Volodymyr KHOMA [1, 2], Artur SMOLCZYK [1], Yaroslav RESHETAR [2]
[1] POLITECHNIKA OPOLSKA, INSTYTUT AUTOMATYKI I INFORMATYKI, ul. Gen. Sosnkowskiego 31, 45-233 Opole
[2] NARODOWY UNIWERSYTET "LWOWSKA POLITECHNIKA", KATEDRA OCHRONY INFORMACJI, ul. Kn. Romana 1, 79000 Lwów

Implementacja kryptoalgorytmu GOST do systemów wbudowanych

Prof. dr hab. inż. Volodymyr KHOMA

Jest profesorem nadzwyczajnym na Wydziale Elektrotechniki i Automatyki Politechniki Opolskiej. W 1990 r uzyskał stopień naukowy doktora nauk technicznych, a w 2001 r doktora habilitowanego. Dorobek naukowy obejmuje ponad sto pięćdziesiąt publikacji i patentów z zakresu ochrony informacji i cyfrowego przetwarzania sygnałów w systemach pomiarowych i informatycznych.

e-mail: v.khoma@po.opole.pl

Dr inż. Artur SMOLCZYK

Jest starszym wykładowcą na Wydziale Elektrotechniki, Automatyki i Informatyki Politechniki Opolskiej. Stopień naukowy doktora nauk technicznych uzyskał w 1993 roku. Dorobek naukowy obejmuje ponad 40 publikacji z zakresu teoretycznych podstaw informatyki, algorytmiki oraz cyfrowego przetwarzania sygnałów.

e-mail: a.smolczyk@po.opole.pl

Streszczenie

Artykuł poświęcono problemowi skutecznej implementacji w systemach wbudowanych blokowego symetrycznego algorytmu kryptograficznego GOST. Przeprowadzone badania programowej implementacji na platformie AVR algorytmu GOST wykazały lepszy wskaźnik wydajność/rozmiar kodu w porównaniu z innymi znanymi algorytmami. Analiza porównawcza implementacji kryptoalgorytmu GOST na bardziej zaawansowanych mikrokontrolerach (ARM) ujawniła ponadto mniejsze zapotrzebowanie na pamięć w porównaniu z najnowszym algorytmem AES przy porównywalnej wydajności.

Słowa kluczowe: kryptoalgorytm GOST, kryptografia lekka (lightweight-cryptography), procesory ARM, szyfrowanie w systemach wbudowanych.

Implementation of GOST algorithm in embedded systems crypto applications

Abstract

This paper discusses the implementation of a cryptographic algorithm GOST in common used 8-bit (AVR) and 32-bit (ARM) processors for embedded systems. The GOST algorithm has a Feistel network structure with 32-rounds and uses simple operations (Fig. 1), which are easily implemented in general purpose microcontrollers by system-level commands. In addition, the algorithm has no expansion key procedure, which is an advantage for lightweight-cryptography. The basic method to improve GOST performance is associated with careful substitution cycle (S1…S8) programming and, first of all, the number of reductions of such iterations (substitution boxes extension, registers exchange for bitwise rotation, key and substitution tables locations in RAM). Considering GOST as a lightweight-algorithm we obtain the best throughput/code size ratio (Fig. 3) compared with known implementations of other algorithms [1, 2]. The GOST efficiency on ARM-based architectures increases more due to the possibility of rotation (<<<11) and addition modulo 2 operations to combine in one instruction. The authors conclude that with similar performance (for AES-128), GOST implementation requires approximately 5 times less memory usage. In the identical key version AES-256 almost loses its advantage for maximum performance variant, outpacing GOST not more than 1.4 times (Tab. 2).

Keywords: cryptography algorithm GOST, lightweight-cryptography, ARM-core processors, encryption for embedded systems.

Mgr Yaroslav RESHETAR

W 2008 r. ukończył studia magisterskie w Narodowym Uniwersytecie "Politechnika Lwowska" na kierunku Obrona informacji o dostępie ograniczonym, oraz automatyzacja jej obróbki. Interesy naukowe: skuteczna sprzętowo-programowa realizacja algorytmów kryptograficyjnych, zapieniająca odporność do ataków przez uboczne kanały ucieczki informacji.

e-mail: reshetar_yv@lp.edu.uae

1. Wstęp

Systemy wbudowane znajdują coraz szersze zastosowanie w wielu dziedzinach, gdyż obecnie dąży się, aby takie urządzenia były inteligentne, mobilne, przydatne do pracy autonomicznej oraz wykonywały coraz bardziej złożone zadania. W systemach takich często jest wymagana ochrona danych. Dzięki wykorzystaniu algorytmów kryptograficznych deweloperzy mogą w znacznym stopniu podnieść poziom bezpieczeństwa. Dlatego istnieje zapotrzebowanie na kryptoalgorytmy, które można skutecznie zaimplementować w urządzeniach o dość ograniczonych parametrach (pamięć, wydajność jądra procesora, oszczędności energii).

Kierunek badań związany z algorytmami kryptograficznymi dla systemów o ograniczonych zasobach nosi nazwę kryptografii lekkiej (lightweight cryptography) [1]. Główny problem realizacji kryptografii lekkiej polega na tym, że w gotowym urządzeniu niezwykle trudno optymalizować poziom bezpieczeństwa, cenę oraz wydajność.

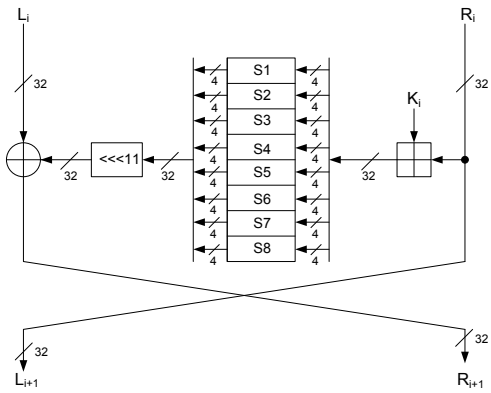
Ponadto, w ostatnich latach obserwuje się zainteresowanie przejściem od typowych 8/16-bitowych systemów wbudowanych do bardziej wydajnych platform 32-bitowych. Na rynku 32-bitowych jąder procesorowych dominuje architektura ARM, specjalnie opracowana dla wysokowydajnych procesorów RISC o niskim zapotrzebowaniu energii. Firma ARM (Advanced RISC Machine) oferuje szeroki wybór jąder procesorowych o wspólnej architekturze, które zostały objęte licencjami czołowych producentów mikrokontrolerów.

Najbardziej rozpowszechnionym kryptoalgorytmem symetrycznym stosowanym obecnie w systemach wbudowanych jest Advanced Encryption Standard (AES). Ten algorytm został specjalnie opracowany dla skutecznej implementacji programowej na procesorach zarówno ośmio-, jak i 32-bitowych. Inny algorytm kryptograficzny – GOST (*ros.* ***Go****sudarstwiennyj* ***St****andard* – *Standard Państwowy*), który pozostał standardem na terenach byłego ZSRR, jest również zorientowany na 32-bitowe platformy oraz posiada niewysoką złożoność, co jest ważne dla realizacji w systemach wbudowanych. Ze względu na to, że algorytm GOST nie znalazł jeszcze szerokiego zastosowania w systemach wbudowanych, warto przeprowadzić analizę porównawczą wyżej wymienionych algorytmów kryptograficznych zarówno w zagadnieniach 8/16-bitowej kryptografii lekkiej, jak i dla bardziej wydajnych, 32-bitowych systemów wbudowanych.

2. Charakterystyka właściwości strukturalnych algorytmu GOST

GOST należy do blokowych algorytmów kryptograficznych zbudowanych według schematu sieci Feistla. Rozmiar bloku wynosi 64 bity, a długość klucza - 256 bity. Podstawowymi operacjami wykonywanymi w ramach poszczególnych rund są: dodawanie modulo 2 i 2^{32}, zamiana nieliniowa $S_1...S_8$ i przesunięcie cykliczne (rys. 1). Takie operacje łatwo realizować w mikrokon-

trolerach ogólnego przeznaczenia dzięki wsparciu na poziomie instrukcji systemowych.



Rys. 1. Schemat strukturalny pojedynczej rundy algorytmu GOST
Fig. 1. GOST round function structure

Przed rozpoczęciem szyfrowania 64-bitowy blok tekstu jawnego jest zapisywany do dwóch 32-bitowych rejestrów $L_0 \parallel R_0$. Algorytm szyfrowania składa się z 32 rund. W *i*-tej rundzie zawartość prawego rejestru R_i jest dodawana modulo 2^{32} do 32-bitowego podklucza rundy K_i. Wynik dodawania zostaje przesunięty o 11 bitów w lewo w bloku podstawiania tabeli *S*.. W końcu wartość funkcji rundy $F(K_i,R_i)$ zostaje poddana operacji XOR z 32-bitową zawartością lewego rejestru L_i. Uzyskany wynik zapisuje się w prawym rejestrze R_{i+1}, a poprzednia zawartość prawego rejestru R_i zostaje bez zmian przepisana do lewego rejestru L_{i+1} jako wartość kolejnej rundy:

$$L_{i+1} = R_i; \quad R_{i+1} = L_i \oplus (S[(K_i + R_i) \bmod 2^{32}] <<< 11),$$

gdzie: $\oplus$ – oznacza operację XOR (dodawanie modulo 2); <<<11 – przesunięcie cykliczne w lewo o 11 bitów.

W końcowej 32 rundzie nie dokonuje się żadnej wymiany zawartości rejestrów, czyli

$$R_{32} = R_{31}; \quad L_{32} = L_{31} \oplus (S[(K_{31} + R_{31}) \bmod 2^{32}] <<< 11).$$

W algorytmie GOST nie wykorzystuje się procedury rozszerzenia klucza, jak w znanym algorytmie DES, co jest zaletą w kryptografii lekkiej. Klucz główny *K* o długości 256 bitów jest rozpatrywany jako konkatenacja ośmiu 32-bitowych podkluczy: $K=K_0 \parallel K_1 \parallel K_2 \parallel K_3 \parallel K_4 \parallel K_5 \parallel K_6 \parallel K_7$. W rundach o numerach $0 \le r \le 23$ podklucz rundy K_i określa się jako

$$K_i = K_{(r) \bmod 8} \text{ albo } K_i = (K_r)_{\bmod 8} \text{ albo (najlepiej) } K_i = K_r \bmod 8,$$

a dla końcowych ośmiu rund $24 \le r \le 31$ jako

$$K_i = K_{7-(r) \bmod 8} \text{ albo } K_i = K_{7-r} \bmod 8.$$

Podczas deszyfrowania jako dane wejściowe algorytmu są podawane 64-bitowe bloki szyfrogramu, a kolejność wykorzystania podkluczy rund jest odwrotna w stosunku do trybu szyfrowania.

Blok podstawiania składa się z ośmiu węzłów zamiany $S_1 ... S_8$, każdy z pamięcią 64 bitów. Na wejście bloku podstawiania jest podawany 32-bitowy wektor, który zostaje podzielony na osiem 4-bitowych wektorów. Każdy taki wektor jest argumentem nieliniowej funkcji *S*, której 4-bitowe wartości uzyskuje się na wyjściu odpowiednich węzłów. Węzeł zamiany jest tabelą z 16 elementami, po 4 bity każdy. W standardzie GOST nie zdefiniowano ściśle bloków podstawiania, ale istnieją dokumenty normatywne zatwierdzone przez odpowiednie jednostki państwowe, podające zawartość *S*-bloków zapewniających niezbędny poziom ochrony kryptograficznej.

3. GOST w aplikacjach kryptografii lekkiej

Najważniejszymi kryteriami oceny algorytmów ze względu na przydatność do wykorzystywania w kryptografii lekkiej są: wymagany rozmiar pamięci wewnętrznej (dla przechowywania kodu źródłowego algorytmu) oraz czas wykonania (jako miara wydajności) [1]. Oprócz tego przy badaniu algorytmów kryptografii lekkiej preferowana jest realizacja programowa, ponieważ w porównaniu z realizacją sprzętową jest ona tańsza, zdecydowanie elastyczniejsza oraz bardziej uniwersalna.

W tych dziedzinach, gdzie najważniejszymi wymaganiami są niska cena i mała energia zasilania, moc obliczeniowa jest zawarta przede wszystkim w małych, niedrogich procesorach, wśród których dominują mikrokontrolery 8-bitowe. Jako platformę dla realizacji algorytmu GOST wybrano mikrokontrolery o architekturze AVR. Są to tanie, wysokowydajne mikrokontrolery RISC o małej mocy zasilania, co umożliwiło ich szerokie wykorzystywanie w systemach wbudowanych. Ponadto, wykorzystywanie mikrokontrolerów AVR pozwala na porównanie parametrów realizacji GOST z już istniejącymi rozwiązaniami dla innych znanych kryptoalgorytmów [1, 2].

W kontekście kryptografii lekkiej ważną właściwością jądra AVR jest organizacja pamięci według architektury harwardzkiej z 8-bitową pamięcią danych typu SRAM (1-8 kB) oraz 16-bitową pamięcią programów typu Flash (8-128 kB). Plik rejestru zawiera 32 rejestry ogólnego przeznaczenia bezpośrednio podłączone do jednostki obliczeniowej. Większość z ponad 130 instrukcji mikrokontrolerów AVR jest wykonywana w czasie jednego taktu. Testy dla kryptoalgorytmów były przeprowadzone na modelu ATmega128.

Napisanie w języku asemblera kodu źródłowego kryptoalgorytmów, ich debugowanie, symulację oraz wyliczanie ilości taktów i rozmiaru kodu, wykonano w darmowym zintegrowanym środowisku deweloperskim AVR Studio 4.

Kryptografię lekką wykorzystuje się w systemach o różnym przeznaczeniu, gdzie głównymi wymaganiami w stosunku do algorytmu mogą być zarówno oszczędne wykorzystywanie pamięci (wariant SIZE), jak i maksymalna wydajność (wariant SPEED). Z tego powodu algorytm GOST został zrealizowany i zbadany dla obu tych przypadków.

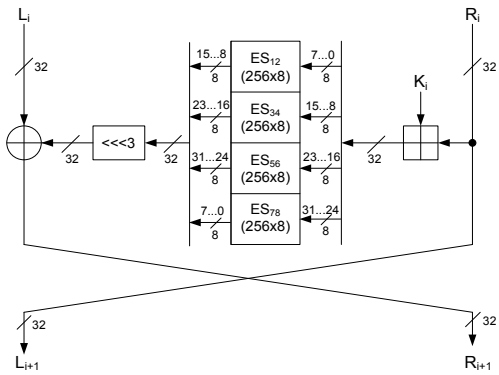
Wariant maksymalnej wydajności. Dla osiągnięcia maksymalnej wydajności realizacji programowej algorytmu należało zoptymalizować te fragmenty kodu, które są najczęściej wykonywane oraz najbardziej złożone w sensie obliczeniowym. W tym celu były wykorzystane najskuteczniejsze instrukcje i sposoby adresacji, które potrzebują minimalnej ilości taktów mikrokontrolera.

Żeby zapewnić maksymalną szybkość wykonania algorytmu GOST przeanalizowano funkcję rundy (rys. 1) uwzględniając właściwości architektury AVR. Ponieważ operacje dodawania modulo dwa są w istocie sprawy operacjami atomowymi (to znaczy są one realizowane za pomocą odpowiednich instrukcji procesora), wiec szybkość całego algorytmu będzie przede wszystkim zależała od skuteczności wykonania operacji podstawiania i przesunięcia cyklicznego. Dlatego główną możliwością zwiększenia szybkości algorytmu GOST jest gruntowne programowanie cyklu podstawiań i, przede wszystkim, zmniejszenie liczby iteracji tego cyklu.

Standard GOST przewiduje wykonanie ośmiu iteracji cyklu podstawiań, a w każdej dokonuje się jednej 4-bitowej zamiany. Dla przechowywania ośmiu tabel 4-bitowych podstawiań $S_1 ... S_8$ wystarczą 64 bajty. Aby zmniejszyć liczbę iteracji dla 8-bitowych mikrokontrolerów odczytujących z pamięci jeden bajt w ciągu jednego taktu, warto jednocześnie wykonać dwa 4-bitowe podstawienia kosztem zwiększenia rozmiaru tabeli podstawiań. W takim przypadku jedna rozszerzona tabela (*ES, Extended S-Box*) będzie zawierać 256 bajtów, a ogólny rozmiar czterech tabel 8-bitowych podstawiań ES_{12}, ES_{34}, ES_{56}, ES_{78} będzie wynosił 1 kB. Takie rozszerzone tablice są z góry zdefiniowane na podstawie długoterminowego klucza kryptograficznego i przechowywane w ROM

(pamięci flash programów), a ich rozmiar jest akceptowalny nawet dla mikrokontrolerów 8-bitowych.

Dodatkową możliwość zwiększenia wydajności daje optymalizacja wykonania przesunięcia cyklicznego. Operacja przesunięcia cyklicznego 32-bitowej wartości o 11 bitów w lewo może być traktowana jak kolejne przesunięcia odpowiednio o 8 bitów i 3 bity. Oczywiście, że przesunięcie cykliczne o 8 bitów jest równoważne wymianie zawartości między 8-bitowymi rejestrami i może być połączone z operacją podstawiania (rys. 2).



Rys. 2. Funkcja rundy z rozszerzonymi tabelami zamian oraz optymalizacją wykonania przesunięcia cyklicznego
Fig. 2. Round function with Extended S-boxes and rotation optimized

Pamięć flash programów jest wykorzystana do przechowywania kodu programu, tabeli zamian i tajnego klucza kryptograficznego. Dla mikrokontrolerów rodziny AVR liczba taktów potrzebnych do odczytania argumentu różni się w zależności od rodzaju pamięci (operacyjna czy stała), w której jest on przechowywany. Posiadając dostateczną objętość RAM można zwiększyć szybkość kosztem wcześniejszego załadowania do RAM klucza, a w niektórych przypadkach i tabeli podstawień.

Wariant minimalnego rozmiaru kodu. Dany wariant ma dwie odmiany względem wariantu maksymalnej wydajności: po pierwsze, nie są wykorzystane rozszerzone tabele zamian, po drugie nie dokonuje się wcześniejszego kopiowania niejawnego klucza i tabeli podstawień z pamięci flash do RAM.

Uzyskane wyniki dla obu wariantów przy częstotliwości zegara mikrokontrolera AVR 4 MHz, a także dane z literatury [1, 2] zostały przedstawione w tabeli 1.

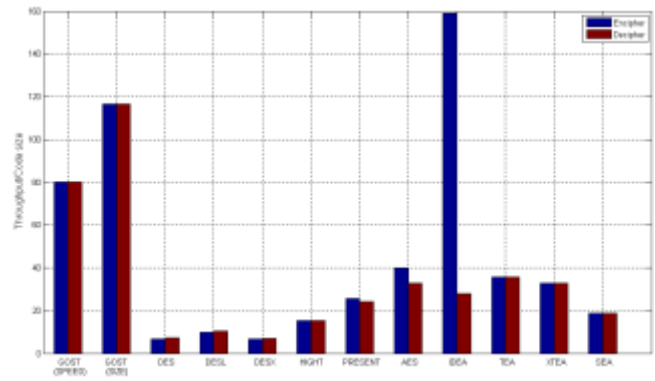
Tab. 1. Wyniki realizacji algorytmów kryptografii lekkiej dla architektury AVR
Tab. 1. Lightweight algorithm implementation results for AVR architecture

Algorytm	Szyfrowanie, takty/blok	Deszyfrowanie, takty/blok	Wydajność szyfrowania, kB/s	Wydajność deszyfrowania, kB/s	Rozmiar ROM, bajt
Szyfry blokowe zorientowane na realizację sprzętową					
DES [2]	8633	8154	29.6	31.4	4314
DESL [2]	8365	7885	30.6	32.5	3098
DESX [2]	8699	8220	29.4	31.1	4406
HIGHT [1]	2964	2964	86.4	86.4	5672
PRESENT [1]	10723	11239	23.9	22.8	936
Szyfry blokowe zorientowane na realizację programową					
AES [2]	3766	4558	136.0	112.3	3410
IDEA [1]	2700	15393	94.8	16.6	596
TEA [2]	6271	6299	40.8	40.6	1140
XTEA [2]	6718	6718	38.1	38.1	1160
SEA [2]	9654	9654	39.8	39.8	2132
Maksymalna wydajność					
GOST	2348	2348	109.0	109.0	1362
Minimalny rozmiar kodu					
GOST	4302	4304	59.5	59.5	510

Jak widać z tab. 1 realizacja algorytmu szyfrowania GOST w wariancie minimalnego rozmiaru kodu potrzebuje mniejszej objętości pamięci aniżeli inne wymienione algorytmy.

Wydajność jest to parametr pozwalający oszacować szybkość szyfrowania i deszyfrowania danych w bitach na sekundę i poprawnie przeprowadzić analizę porównawczą algorytmów. Dla konkretnych urządzeń szybkość działania zależy nie tylko od częstotliwości zegara mikrokontrolera, lecz również od długości bloku danych wejściowych oraz od liczby taktów, potrzebnych do zaszyfrowania czy odszyfrowania tego bloku. Jak wykazały badania, algorytm GOST w wariancie maksymalnej szybkości wykazuje stosunkowo wysoką wydajność, ustępując jedynie algorytmowi AES i wyprzedzając wszystkie pozostałe algorytmy.

Większość algorytmów kryptograficznych umożliwia zwiększanie szybkości kosztem wzrostu rozmiaru kodu przez realizację najbardziej złożonych operacji metodą tablicową, gdy wszystkie możliwe wyniki operacji są przechowywane w pamięci ROM w tak zwanych tabelach LUT (Look-Up Tables). Ze względu na to wprowadzono dodatkowy kompleksowy wskaźnik, wyliczany jako stosunek wydajności do rozmiaru kodu (rys. 3).



Rys. 3. Wskaźnik wydajność/rozmiar kodu realizacji algorytmów kryptografii lekkiej
Fig. 3. Throughput/code size ratio for lightweight algorithms implementations

Jak wynika z rys. 3, ze względu na kompleksowy wskaźnik wydajność/rozmiar kodu obie realizacje algorytmu GOST wyprzedzają wszystkie inne algorytmy, ustępując jedynie algorytmowi IDEA w trybie szyfrowania, ale zdecydowanie wyprzedzając go w trybie deszyfrowania.

4. GOST dla wysokowydajnych systemów wbudowanych

Równolegle z istnieniem klasy niedrogich systemów wbudowanych o ograniczonych zasobach, ciągłe wzrasta liczba systemów, w których krytycznym jest przetwarzanie informacji w czasie rzeczywistym. Do realizacji takich projektów, jak już wcześniej wspomniano, bardziej przydatną jest architektura ARM.

W celu przeprowadzenia testów porównawczych autorzy niniejszej pracy wybrali najbardziej rozpowszechnione jądra ARM w systemach wbudowanych – ARM7TDMI-S i ARM Cortex-M3. Obecnie na bazie tych urządzeń są produkowane setki rodzajów mikrokontrolerów dla najróżniejszych dziedzin i zastosowań. Podobnie jak w przypadku kryptografii lekkiej na platformie AVR, badania zostały przeprowadzone dla przypadku oszczędnego wykorzystywania pamięci (SIZE) i przypadku maksymalnej wydajności (SPEED). Dla każdego z wariantów rozpatrzone zostały realizacje w języku asemblera (ASM) i C.

W realizacjach funkcji szyfrowania/deszyfrowania GOST w asemblerze wykorzystano interfejs C, co daje możliwość wywoływania ich z programów w języku C/C++.

Wszystkie wyniki dla jądra ARM7TDMI uzyskano w trybie ARM, ponieważ tryb THUMB, jak wykazały badania, nie prowadzi do istotnej przewagi w rozmiarze kodu przy prawie dwukrotnym obniżeniu wydajności.

Opracowywanie oprogramowania w językach asemblera i C oraz oszacowanie liczby taktów i rozmiaru kodu przeprowadzono w środowisku IAR Embedded Workbench for ARM 5.30.

Wariant maksymalnej wydajności. Żeby zmniejszyć liczbę iteracji w przypadku mikrokontrolerów ARM, które przy jednym dostępie do pamięci odczytują co najmniej jeden bajt, celowym jest jednoczesne wykonanie dwu 4-bitowych podstawień kosztem zwiększonego rozmiaru tabeli zamian (patrz rozdz. 3).

Dla procesorów ARM istnieje możliwość wykonania operacji logicznej z jednoczesnym przesunięciem cyklicznym jednego z operandów w prawo. Dla tego procesora operacja cyklicznego przesunięcia argumentu o 11 bitów w lewo może być połączona z operacją dodawania modulo 2 wyniku i zawartości rejestru L_i, co nie wydłuża czasu wykonania komendy. W tym przypadku należy uwzględnić, że przesunięcie cykliczne 32-bitowej wartości o 11 bitów w lewo jest równoważne przesunięciu cyklicznemu w prawo o 21 bitów:

$$\text{EOR R2,R1,R3,ROR\#21;} \quad R2 = R1 \oplus (R3 >>> 21)$$

Struktura programu zawiera cztery iteracje po 8 rund. Aby zmniejszyć liczbę taktów potrzebnych dla 32 wywołań funkcji rundy, została ona włączona bezpośrednio w ciało programu (funkcja inline).

Dzięki dostępności w jądrze ARM dużej liczby rejestrów ogólnego przeznaczenia, wszystkie zmienne pośrednie algorytmu są lokalizowane w rejestrach, co powoduje obniżenie liczby odwołań do pamięci, a więc zwiększa szybkość. Dla jądra ARM Cortex-M3 zmienne pośriednie trzeba starać się rozmieścić w rejestrach R0-R7, ponieważ w tym przypadku są generowane instrukcje 16-bitowe, zamiast 32-bitowych.

Wyniki realizacji w językach asemblera i C dla wariantu maksymalnej szybkości zostały podane w tabeli 2.

Tab. 2. Wyniki realizacji algorytmów GOST i AES dla architektury ARM
Tab. 2. GOST and AES implementation results for ARM architecture

Platforma ARM	Szyfrowanie, takty/blok	Deszyfrowanie, takty/blok	Szyfrowanie, takty/bajt	Deszyfrowanie, takty/bajt	ROM, bajt
AES-128					
7TDMI (unrolling) [3]	1675	2074	105	130	Brak danych
7TDMI (on-the-fly) [3]	2074	2378	130	149	Brak danych
7TDMI [4]	639	638	40	40	5966
7TDMI [5]	1994	1994	125	125	7944
7TDMI [5]	5542	5542	346	346	2292
Cortex-M3 [6]	1329	1347	83	84	5272
AES-192					
7TDMI [4]	747	746	47	47	5966
AES-256					
7TDMI [4]	855	854	53	53	5966
GOST (Maksymalna wydajność)					
Cortex-M3 (ASM/C)	477/611	479/595	60/76	60/74	1810/2116
7TDMI-S (ASM/C)	608/702	610/678	76/88	76/85	1972/2152
GOST (Minimalny rozmiar kodu)					
Cortex-M3 (ASM/C)	1210/1269	1212/1265	151/159	152/158	434/474
7TDMI-S (ASM/C)	1566/1499	1568/1501	196/187	196/188	420/620

Wariant z minimalnym rozmiarem kodu. Dwie cechy odróżniają wariant z minimalnym rozmiarem kodu od wariantu maksymalnej szybkości. Po pierwsze, nie wykorzystuje się w nim rozszerzonych tabel zamian. Po drugie wykonuje się wywołanie funkcji rundy. Wyniki realizacji algorytmu GOST w językach asemblera i C dla wariantu minimalnego rozmiaru kodu, a także, dla porównania, dane z realizacji AES zostały zawarte w tabeli 2.

Analizując dane z tabeli 2 należy podkreślić, że jądro ARM Cortex-M3 zapewnia na ogół lepsze wyniki zarówno co do szybkości jak i rozmiaru kodu. Wydajność szyfrowania/deszyfrowania różni się nieznacznie. Wariant SIZE potrzebuje około 3,5-4,5 razy mniej pamięci w porównaniu do wariantu maksymalnej szybkości SPEED. Natomiast szybkość dla wariantu SPEED, przy wykorzystaniu języka asemblera jest 2,5 razy wyższa od szybkości realizacji SIZE, dla języka C – odpowiednio 2 razy.

Dla 60 MHz, typowej maksymalnej częstotliwości zegara mikrokontrolerów ARM7TDMI, można osiągnąć wydajność szyfrowania/deszyfrowania na poziomie 6 Mb/s. Mikrokontrolery z jądrem Cortex-M3 przy częstotliwości zegara 120 MHz zapewniają wydajność rzędu 16 Mb/s. Ze względu na to, że najszybszy interfejs standardowy USB, dostępny w wielu modelach mikrokontrolerów ARM, zapewnia szybkość transmisji danych do 12 Mbit/s (w trybie Full Speed) staje się oczywistym, że wydajność realizacji algorytmu GOST będzie ograniczona przede wszystkim nie przez wydajność procesora centralnego, a przez wydajność interfejsów wejścia-wyjścia.

5. Wnioski

Przedstawione w artykule wyniki badań potwierdzają perspektywiczność i celowość wykorzystania kryptoalgorytmu GOST w systemach wbudowanych. Dla aplikacji lekkich ograniczonych ze względu na zasoby (platforma AVR) kryptoalgorytm GOST zapewnia najwyższy stosunek wydajność/rozmiar kodu spośród wszystkich rozpatrzonych konkurencyjnych kryptoalgorytmów przy zachowaniu wysokiego poziomu bezpieczeństwa (długość klucza 256 bity).

Analiza porównawcza wskaźników dla typowych jąder ARM (tab. 2) daje podstawy do stwierdzenia, że dla obu wariantów realizacji (SPEED i SIZE), niezależnie od języka programowana oraz architektury procesora, algorytm GOST demonstruje przewagę w ekonomii zasobów pamięci przy porównywalnej wydajności. Minimalny rozmiar pamięci ROM niezbędny do realizacji algorytmu AES jest 5,4 razy większy, niż minimalny rozmiar ROM dla realizacji algorytmu GOST. Rezygnacja z użycia tablicy LUT przy realizacji algorytmu AES pozwala otrzymać rozmiar kodu prawie jednakowy z wariantem SPEED przy znacząco mniejszej wydajności (76 taktów/bajt dla algorytmu GOST, 346 taktów/bajt dla algorytmu AES-128). Najszybsza realizacja algorytmu AES-128 jest 1,5 razy szybsza od odpowiedniej realizacji GOST dla jądra ARM Cortex-M3 i, odpowiednio, 1,9 razy szybsza dla jądra ARM7TDMI-S. Przy jednakowych długościach klucza algorytm AES-256 praktycznie traci przewagę w szybkości, wyprzedzając realizację GOST odpowiednio 1,1 i 1,4 razy dla jąder ARM Cortex-M3 i ARM7TDMI-S.

6. Literatura

[1] Eisenbarth T. et al.: A survey of lightweight cryptography implementations. IEEE Design & Test of Computers, vol. 24, Nr 6, s. 522-533, 2007.

[2] Rinne S. et al.: Performance analysis of contemporary light-weight block ciphers on 8-bit microcontrollers. ECRYPT Workshop SPEED, s. 33-43, 2007.

[3] Bertoni G. et al.: Efficient software implementation of AES on 32-bit platforms. CHES 2003, vol. 2779, s. 159-171, Springer, 2003.

[4] Atasu K. et al.: Efficient AES implementations for ARM based platforms. In Proc. ACM SAC 2004, s. 841-845, 2004.

[5] Darnall M. et al.: AES software implementations on ARM7TDMI. In INDOCRYPT 2006, vol. 4329, s. 424-435, Springer, 2006.

[6] Using AES encryption and decryption with stellaris microcontrollers. Application Note, AN01251-03, Texas Instruments, 2010.

otrzymano / received: 11.08.2013
przyjęto do druku / accepted: 01.11.2013 *artykuł recenzowany / revised paper*