Dariusz BURAK
WEST POMERANIAN UNIVERSITY OF TECHNOLOGY
Zolnierska street 52, 71-210 Szczecin, Poland

Parallelization of the ARIA Encryption Standard

Ph.D. Dariusz BURAK

Received the PhD degree in Computer Science in 2007 from Szczecin University of Technology. He is an assistant professor of the Computer Science at the West Pomeranian University of Technology. His current research interests are focused on cryptography and software optimization.

e-mail: dburak@wi.zut.edu.pl

Abstract

In this paper there are presented the results of ARIA encryption standard parallelizing . The data dependence analysis of loops was applied in order to parallelize this algorithm. The OpenMP standard is chosen for presenting the algorithm parallelism. There is shown that the standard can be divided into parallelizable and unparallelizable parts. As a result of the study, it was stated that the most time-consuming loops of the algorithm are suitable for parallelization. The efficiency measurement for a parallel program is presented.

Keywords: ARIA encryption standard, parallelization, data dependency analysis, OpenMP.

Zrównoleglenie standardu szyfrowania ARIA

Streszczenie

W artykule zaprezentowano proces zrównoleglenia koreańskiego standardu szyfrowania ARIA. Przeprowadzono analizę zależności danych w pętlach programowych celem redukcji zależności danych blokujących możliwości zrównoleglenia algorytmu. Standard OpenMP w wersji 3.0 został wybrany celem prezentacji równoległości najbardziej czasochłonnych obliczeniowo pętli odpowiedzialnych za procesy szyfrowania oraz deszyfrowania danych w postaci bloków danych. Pokazano, że zrównoleglona wersja algorytmu składa się z części sekwencyjnej zawierającej instrukcje wejścia/wyjścia oraz równoległej, przy czym najbardziej czasochłonne pętle programowe zostały efektywnie zrównoleglone. Dołączono wyniki pomiarów przyspieszenia pracy zrównoleglonego standardu szyfrowania oraz procesów szyfrowania oraz deszyfrowania danych z wykorzystaniem dwóch, czterech, ośmiu, szesnastu oraz trzydziestu dwóch wątków oraz zastosowaniem ośmioprocesorowego serwera opartego na czterordzeniowych procesorach Quad Core Intel Xeon.

Słowa kluczowe: standard szyfrowania ARIA, zrównoleglenie, analiza zależności danych, OpenMP.

1. Introduction

In addition to security level, the cipher speed is the second most important feature of cryptographic algorithms. So even a little difference of speed may cause choice of the faster cipher if only the security level is the same. Therefore, it is important to enable the use of multiprocessors for encryption algorithms processing. The paper describes a software approach to the ARIA encryption algorithm parallelization. The major purpose of this paper is to present a parallelization process of the ARIA encryption algorithm along with the experimental results with shared memory multiprocessor. The paper is organized as follows. In Section 2, the ARIA encryption algorithm is briefly described. Section 3 contains a description of types of dependences in the loops of encryption algorithms. Section 4 discusses parallelization process of the ARIA encryption algorithm. Section 5 shows experimental results regarding the parallelized ARIA encryption algorithm. Conclusion remarks are given in Section 6.

2. ARIA encryption standard

ARIA is a block encryption algorithm developed by South Korean researchers in 2004 (version 1.0) [1] based on a substitution permutation network (SPN) that operates on 128-bit data blocks with a 128-, 192- or 256-bit key and with a variable number of rounds- 12, 14 or 16 (depending on the key size). ARIA was established as a South Korean standard block cipher algorithm in 2004 (KS X 1213:2004) [2] and was included in PKCS #11 in 2007 [3]. The ARIA algorithm can be divided into two parts: key scheduling and data randomizing part. The key scheduling consists of two phases: a non-linear expansion phase, in which key is expanded into four 128-bit words (W_i) and a linear key-schedule phase, in which the sub-keys are derived from the words (W_i). The data randomizing part takes a 128-bit input data and transforms it into a 128-bit ciphertext (in the case of encryption process) or a 128-bit plaintext (in the case of decryption process) using an involution diffusion layer (a 16 x 16 binary matrix), two kinds of substitution layer, where the state goes through 16 S-boxes and a round key addition, where the state is XORed with a 128-bit round key.

The encryption process with 128-bit data block (P) and the 192-bit key (K) is as follows:

```
P1 = FO (P, ek1);                 //round 1
P2 = FE (P1, ek2);                //round 2
P3 = FO (P2, ek3);                //round 3
P4 = FE (P3, ek4);                //round 4
P5 = FO (P4, ek5);                //round 5
P6 = FE (P5, ek6);                //round 6
P7 = FO (P6, ek7);                //round 7
P8 = FE (P7, ek8);                //round 8
P9 = FO (P8, ek9);                //round 9
P10 = FE (P9, ek10);              //round 10
P11 = FO (P10, ek11);             //round 11
P12 = FE (P11, ek12);             //round 12
P13 = FO (P12, ek13);             //round 13
C = SL2 (P13 ^ ek14)^ek15;        //round 14
```

where P- plaintext; P1÷P13- intermediate text; C- ciphertext; ek1÷ek15- encryption round keys (defined by K); FO- an odd round function; FE- an even round function; SL2- type 2 of substitution layer.

The decryption process of ARIA is the same as the encryption process except that encryption round keys are replaced by decryption round keys. For example, encryption round keys ek1÷ek15 of the14-round ARIA algorithm are replaced by decryption round keys dk1÷dk15, respectively **.**

To encrypt or decrypt more than one block, there are five official modes of the ARIA algorithm: ECB, CTR, CBC, CFB and OFB[4].

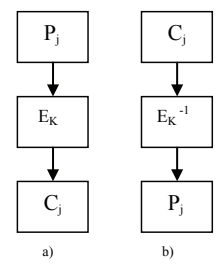

Fig. 1. The ECB mode of operation a) encryption process b) decryption process
Rys. 1. Schemat pracy trybu ECB a)szyfrowanie b) deszyfrowanie

The ECB mode (illustrated in Fig. 1) is defined as follows:
a) encryption process:

$$C_j = E_K(P_j) \text{ for } j = 1 \div n;$$

b) decryption process:

$$P_j = E_K^{-1}(P_j) \text{ for } j = 1 \div n;$$

where: P_j - data blocks, C_j - ciphertext blocks, E_K- encryption procedure; E_K^{-1}- decryption procedure.

The ARIA encryption algorithm is the South Korean encryption standard- alternative to the Advanced Encryption Standard (AES) [5] or Japanese encryption standard- Camellia [6].

3. Types of dependences

There are the following types of dependencies blocking parallelism in "for" loops of encryption algorithms [7, 8, 9]:

A Data Flow Dependence indicates a write-before-read ordering that must be satisfied for parallel computing. This dependence cannot be avoided and limits possible parallelism. The following loop yields such dependences:

```
for(i=0; i<n; i++) {
        a[i] = a[i - 1];
        }.
```

A Data anti-dependence indicates a read-before-write ordering that should not be violated when performing computations in parallel. There are techniques for eliminating such dependences. The loop below produces anti-dependences:

```
for(i=0; i<n; i++) {
        a[i] = a[i + 1];
        }.
```

An Output Dependence indicates a write-before-write ordering for parallel processing. There are techniques for eliminating such dependencies. The following loop yields output dependences:

```
for(i=0; i<n; i++) {
        a[0]=a[i];
        }.
```

In the case of a Control Flow Dependence value of variable (in S2) depends on the flow of control (in S1) like in the following example:

```
      for(int i=0; i<n; i++) {
S1:       if(x!=0)
S2:         y = 1.0/x;
      }.
```

All of the above loops cannot be executed in parallel in such a form, because results generated by the loops would be not the same as whose yielded with sequential loops. Thus, it is necessary to transform these loops so as to reduce such dependences.

4. Parallelization process

In order to present the parallelized source code of the ARIA encryption standard, the OpenMP API was applied.

The OpenMP Application Program Interface (API) supports multi-platform shared memory parallel programming in C/C++ and Fortran on all architectures including Unix and Windows NT platforms. OpenMP is a collection of compiler directives, library routines and environment variables that can be used to specify shared memory parallelism. OpenMP directives extend a sequential programming language with Single Program Multiple Data (SPMD) constructs, work-sharing constructs and synchronization constructs and enable to operate on private data [10, 11].

In order to parallelize the ARIA encryption standard in the ECB mode, there was used the sequential ARIA algorithm version 1.0 written in ANSI C [12]. This choice makes it possible to perform efficient parallelization in view of a high clarity code, enclosing most computations in iterative loops and a little number of the used functions. In order to enable enciphering and deciphering the whichever number of data blocks, there have been created two new functions, the ARIA_enc() for the encryption process and the ARIA_dec() for the decryption based on Crypt(), by analogy with similar functions included in the C source code of the cryptographic algorithms (DES- the des_enc(), the des_dec(), LOKI91- the loki_enc(), the loki_dec(), IDEA- the idea_enc(), the idea_dec(), etc.) presented in [13].

The main aim of the study is to enable enciphering and deciphering in parallel form messages that consist of many data blocks using the parallelized ARIA encryption algorithm working in the ECB mode of operation.

The process of the ARIA encryption standard parallelization can be divided into the following stages:

- finding the most time-consuming functions of the ARIA encryption standard;
- making preliminary transformations of the most time-consuming loops;
- data dependence analysis of the most time-consuming loops using Petit program [14];
- removal of data and control dependences (when it is possible);
- constructing parallel loops (in accordance with the OpenMP standard).

There have been carried out experiments with the sequential ARIA encryption standard that encrypts and then decrypts 3 megabytes plaintext in order to find the most time-consuming functions including no I/O functions. It has been discovered that such functions are included in the ARIA_enc() and in the ARIA_dec(), thus their parallelization is critical for reducing the total time of the algorithm execution.. Taking into account the strong similarity of both loops there is shown only ARIA_enc() function:

```
typedef unsigned char Byte;
typedef unsigned int  Word;

void ARIA_enc( const Byte *i, int Nr, const Byte *rk, Byte *o,
int blocks) {

int ii;
Word t0, t1, t2, t3;

for (ii=0; ii<blocks; ii++) {
  WordLoad(WO(i+16*ii,0), t0); WordLoad(WO(i+16*ii,1), t1);
  WordLoad(WO(i+16*ii,2), t2); WordLoad(WO(i+16*ii,3), t3);

  if (Nr > 12) {KXL FO KXL FE}
  if (Nr > 14) {KXL FO KXL FE}
  KXL FO KXL FE KXL FO KXL FE KXL FO KXL FE
  KXL FO KXL FE KXL FO KXL FE KXL FO KXL

#ifdef LITTLE_ENDIAN
  o[ 0+16*ii] = (Byte)(X1[BRF(t0,24)]   ) ^ rk[ 3];
  o[ 1+16*ii] = (Byte)(X2[BRF(t0,16)]>>8) ^ rk[ 2];
  o[ 2+16*ii] = (Byte)(S1[BRF(t0, 8)]   ) ^ rk[ 1];
  o[ 3+16*ii] = (Byte)(S2[BRF(t0, 0)]   ) ^ rk[ 0];
  o[ 4+16*ii] = (Byte)(X1[BRF(t1,24)]   ) ^ rk[ 7];
  o[ 5+16*ii] = (Byte)(X2[BRF(t1,16)]>>8) ^ rk[ 6];
  o[ 6+16*ii] = (Byte)(S1[BRF(t1, 8)]   ) ^ rk[ 5];
  o[ 7+16*ii] = (Byte)(S2[BRF(t1, 0)]   ) ^ rk[ 4];
  o[ 8+16*ii] = (Byte)(X1[BRF(t2,24)]   ) ^ rk[11];
  o[ 9+16*ii] = (Byte)(X2[BRF(t2,16)]>>8) ^ rk[10];
  o[10+16*ii] = (Byte)(S1[BRF(t2, 8)]   ) ^ rk[ 9];
```

```
  o[11+16*ii] = (Byte)(S2[BRF(t2, 0)]   ) ^ rk[ 8];
  o[12+16*ii] = (Byte)(X1[BRF(t3,24)]   ) ^ rk[15];
  o[13+16*ii] = (Byte)(X2[BRF(t3,16)]>>8) ^ rk[14];
  o[14+16*ii] = (Byte)(S1[BRF(t3, 8)]   ) ^ rk[13];
  o[15+16*ii] = (Byte)(S2[BRF(t3, 0)]   ) ^ rk[12];
#else
  o[ 0+16*ii] = (Byte)(X1[BRF(t0,24)]   );
  o[ 1+16*ii] = (Byte)(X2[BRF(t0,16)]>>8);
  o[ 2+16*ii] = (Byte)(S1[BRF(t0, 8)]   );
  o[ 3+16*ii] = (Byte)(S2[BRF(t0, 0)]   );
  o[ 4+16*ii] = (Byte)(X1[BRF(t1,24)]   );
  o[ 5+16*ii] = (Byte)(X2[BRF(t1,16)]>>8);
  o[ 6+16*ii] = (Byte)(S1[BRF(t1, 8)]   );
  o[ 7+16*ii] = (Byte)(S2[BRF(t1, 0)]   );
  o[ 8+16*ii] = (Byte)(X1[BRF(t2,24)]   );
  o[ 9+16*ii] = (Byte)(X2[BRF(t2,16)]>>8);
  o[10+16*ii] = (Byte)(S1[BRF(t2, 8)]   );
  o[11+16*ii] = (Byte)(S2[BRF(t2, 0)]   );
  o[12+16*ii] = (Byte)(X1[BRF(t3,24)]   );
  o[13+16*ii] = (Byte)(X2[BRF(t3,16)]>>8);
  o[14+16*ii] = (Byte)(S1[BRF(t3, 8)]   );
  o[15+16*ii] = (Byte)(S2[BRF(t3, 0)]   );
  WO(o+16*ii,0)^=WO(rk,0); WO(o+16*ii,1)^=WO(rk,1);
  WO(o+16*ii,2)^=WO(rk,2); WO(o+16*ii,3)^=WO(rk,3);
#endif
        }
}.
```

The parallelization process is presented only for the loops included in the function ARIA_enc() (however, this analysis is valid also for the ARIA_dec() function).

To reduce data dependences existing in the source code, we have to make the privatization of the indexing variable ii and variables t0, t1, t2 and t3.

The source code of the loop is suitable to apply the following OpenMP API constructs [10], [11]:

- parallel region construct ("parallel" directive);
- work-sharing construct ("for" directive)- all the iterations of the associated loop can be executed in parallel in this case.

Thus, the ARIA_enc() function with the parallelized most time-consuming loop has the following form (in accordance with the OpenMP API):

```
typedef unsigned char Byte;
typedef unsigned int  Word;

void ARIA_enc( const Byte *i, int Nr, const Byte *rk, Byte *o,
int blocks) {

int ii;
Word t0, t1, t2, t3;

#pragma omp parallel private(ii,to,t1,t2,t3)
#pragma omp for
for (ii=0; ii<blocks; ii++) {
        …//body of the loop
        }
}.
```

The most outer loop included in the paralllelized ARIA_enc() function (indexed by ii variable (and responsible for encryption of 192-bit data blocks)) can be executed in parallel by multiple processors. So the loop-level parallelism is exploited by multiprocessors in this case.

5. Experimental results

In order to study the efficiency of the parallel code of the ARIA encryption standard, there has been used a computer with eight Quad-Core Intel® Xeon processors E7310 - 1,60 GHz and the Intel® C++ Compiler ver. 11.0 (that supports the OpenMP 3.0). The results received for a 5 megabytes input file, 128-bit data blocks, 192-bit key (14 rounds) using two, four, eight, sixteen and thirty two processors versus the only one are given in Table 1.

The total running time of the ARIA encryption standard consists of the following time-consuming operations:

- data receiving from an input file,
- data encryption,
- data decryption,
- data writing to an output file (both encrypted and decrypted text).

Tab. 1. Speed-up measurements of the ARIA encryption standard
Tab. 1. Pomiary przyspieszenia pracy standardu szyfrowania ARIA

Number of Threads	Speed-up Encryption	Speed-up Decryption	Speed-up of Whole Algorithm
1	1	1	1
2	1,8	2	1,5
4	3,6	3,9	2,1
8	3,9	4,5	2,4
16	4,2	4,9	2,7
32	4,1	4,3	2,3

The total speed-up of the parallelized ARIA encryption standard depends considerably on two major factors: whether the most time-consuming loops are parallelizable and the method of data reading and data writing.

The results confirm that the paralleled codes of the most time-consuming loops (placed in the ARIA_enc() and the ARIA_dec() functions) have sufficient efficiencies.

The block method of reading data from an input file and writing data to an output file was used. The following C language functions and block sizes were applied: the fread() function and the 512-bytes block for data reading and the fwrite() function and the 256-bytes block for data writing. The optimal sizes of the blocks were chosen via the appropriate number of tests with various block sizes.

In accordance with Amdahl's Law [15] the maximum speed-up of the whole ARIA encryption standard is limited to 5.30, because the fraction of the code that cannot be parallelized is 0.187. This fraction is calculated as the quotient of a sum of the execution time of all unparallelizable operations divided by the execution time of the whole algorithm.

The difference between the speed-ups obtained for the encryption and decryption processes and for eight, sixteen and thirty two processors is due to the fact that during the decryption process (which is executed after the encryption process) data is stored in the local memory.

6. Conclusions

In this paper there is described the parallelization of the ARIA encryption standard. The ARIA encryption standard was divided into parallelizable and unparallelizable parts. There was shown that the iterative loops included in the most time-consuming functions (responsible for the data blocks encryption and decryption) were fully parallelizable. In order to parallelize these loops, it is necessary to make appropriate transformations of the body loops (described in the Section 4) and use the variable privatization technique.

The experiments carried out on the shared memory multiprocessor with one, two, four, eight, sixteen and thirty two threads show that the application of the parallelized ARIA

encryption standard considerably boost the time of the data encryption and decryption. It seems that the speed-ups received for the most time-consuming loops are satisfactory. The rest of the time-consuming parts of the code, contains I/O functions that are unparallelizable because the access to memory is, by its nature, sequential. Hence, the total speed-up is less than that for the parallelizable part.

The parallelized ARIA encryption standard presented in this paper can be also helpful for hardware implementations of this algorithm or NVIDIA CUDA based implementations. The hardware synthesis of the ARIA encryption standard will depend on the appropriate adjustment of the data transmission capacity and the computational power of hardware.

7. References

[1] Lee J., Lee J., Kim J., Kwon D. and C. Kim: A Description of the ARIA Encryption Algorithm, RFC 5794, March 2010.
[2] Kwon D., et al.: New Block Cipher: ARIA, ICISC 2003.
[3] Biryukov A., et al.: Security and Performance Analysis of ARIA, K.U. Leuven, 2003.
[4] Dworkin M.: Recommendation for Block Cipher Modes of Operation: Methods and Techniques, NIST Special Publication 800-38A, December 2001.
[5] Advanced Encryption Standard, (AES) FIPS PUB197, November 2001.
[6] Aoki K., Ichikawa T., Kanda M., Matsui M., Moriai S., Nakajima J. and Tokita T.: Camellia: A 128-Bit Block Cipher Suitable for Multiple Platforms - Design and Analysis -, In Selected Areas in Cryptography, 7th Annual International Workshop, SAC 2000, August 2000, Proceedings, Lecture Notes in Computer Science 2012, pp.39-56, Springer-Verlag, 2001.
[7] Moldovan D.I.: Parallel Processing. From Applications to Systems, Morgan Kaufmann Publishers, Inc., 1993.
[8] Allen R., Kennedy K.: Optimizing compilers for modern architectures: A Dependence-based Approach, Morgan Kaufmann Publishers, Inc., 2001.
[9] Bielecki W.: Essentials of parallel and distributed computing, Informa, 2002.
[10] OpenMP C and C++ Application Program Interface. Version 3.0, 2008.
[11] Chandra R., Dagum L., Kohr D., Maydan D., McDonald J., Menon R.: Parallel Programming in OpenMP, Morgan Kaufmann Publishers, Inc., 2001.
[12] http://code.google.com/p/openariasrtp/source/browse/trunk/srtp/crypto/cipher/aria050117.c
[13] Schneier B.: Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition, John Wiley & Sons, 1995.
[14] Kelly W., Maslov V., Pugh W., Rosser E., Shpeisman T., Wonnacott D.: New User Interface for Petit and Other Extensions. User Guide, 1996.
[15] Amdahl G.M.: Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities, In AFIPS Conference Proceedings, pages 483—485, 1967.

otrzymano / received: 06.12.2011
przyjęto do druku / accepted: 03.01.2012 *artykuł recenzowany / revised paper*