylko wtedy, gdy jednocześnie $R_\beta > 0.9833$ oraz $P_T \geq 0,0001$. Znak * oznacza, że dany test składa się z wielu podtestów, a w tabeli zamieszczono rezultat najgorszy. Symbol ** wskazuje, że minimalna wartość potrzebna do zaliczenia tego testu jest różna od pozostałych i wynosi tutaj 0.9777.

3. Wnioski

Podstawowym wnioskiem z przeprowadzonych badań jest to, że zastosowanie funkcji skrótu SHA-256 dla generatora o relatywnie słabych właściwościach statystycznych pozwala uzyskać ciąg spełniający wszystkie testy z pakietu NIST 800-22. Wykorzystanie metody restartów sprawdzającej czy generator TRNG wytwarza ciągi losowe pokazuje, że dzięki zastosowaniu funkcji SHA-256 losowość ciągu zostaje zakumulowana. Uwarunkowane jest to między innymi tym, że zmiana pojedynczego bitu w wiadomości wejściowej funkcji SHA-256 powoduje zmianę przynajmniej połowy bitów wyjściowych [9]. Jest to tak zwany efekt lawinowy. Prezentowane rozwiązanie zajmuje tylko 8% zasobów logicznych układu Virtex-5 (XL5VLX50T) i przy częstotliwości taktowania równej 100 MHz dostarcza losowych bitów ze średnią prędkością około 26 Mbit/s.

Pracę sfinansowano z projektów 08/83/DSMK/4706 i 08/83/DSPB/4707.

4. Literatura

[1] Leśniewicz M.: Sprzętowa generacja losowych ciągów binarnych, Wojskowa Akademia Techniczna, 2009.
[2] Jessa M. Matuszewski L.: Producing random bits with delay-line based ring oscillators, International Journal of Electronics and Telecommunications, vol. 59, 1/ 2013, s. 41-50.
[3] Dichtl M, Golić J., D.: High speed true random number generation with logic gates only, CHES 2007, Lecture Notes in Computer Science, LNCS 4727, s. 45-62, 2007.
[4] Jessa M., Matuszewski L.: Enhancing the Randomness of a Combined True Random Number Generator Based on the Ring Oscillator Sampling Method, ReConFig'2011, Nov. 30 – Dec. 2, 2011, s. 274-279.
[5] http://www,xilinx,com/support/documentation/virtex-5.htm
[6] Descriptions of SHA-256, SHA-384, and SHA-512 http://csrc.nist.gov/groups/STM/cavp/documents/shs/sha256-384-512.pdf
[7] Jessa M., Jaworski M.: Randomness of a combined TRNG based on the ring oscillator sampling method, ICSES'10, Sept. 7-10, 2010, s. 323-326.
[8] Rukhin A., Soto J., Nechvatal J., Smid M., Barker E, Leigh S., Levenson M., Vangel M., Banks D., Heckert A., Dray J., Vo S.: A statistical test suite for random and pseudorandom number generators for cryptographic applications, NIST special publication 800-22, Revised: April 2010, USA, http://csrc.nist.gov/rng/
[9] Agarwal S., Rungta A., Padmavathy R., Shankar M., Rajan N.: An Improved Fast and Secure Hash Algorithm, Journal of Information Processing Systems, Vol. 8, No. 1, March 2012 s. 119-132.

otrzymano / received: 09.04.2014
przyjęto do druku / accepted: 02.06.2014 *artykuł recenzowany / revised paper*