Parallel Model for Rabbit Stream Cipher over Multi-core Processors

KHALED SUWAIS
ITC Department
Arab Open University (AOU)
Riyadh 11681
SAUDI ARABIA
khaled.suwais@arabou.edu.sa

Abstract: - This paper presents a new parallel model for Rabbit stream cipher. The goal of this model is to enhance the performance of Rabbit cipher by accelerating its keystream generation and encryption processes. The underlying concept of the new model was built based on utilizing multi-core processors to generate multiple keystreams simultaneously. The results showed that the new parallel model could enhance the encryption speed of Rabbit of about 1.4, 1.6 and 2.3 times on single, dual and quad core processors.

Key-Words: - Parallel processing, Multi-core processors, Rabbit cipher, Stream cipher, Multithreading.

1 Introduction

Encryption algorithms are concerned of transforming readable texts (plaintext) to unreadable and uncomprehending text (ciphertext). In stream ciphers, the encryption algorithm generates a stream of bits that are exclusively-ORed (XOR'ed) with a stream of plaintext bits to generate a stream of ciphertext bits. Traditionally, stream ciphers use textual secret key to initiate the key generation process. The textual secret key is used as Initial Vector (IV) in all stream ciphers. For security purposes, these keys should be long enough (128 bit as minimum) to satisfy the minimum security requirements.

Rabbit stream cipher is one of the secure ciphers [1] [2] [3] which is based on iterating a set of coupled non-linear functions (discretized chaotic maps) [4]. It uses a 128-bit Secret Key (SK) and 64-bit Initial Vector (IV) as input parameters to generate a stream of 128-bit blocks.

In this research we present a new parallel design for Rabbit stream cipher which also replaces the textual keys by images. The proposed model aims to utilize multi-core processors using multithreading techniques. On the other hand, the new model replaces the textual IVs by images, where the input image is treated as general key for extracting both of SK and IV of Rabbit stream cipher. The reason behind replacing the textual keys by images is to extend the security and practicality levels of Rabbit cipher, where we can generate secure keys easily (using one single key instead of two separate IV and SK). The new design ensure satisfying both the security and performance requirements. The security level is ensured as the keystream generator depends mainly on the secure Rabbit cipher, while the performance is ensured through parallelism over multi-core processors.

The rest of the paper is organized as follows: Section 2 introduces related concepts on stream ciphers, Rabbit cipher and parallel computing over multi-core processors. The related works is discussed in Section 3. Section 4 describes the structure of the proposed parallel model. The security analysis is discussed in Section 5. The implementation and performance evaluation of the proposed algorithm is presented in Section 6. Finally, concluding remarks are presented at the end of the paper.

2 Definitions and Preliminaries

2.1 Stream Ciphers

2.1.1 Overview

The idea of stream ciphers was inspired from the famous cipher called the One-time Pad [5] [6]. This cipher is based on XOR'ing ($\oplus$) the message bits and the key bits. The One-time pad is defined as in Equation 1:

$$\boldsymbol{E} : \{0,1\} \times \{0,1\} \rightarrow \{0,1\},\ (\boldsymbol{m}, \boldsymbol{k}) \rightarrow \boldsymbol{m} \oplus \boldsymbol{k} \qquad (1)$$

where plaintext, keystream and ciphertext bits are in the space {0, 1}. The encryption transformation is given by:

$$E_{k_i}(m_i) = m_i \oplus k_i = c_i \in C \quad (2)$$

and the decryption transformation is given by:

$$D_{k_i}(c_i) = c_i \oplus k_i = m_i \in M \quad (3)$$

Definition 1: Let k_1, k_2, …, k_n be a set of keystream in the key space K, m_1, m_2, …, m_n be a set of plaintext in the plaintext space M, and $c_1, c_2, ..., c_n \in C$ be a set of ciphertext in the ciphertext space C. The encrypted ciphertext is generated by:

$$E_{k_i}(m_i) = c_1, c_2, ..., c_n \in C \qquad \forall i : 1 \le i \le n \quad (4)$$

From the above definition, the encryption process of a stream cipher E_k is bijective for every k_i. The plaintext space and key space are typically represented in bit or byte representations. Most importantly, keeping the key k is essential for the security of stream ciphers. Fig. 1 illustrate the general structure of stream ciphers.

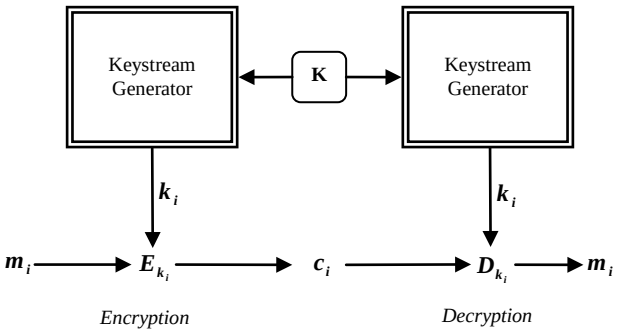

Fig. 1. Structure of Stream Cipher

The encryption process of a synchronous stream cipher is best described by the following equations:

$$\partial_{i+1} = NS(\partial_i, K), \quad (5)$$

$$k_{s_i} = KG(\partial_i, K), \quad (6)$$

$$c_i = EN(k_{s_i}, m_i) \quad (7)$$

where ∂_0 is the initial state determined by the input key K, NS is the next-state function, KG is the keystream generation function which generates k_s, and EN is the output function which generates the ciphered text c_i by combining both the plaintext m_i and the keystream k_{si}.

2.1.2 Rabbit Stream Cipher

The Rabbit cipher is a stream cipher that utilize a 128-bit secret key with a 64-bit Initialization Vector (IV). The Rabbit encrypts 128-bits in each iteration synchronously to provide an effective ciphered bit stream. The internal structure of Rabbit is divided into four stages: Key/IV insertion, Key setup, IV setup and Encryption. Note that the authors of Rabbit have not specified the insertion of Key/IV. Fig. 2 illustrates the main stages of Rabbit stream cipher.

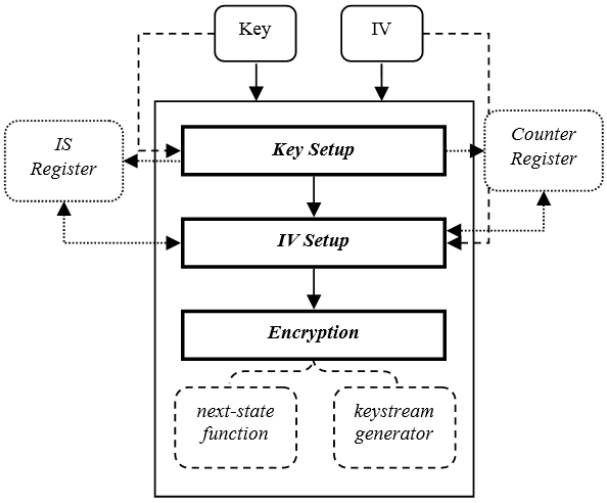

Fig. 2. Main stages of Rabbit cipher

In the second stage of Rabbit cipher, the key is initialized using internal state (IS) registers (denoted by X) and counter registers (denoted by C). In this stage, the key bits are assigned to all X's ($x_{j,0}$) and C's values ($c_{j,0}$). After that, the next-state function is called to scramble values of X and increment the counters C. At the end of this stage, the counters are re-initialized by XOR'ing the state registers with the previously-initialized values of counters.

In the third stage, the 64-bit IV is utilized such that the eight counter registers are modified by XOR'ing every bit of the IV with every bit in the counters. This technique is used to ensure that we have 2^{64} possible unique keystreams for any given secret key. After combining the IV with the counters, the IV setup call the next-state function to ensure the right mixture of bits of IV. Calling the next-state function will mix the IV bits using the values of the counters and the state registers.

Finally, the encryption stage is carried out by initiating two internal functions: the next-state function and the keystream generator. Here, the counter registers are updated by combining the current state of all counter registers with a constant A_j and the carry bit value. Updating the counter

states is accomplished sequentially since the value of counter C_{i+1} depends on the state of C_i.

After updating the counter registers, the state registers $X_{j,i+1}$ are calculated accordingly. Each state register and its associated counter register are used to generate a G value as a function $g_j(x_{j,i},c_{j,i+1})$. Upon calculating all g_j values, each state register is updated as a function of G values.

Once the counter and state registers are iterated, the keystream generator is initiated. In this generator, the XOR operation is applied on different state registers to create eight 16-bit keystream registers. The resulted random keystream bits are then used to XOR the bits of the plaintext stream.

2.2 Parallel Computing and Multi-core Processors

The need for secure and high performance communication has become an important ingredient in our daily life. Achieving security and high performance is possible by utilizing more computer resources, where several jobs are accomplished simultaneously and therefore completed in shorter time. Parallel processing is described as the concurrent use of multiple processing resources (e.g. multiple CPUs) to solve a computational problem, where a given problem is broken into smaller segments and solved concurrently using multiple processors.

Multithreading technique was introduced in 1960s [7] to enhance the performance of applications by running them in parallel over the available resources. This technique aims to create a virtual multiprocessors environment to run multiple tasks on single processor. The recent hardware revolution has played a significant role in improving systems performance through the multi-core technology. Multi-core is a technology where a single physical processor contains the logical core of more than one processor.

Multithreading technique maps independent tasks to threads to give the operating system greater flexibility in process scheduling, which in turn hides program latency [8] as shown in Fig. 3. The Lightweight Processes form the underlying threads of control, which are supported by the kernel. Each of the processes can be executed by multiple threads, resulting in faster execution in a shorter period of time. Examples of multithreaded libraries found in the literature include: *libthread* [9], *Posix* [10], *Pth* [11] and *bb_threads* [12].

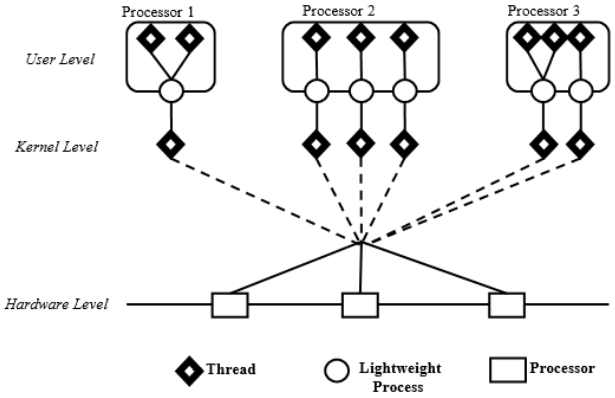

Fig.3 Multithreading Architecture

3 Related Works

Parallel cryptographic primitives have been studied from two perspectives: the design of new cryptographic hardware (*hardware-based parallelism*), and the parallelization of the mathematical and logical construct of the internal design (*software-based parallelism*). Several researches on hardware-based parallelism have been conducted. Examples of hardware-based parallelism includes: the implementation of Rijndael algorithm on VHDL (VHSIC Hardware Description Language) [13], the Field-Programmable Gate Array (FPGA) implementation of RSA [14], and the FPGA-based symmetric encryption algorithms [15].

Software parallelism is implemented hierarchically in two levels: job-level (*highest level*) and instruction-level (*lowest levels*). One of the software parallelism-based researches and implementations is interested in applying different techniques of parallelism including per-connection, per-packet and intra-packet parallelism to increase the throughput of different protocols stack [16]. Another research in the same category parallelized the three-layer model for elliptic curve scalar multiplication, which can be applied on different cryptographic applications due to its higher performance compared to the conventional implementation of elliptic curve cryptography [17].

The existence of parallel stream cipher designs is limited in the literature. One design was presented in [18] where a parallel-structured PS-LFSR (Linear Feedback Shift Register) is used to perform *m*-bit shifting/outputting for one clock and parallelizing many similar keystream generators for faster processing. Another parallelism technique applied in the parallel stream cipher proposed in [19] relies on parallelizing four types of nonlinear combiners: *m*-parallel nonlinear combiner without memory, *m*-parallel nonlinear combiner memories, *m*-parallel

nonlinear filter function and *m*-parallel clock-controlled function using PS-LFSR.

4 Parallel Model for Rabbit Cipher

In order to facilitate parallelism, the proposed stream cipher is composed of three components: The Secret Key Initial Vector Initializer (**SK/IVI**), the Keystream Generator (**KsG**) and the Plaintext Encoder (**PtE**). The initial vector initializer is responsible for extracting the general secret key from the input image key. The extracted key will be used as input to the keystream generator that is responsible for generating pseudorandom keystreams. These keystreams should be secure enough to be used for encrypting a stream of plaintext bit in the plaintext encoder.

4.1 Key/IV Initializer

In this component, the secret key (SK) and the initial vector (IV) of Rabbit cipher are computed based on the input image. The image is divided into four quarters (as illustrated by Fig. 4). Each quarter is used to extract a portion of the SK/IV. The complete SK/IV is formed by combining the extracted portions from the four quarters.

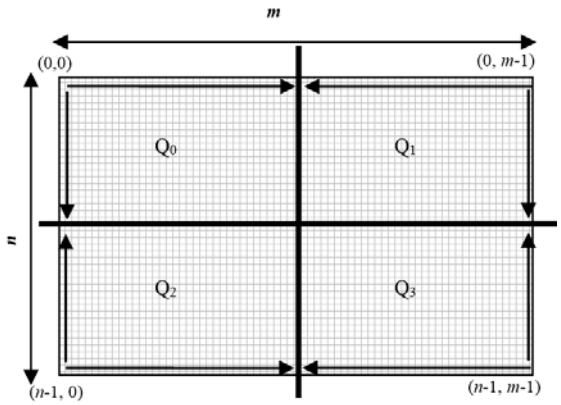

Fig. 4. Dividing input image for initializing the secret key

Obviously, there will be four subsystems to handle the computation of SK/IV. These subsystems are described in Equations 8-11:

$$Q_0 = [P_{(0,0)} \dots P_{(0,m/2-1)}] \lozenge [P_{(0,0)} \dots P_{(n/2-1,0)}] \quad (8)$$
$$Q_1 = [P_{(0,m-1)} \dots P_{(0,m/2)}] \lozenge [P_{(0,m-1)} \dots P_{(n/2-1,m-1)}] \quad (9)$$
$$Q_2 = [P_{(n-1,0)} \dots P_{(n/2,0)}] \lozenge [P_{(n-1,0)} \dots P_{(n-1,m/2-1)}] \quad (10)$$
$$Q_3 = [P_{(n-1,m-1)} \dots P_{(n/2,m-1)}] \lozenge [P_{(n-1,m-1)} \dots P_{(n-1,m/2)}] \quad (11)$$

where $\lozenge$ denotes the concatenation. Consequently the final output (τ) is calculated using the SHA-256 (denoted by SHA) algorithm as follows:

$$\tau = \mathrm{SHA}(Q_0 \lozenge Q_1 \lozenge Q_2 \lozenge Q_3) \quad (12)$$

Once the final output is obtained, the secret key and initial vectors are generated such that:

$$SK = \tau\,[b_0, \dots, b_{127}] \quad (13)$$
$$IV = \tau\,[b_{128}, \dots, b_{191}] \quad (14)$$

where b_i denotes the bit number of τ that is considered in computing both of SK and IV.

Computing the SK/IV of Rabbit cipher is carried out in parallel, where the values of Q_i (for all i= 0,…,3) is executed simultaneously. Note that the number of the generated threads in this stage is four. For instance, if the model is running over dual-core processors, each two threads will be associated with one core. On the other hand, on quad-core processors, the model will associate each thread with one single core ($\hat{C}$) as shown in Fig. 5.

The generated SK/IV from this component is the input to the second component (**KsG**), where stream of keys is generated to encrypt stream of plaintext securely and efficiently.

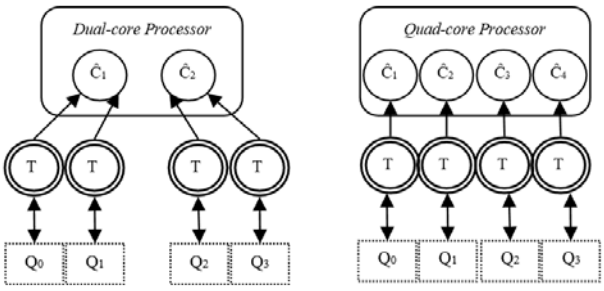

Fig. 5. Multithreading scheme for computing SK/IV over multi-core processors

4.2 Keystream Generator (KsG)

The Keystream Generator (**KsG**) is mainly responsible for performing the next-state function of Rabbit. Therefore, the main purpose of **KsG** is to parallelize the keystream generation function of Rabbit cipher such that Rabbit can generate multiple keystreams simultaneously. Generating multiple keystreams requires creating and controlling multiple threads. The number of created threads, in this particular component, is mainly based on the number of available cores in the running processor.

Each processor is responsible for running a set of counter values which are incremented sequentially in each iteration of the next-state function. Generally, For a given processor of *n* cores, the created threads are controlled as stated in Equation 15:

$\hat{C}_i$ **§** T_i **§** R_i *for all* $0<i<\rho$ (15)

where **§** denotes the association between processors cores ($\hat{C}_i$), threads (T_i) and counters (R_i), while **ρ** denotes the total number of generated threads. The counter set of R_i includes the following sub-systems*: Carry-bit Resolution* (**CbR**), *Counter Iteration* (**CI**) and *State Iteration* (**SI**). In **CbR**, the carry-bit Φ is modified for each counter register update as shown in Equations 16 and 17:

$\Phi_{j,i+1} = 1$ if $[(((c_{j,i}+\alpha_j+\Phi_{j,i+1}) \ mod \ 32) \geq 2^{32})$ *and* $(j=0)]$
or
$[(((c_{j,i}+\alpha_j+\Phi_{(j-1),(i+1)}) \ mod \ 32) \geq 2^{32})$ & $(j\geq 1)]$ (16)
$\Phi_{j,i+1} = 0$ *for all other cases* (17)

where $c_{j,i}$ refers to the register index j in iteration number i, and α_j refers to a hard-coded constant value.

In the second sub-system, the counter iteration is carried out. In **CI**, the counter registers are updated sequentially. Each counter is updated using its current value accompanied by hard-coded register value, and the previous value of Φ. Equations 18-25 describes the updating process of the counter registers:

$C_{0,i+1} = C_{0,i} + \alpha_0 + \Phi_{7,i} \ mod \ 2^{32}$ (18)
$C_{1,i+1} = C_{1,i} + \alpha_1 + \Phi_{0,i+1} \ mod \ 2^{32}$ (19)
$C_{2,i+1} = C_{2,i} + \alpha_2 + \Phi_{1,i+1} \ mod \ 2^{32}$ (20)
$C_{3,i+1} = C_{3,i} + \alpha_3 + \Phi_{2,i+1} \ mod \ 2^{32}$ (21)
$C_{4,i+1} = C_{4,i} + \alpha_4 + \Phi_{3,i+1} \ mod \ 2^{32}$ (22)
$C_{5,i+1} = C_{5,i} + \alpha_5 + \Phi_{4,i+1} \ mod \ 2^{32}$ (23)
$C_{6,i+1} = C_{6,i} + \alpha_6 + \Phi_{5,i+1} \ mod \ 2^{32}$ (24)
$C_{7,i+1} = C_{7,i} + \alpha_7 + \Phi_{6,i+1} \ mod \ 2^{32}$ (25)

In the last sub-system (**SI**), the 32-bit X internal-states values are calculated by applying some rotation and addition operations over the previously calculated G values as follows:

$g_{j,i} = ((x_{j,i} + c_{j,i+1})^2 \oplus (x_{j,i} + c_{j,i+1})^2 \ \gg 32)) mod 2^{32}$ (26)

Once we obtain the G values, the internal states are updated as shown in Equations 27 and 28, where each equation is executed based on the value of *j*:

$x_{j,i+1} = g_{j,i}+(g_{7,i} <<<16)+(g_{6,i} <<<16)$ *for even j* (27)
$x_{j,i+1} = g_{j,i} + (g_{0,i} \ <<< 8) + g_{7,i}$ *for odd j* (28)

In **KsG**, we treat the above discussed sub-systems as one single unit. In this component, parallelism is carried out by creating multiple copies of the keystream generation of Rabbit. Each copy of the keystream generator of Rabbit is associated with a specific thread, which runs over a specific processor core. Practically, each copy of Rabbit keystream generator is associated with its own set of IV/SK generated from the IV/SK Initializer as illustrated by Fig. 6.

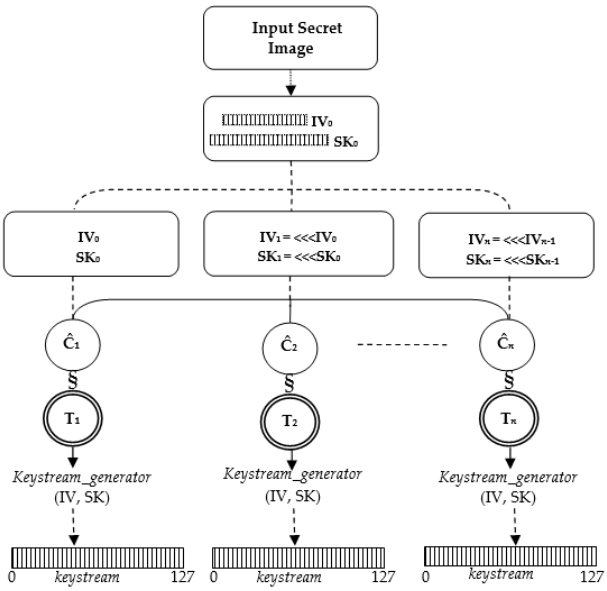

Fig. 6. Parallel scheme for generating multiple keystreams simultaneously

To parallelize the Rabbit keystream generator efficiently and consistently, each thread should be associated with its own set of IV/SK. Unlike the original implementation of Rabbit, our implementation associate each thread with its own set of IV/SK values, such that the values of IV/SK used by T_i differ from the IV/SK set used by the rest T_{n-1} threads. In order to achieve such free-dependency (as crucial factor in parallelism), the extracted IV/SK from the input secret image is shifted sequentially, such that the first thread (T_1) uses the originally extracted IV/SK (denoted by IV_0, SK_0), the second thread (T_2) perform 2-bit left rotation over IV_0, SK_0 to generate IV_1, SK_1. Sequentially, thread T_n performs 2-bit left rotation over the previously calculated IV_{n-1}, SK_{n-1} to generate IV_n, SK_n. Note that with 2-bit rotation scheme, our model can support up to 32 processors cores, since having a processor with higher number of cores results in duplicating the IV/SK values.

As keystream-synchronization is essential for the correctness of encryption and decryption operations.

The number of generated threads (represented by the attribute **ρ**) should be shared between sender and receiver. The value of **ρ** determines the number of threads that should be created on the receiver's side regardless the number of cores available on the receiver's processor. When the value of **ρ** at the sender's side (denoted by $\boldsymbol{\rho}_{sender}$) differ from the value of **ρ** at receiver's side (denoted by $\boldsymbol{\rho}_{receiver}$) the following two equations (29-30) are used to specify the total number of threads that should be associated with each processor core ($\hat{C}_i$):

$$\hat{C}_i \S (\boldsymbol{\rho}_{sender}/\boldsymbol{\rho}_{receiver}) threads \quad if\ \boldsymbol{\rho}_{sender} > \boldsymbol{\rho}_{receiver} \tag{29}$$

$$\hat{C}_{(i/\boldsymbol{\rho}_{sender})} \S (\boldsymbol{\rho}_{sender}/\boldsymbol{\rho}_{receiver}) threads\ if\ \boldsymbol{\rho}_{sender} < \boldsymbol{\rho}_{receiver} \tag{30}$$

By controlling the number of generated threads on both sides, one can assure that the keystream generated during the encryption and decryption are aligned.

4.3 Plaintext Encoder (PtE)

This component is responsible for XOR'ing the keystream bit generated by each thread with their corresponding plaintext bits. To achieve parallelism in **PtE**, the plaintext bit should be synchronized with the keystream bits. Based on the number of generated threads (**ρ**), each 128-bit of plaintext is associated with a specific thread sequentially, such that thread T_1 is associated with bits [b_0, …, b_{127}], T_2 is associated with bits [b_{128}, …, b_{255}] and so on. As the encryption process is parallelized in **PtE**, each thread needs to work on specific set of plaintext bits (plaintext segments). To achieve that, each thread use a special counter (C*tr*) that is incremented sequentially and independently from the other threads.The Ctr_0 values of all threads are initialized such that:

$$Ctr_0 = thread\_id \tag{31}$$

Consequently, the subsequent rounds will increment the value of Ctr_i as shown in Equation 32:

$$Ctr_{i+1} = Ctr_i + \boldsymbol{\rho} \tag{32}$$

Table 1 shows an illustration on the associated counter values Ctr_i and plaintext segments with its corresponding threads in three rounds of generating new keystream on 8-cores processor (**ρ**= 8).

Table 1: Association between the counters, plaintext segments and thread IDs

thread_id	1	2	3	4	5	6	7	8
Ctr_0	1	2	3	4	5	6	7	8
plaintext bits	0-127	128-255	256-383	384-511	512-639	640-767	768-895	896-1023
Ctr_1	9	10	11	12	13	14	15	16
plaintext bits	1024-1151	1152-1279	1280-1407	1408-1535	1536-1663	1664-1791	1792-1919	1920-2047
Ctr_2	17	18	19	20	21	22	23	24
plaintext bits	2048-2175	2176-2303	2304-2431	2432-2559	2560-2687	2688-2815	2816-2943	2944-3071

Generating the ciphertext $C_{T[m,\ldots,n]}$ for a segment of size 128 bits (for all bits from *m* to *n*) is carried out by XOR'ing the plaintext bits $P_{T[m,\ldots,n]}$ with their corresponding keystream bits $K_{S[m,\ldots,n]}$ as illustrated in Equation 33:

$$C_{T[m,\ldots,n]} = (P_{T[m,\ldots,n]} \S Ctr_i) \oplus (K_{S[m,\ldots,n]} \S T_i) \tag{33}$$

5 Security Analysis

In this research we rely on the keystream generator of Rabbit stream cipher to generate keystreams. Therefore, we are not performing further analysis on the security of the keystream generator itself. On the other hand, our modification on the original Rabbit cipher includes replacing the two textual keys (IV and SK) by a secret image, where the values of both IV/SK are extracted mathematically from the input image. The pre-calculation of IV/SK includes using the secure hash algorithm SHA-256 to increase the security of generating these two values. Such technique restrict cryptanalysis attacks (especially brute force attacks) from attacking the secret key due to the huge key space. Practically, the attacker must find the right combination of IV/SK keys in the space of $2^{128}\times2^{64}$ possible keys, which is considered infeasible on current computing powers.

From the other perspective, our modified algorithm requires the knowledge of the total number of cores used at the sender's side ($\boldsymbol{\rho}_{sender}$). This value should be securely shared along with the secret image in order to synchronize the decryption process at the receiver's side. This technique add one more layer of security to Rabbit stream cipher.

6 Performance Analysis

In order to analyze the performance gained from parallelizing Rabbit stream cipher, three different platform are chosen to run the experiments. The

processors specifications of the selected platform are as follows:

Platform 1: Intel Pentium IV® of CPU speed 1.93GHz (*single core*).

Platform 2: Intel Dual-Core® of CPU speed 2.93GHz (*two cores*).

Platform 3: Intel Core 2 Quad® of CPU speed 2.40 GHz (*four cores*).

Experiments results showed that the parallelized version (using *Posix* multithreading library) of Rabbit cipher (denoted by *p*-Rabbit) outperforms the original design (sequential design) of Rabbit cipher (denoted by Rabbit). The results shows that the new design is able to utilize the power of the available cores of the processors. Table 2 shows the encryption rate achieved on the three platforms.

Table 2: Association between the counters, plaintext segments and thread IDs

	Platform 1	*Platform 2*	*Platform 3*
Rabbit	**495**MB/S	**610**MB/S	**618**MB/S
***p*-Rabbit**	**702**MB/S	**985**MB/S	**1448**MB/S

It is obvious that our parallelized Rabbit utilizes the processor cores. The encryption rate of *p*-Rabbit achieved on quad-core processor is double the encryption rate of Rabbit on the same processor. Practically, achieving such high encryption rate is possible on multi-core processors due to the efficiency of the memory architecture (L1, L2 cache memories) on these processors if compared to multi-processors memory architecture. Fig. 7 illustrates the projected performance of Rabbit and *p*-Rabbit ciphers on multi-core processors of 8-cores and up to 32-cores (assuming CPU speed of 2.4GH).

4 Conclusion

In this research, a parallel design of Rabbit stream cipher is presented. The new design focused on parallelizing the keystream generator of Rabbit to generate multiple keystreams simultaneously. Parallelizing the keystream generator is based on multithreading technique, where *n* threads are generated to run their own copies of keystream generators. Moreover, our new design replaced the textual initial vector and secret key by an image. The image is used to extract the values of the IV/SK securely. However, the performance analysis revealed that the encryption rate of the parallelized version of Rabbit outperform the original design of Rabbit on different number of processors' cores. The average encryption rates enhancements were 1.4, 1.6 and 2.3 times over single, dual and quad core processors, respectively.

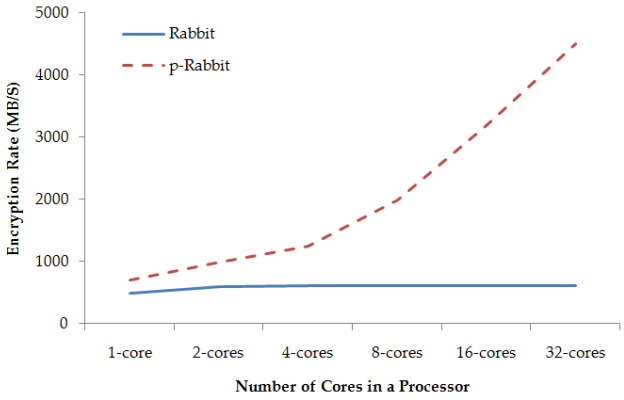

Fig. 7. Projected performance of Rabbit and *p*-Rabbit over multi-core processors

Acknowledgment

The author thanks the Arab Open University (AOU), Saudi Arabia Branch for supporting this study.

References:

[1] Lingling, S., Zhigang, J., & Zhihui, W. The Application of Symmetric Key Cryptographic Algorithms in Wireless Sensor Networks. *Physics Procedia*. 2012. *25* 552–559.

[2] Sabater, A. Computing Classes of Cryptographic Sequence Generators. *Procedia Computer Science*, 2013, 18 2440–2443.

[3] Al-Janabi, S., Rijab, K., & Sagheer, A. Video Encryption Based on Special Huffman Coding and Rabbit Stream Cipher. *2011 Developments in E-systems Engineering*, 2011, 413-418.

[4] Boesgaard, M. V., & Scavenius, O. Rabbit: A New High-Performance Stream Cipher. In *Fast Software Encryption*, 2003, 2887 307-329.

[5] Mollin, R. A. *An Introduction to Cryptography* (2nd Edition ed.). (K. H. Rosen, Ed.) Boca Raton: Chapman & Hall/CRC, 2007.

[6] Delfs, H. *Introduction to Cryptography: Principles and Applications.* Springer, 2002.

[7] SunSoft. Multithreaded Programming Guide. CA: Sun Microsystems, 2002.

[8] Gabb, H. *Common Concurrent Programming Errors*, 2002. Retrieved March 2, 2008, from Linux Magazine: www.linux-mag.com/content/view/983/2038/1/0/

[9] Sun Microsystems. *The Multithread Library*. Retrieved April 25, 2008, from:http://w3.mit.edu/sunsoft_v5.1/www/pascal/user_guide/mlthrd11.doc.html, 2009.

[10] Leroy, X. *The LinuxThreads library*. Retrieved April 26, 2008, from http://pauillac.inria.fr/~xleroy/linuxthreads/, 2006.

[11] Engelschall, R. S. *GNU Portable Threads*. Retrieved April 26, 2008, from http://www.gnu.org/software/pth/, 2006.

[12] Neufeld, C. *Threads, Bare-Bones*. Retrieved March 29, 2008, from ftp://caliban.physics.utoronto.ca/pub/linux/, 1996.

[13] Umamaheswari, G., & Shanmugam, A. Efficient VLSI implementation of the block cipher Rijndael algorithm. *Academic Open Internet Journal, 12*. 2004.

[14] Ciet, M., Neve, M., Jean, J., P., E., & Quisquater. Parallel FPGA Implementation of RSA with Residue Number Systems. *46th IEEE Midwest Symp. on Circuits and Systems*.2003.

[15] Swankoski, E. B., & Irwin, M. A Parallel Architecture for Secure FPGA Symmetric Encryption. *18th International Parallel and Distributed Processing Symposium (IPDPS'04) - Workshop 3*, 2004, p. 132-136.

[16] Nahum, E. O., & Schroeppel, R. Towards High Performance Cryptographic Software. *The Third IEEE Workshop on the Architecture and Implementation of High Performance Communications Subsystems (HPCS '95),*. Mystic, Conn. 1995.

[17] Henriquez, F. S., & A.Perez. A fast parallel implementation of elliptic curve point multiplication over GF(2^m). *Microprocessors and Microsystems*, 2004 , 329-339.

[18] Hoonjae, L., & Sangjae, M. Parallel stream cipher for secure high-speed communications. *Signal Processing*, 2002, 259-265.

[19] Jae, L. S., YoungHo, P., & Yong, K. On the m-Parallel Nonlinear Combine Functions for the Parallel Stream Cipher. *International Conference on Hybrid Information Technology, ICHIT'06*. Seoul, Korea, 2006.