Parallel Platform for Supporting Stream Ciphers Over Multi-core Processors

Sally Almanasra
Faculty of Computer Studies
Arab Open University (AOU)
Riyadh, Saudi Arabia

***Abstract*—Designing secure and fast cryptographic primitives is one of the critical issues in the current era. Several domains, including Internet of Things (IoT), military and banking, require fast and secure data encryption over public channels. Most of the existing stream ciphers are designed to work sequentially and therefore not utilizing available computing power. Also, other stream ciphers are designed based on complex mathematical problems which makes these ciphers slower due to the complex computations. For this purpose, a novel parallel platform for enhancing the performance of stream ciphers is presented. The platform is designed to work efficiently over multi-core processors using multithreading techniques. The architecture of the platform relies on independent components that can operate over multiple cores available on the corresponding communication ends. Two groups of stream ciphers were considered as case studies in our experiments. The first category includes stream ciphers of a sequential design, while the second category includes parallelizable stream ciphers. Performance tests and analysis shows that the parallel platform was able to maximize the encryption throughput of the selected stream ciphers dramatically. The enhancements on the encryption throughput is relative to the constructional design of the stream ciphers. Parallelized stream ciphers (Salsa20, DSP-128, and ECSC-128) was able to achieve higher throughput compared to other sequentially designed stream ciphers.**

Keywords—Stream ciphers; parallel computing; multithreading; cryptographic primitives; multi-core processors

I. Introduction

High-performance computing is progressively in demand in many day-to-day applications. Current computing resources present a tremendous opportunity for creating higher performance models through parallelism. The main concept of parallelism relies on allowing several tasks to be accomplished simultaneously and completed in a shorter period of time. Concurrent use of multiple processing resources is able to solve complex computational problems. A given problem is broken into smaller portions and solved concurrently using multiple computing units. To obtain the best of parallelism, scientists are focusing on faster hardware devices and processing techniques [1][2].

The multithreading technique is one processing technique that aims to create a virtual multiprocessor environment to execute multiple tasks on single processor [3][4]. The recent hardware revolution plays an important a role in improving system performance through multi-core technology. Multi-core processors are designed as a single physical processor that consists of the logical core of more than one processor.

Such processors' architecture enables the multi-core processor to run multiple tasks concurrently in order to achieve a higher performance compared to single-core processors. However, multi-core processors have a great advantage over multi-processors' architecture as a major proportion of intra-communication latency between communicating cores is minimized in multi-core processors, compared to the inter-communication latency carried out in multi-processor architectures. The architecture of multi-core processors is presented in Fig. 1.

Nowadays, a vast range of critical applications require a design of secure and high performance encryption algorithms to facilitate secure communication over public channels [5]. This is also an urgent issue for IoT solutions. In this research we aim to design a high-performance parallel platform to support stream cipher algorithms that depends on complex mathematical operations. Complex algorithms are known to perform slowly as internal operations require massive complex computations. In later sections we discuss some general concepts on parallel computing, which contribute to speeding up systems and applications.

The rest of the paper is organized as follows. Section 2 provides an overview of parallelism over multi-core processors. Section 3 presents the structure of stream ciphers. The design and structure of the parallel platform is introduced in Section 4. A number of performance tests were performed and results are presented in Section 5. A concluding remark is given in Section 6.

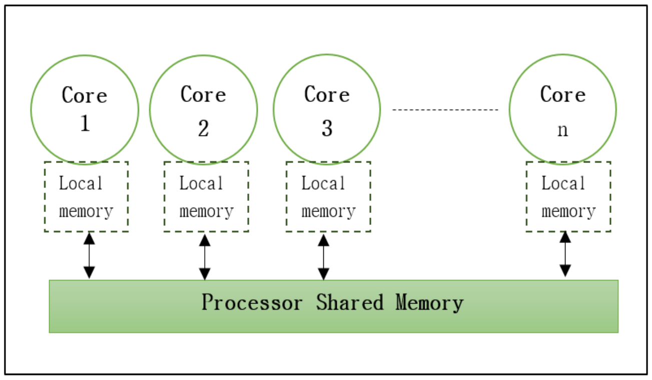


Fig. 1. The Architecture of Multi-Core Processors.

II. Parallelism over Multi-Core Processors

Multi-core computers are commonly used by individuals and enterprises. Software which are designed on a sequential base has become obstructive to performance. In order to make use of the extra cores, new algorithms must be designed in parallel bases. Such parallel designs effectively facilitate the utilization of multi-core processors.

Parallelism is usually presented in the form of threads. This technique maps independent tasks to threads at the lowest level [6]. Multithreading techniques aim to improve the performance of the running processes by allowing proper distribution of tasks among the available cores in a particular computing unit [7]. Fig. 2 illustrates the performance gained by applying multithreading techniques to a multi-core machine.

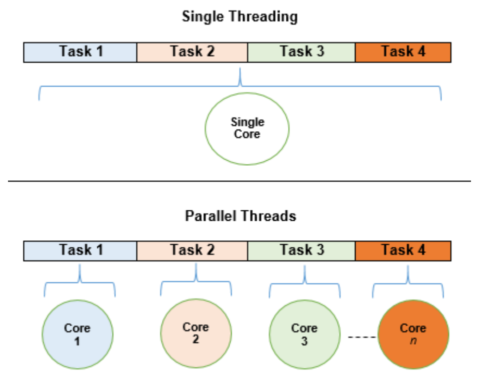


Fig. 2. Sequential Versus Parallel Execution over Multi-Core Processor.

Multithreading allows algorithms to execute several instructions per cycle, resulting in higher processor utilization and significant throughput speedup. Applying parallel techniques in cryptography has become essential for higher throughput and improved performance, especially with the current available resources. Therefore, in this study, we utilize multi-core technology with multithreading techniques to speed up the encryption process in stream ciphers, in order to provide secure and better-performing cryptosystems.

III. Symmetric-Key Encryption: Stream Ciphers

Stream ciphers are one of the cryptographic primitives that are used to secure communication over public and unsecured channels [8]. Stream cipher algorithms generate a pseudorandom keystream to encrypt a stream of plaintext, producing a stream of incomprehensible text known as ciphertext [9], as shown in Fig. 3.

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

$$E_{k_i}(m_i) = c_1, c_2, \cdots c_n \in C \qquad \forall i : 1 \leq i \leq n \qquad (1)$$

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.

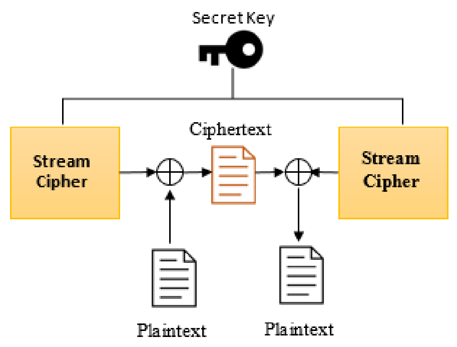


Fig. 3. Stream Cipher Algorithm.

The state of art reveals different designs of stream ciphers. The majority of existing stream ciphers are designed to run over single-core processors (e.g. RC4 [10], Sosemanuk [11]). Very few stream ciphers are designed to support parallelism. Examples of parallelized stream ciphers include Salsa20 [12] and ChaCha [13], DSP-128 [14] and ECSC-128 [15] stream ciphers.

However, the parallelized parts of these algorithms are restricted to some internal sections and do not focus on the general production of keystreams. In the next section, we introduce a parallel platform for supporting efficient parallelization of stream ciphers.

IV. Proposed Parallel Platform for Stream Ciphers

The proposed parallel platform is designed to support stream cipher operations on machines of different numbers of cores. For instance, the sender may encrypt a text on a machine with two cores, while the receiver may decrypt the text on a machine with any number of cores. The platform works flexibly regardless the number of cores on both sides. Our parallel platform is presented in Fig. 4.

The overall design of the parallel platform is divided into multiple parts to ensure maximum parallelism and a balanced workload among the available cores. In addition, the design of the platform also focuses on avoiding synchronization among the running threads for higher performance. This is possible through the use of multiple controllers in the platform.

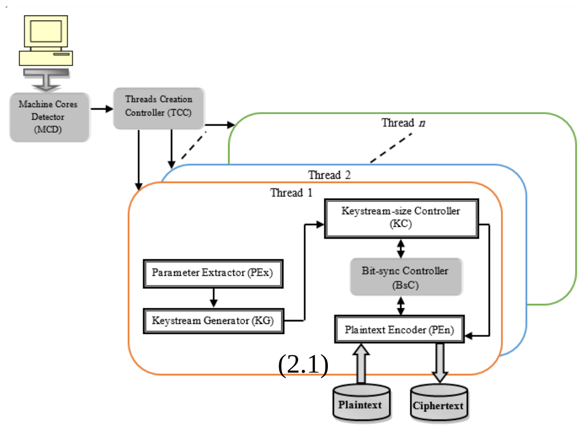

Machine Cores Detector (MCD)
Threads Creation Controller (TCC)
Thread n
Thread 2
Thread 1
Keystream-size Controller (KC)
Parameter Extractor (PEx)
Bit-sync Controller (BsC)
Keystream Generator (KG)
Plaintext Encoder (PEn)
(2.1)
Plaintext
Ciphertext

Fig. 4. The Architecture of the Parallel Platform.

The parallelized platform shown in Fig. 4 consists of several components, controllers and detectors. The main components of the platform are: Parameter Extractor (PEx), Keystream Generator (KG), Keystream-size Controller (KC) and Plaintext Encoder (PEn). In addition, the platform uses three supportive controllers and detectors: Machine Core Detector (MCD), Thread Creation Controller (TCC) and Bit-sync Controller (BsC). These controllers and detectors are designed to achieve the optimum level of performance gained from the parallel platform.

The parallel platform is also designed such that there is no direct-dependency among the components. In other words, the design of these components will enable us to easily parallelize the workload between the running threads. From another perspective, we have designed special detectors and controllers to ensure the consistency and accuracy of the keystream generation and plaintext encryption process.

As our platform may support different types of stream ciphers, the parallel platform is able to extract the corresponding parameters of the corresponding stream cipher. Accordingly, the Parameter Extractor component (PEx) extracts the required parameters to be used in other components, as shown in Fig. 5. The extracted parameters vary from one keystream generator to another in terms of the number of parameters, the size of the parameters and the representation of the parameters. For instance, if the DSP-128 stream cipher is selected, two parameters are extracted: parameter C (integer) of 128-bit length and parameter $\mathfrak{B}$ (polynomial) of degree 128.

The functionality of the Keystream Generator (KG) component is the most important part of the platform, whereby it is responsible for generating sequences of keystream bits. In this stage, a counter is used to increment the corresponding parameters, as shown in Fig. 6.

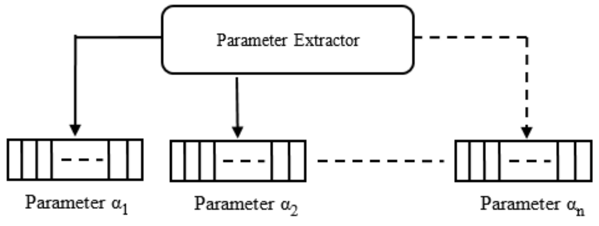


Fig. 5. Parameter Extractor (PEx) Component.

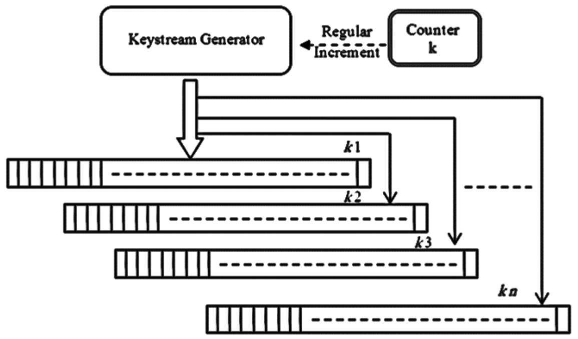


Fig. 6. Parallel Keystream Generated by the Keystream Generator (KG) Component.

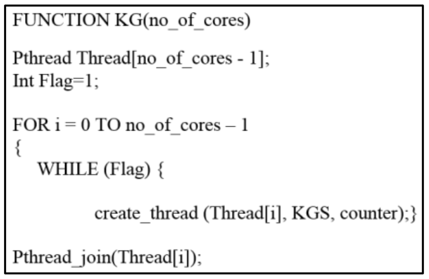
The generation process of the keystream bits is mainly dependent on the parameters obtained from PEx to be applied on the mathematical problem being used in the core of the KG component. Fig. 6 illustrates the process of producing *n*-keystreams in *n*-rounds associated with every increment carried out by the counter *k*. Applying parallelism on the KG component to generate multiple keystreams concurrently is possible with the existence of controllers, which will be discussed later in this section. However, the only task that the KG component has to accomplish at this stage is to generate multiple keystreams in parallel, based on the incremented value of the extracted counter parameter, as shown in Fig. 7.

The KG component will use the initial value of the *counter* and the thread number (Thread_ID) to increment the value of the *counter*. At this stage, the KG component creates *n* threads (where *n = no. of_cores*) to handle the generation of new keystreams, concurrently.

Variable keystream lengths might be generated from the KG stage owing to the differences in the deployed stream cipher. Therefore, the Keystream-size Controller (KC) component is designed to standardize and control the size of the generated keystreams and limit their size to a 32-bit length.

As shown in Fig. 8, the keystream controller maps the *n*-bits of the reformatted key to a fixed-size key of *m*–bits length. The mapping process is known as pre-encryption processing. The size of the keystream can vary from one byte to five bytes. If the size of *n* is greater than 32 bits, **KC** will truncate the keystream to 32 bits and uses the rest of the bits in the following round, as presented by Equation (2):

$$K_{new} = Trunc\ (K_S, m) \qquad (2)$$



```
FUNCTION KG(no_of_cores)

Pthread Thread[no_of_cores - 1];
Int Flag=1;

FOR i = 0 TO no_of_cores – 1
{
    WHILE (Flag) {

            create_thread (Thread[i], KGS, counter);}

Pthread_join(Thread[i]);
```

Fig. 7. The Code Snippet of KG in the Parallel Platform.

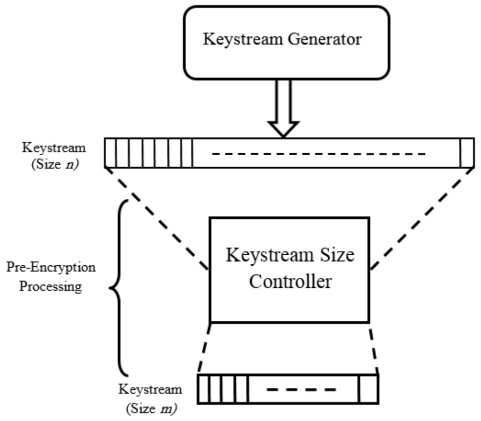


Fig. 8. Keystream-Size Controller (KC) Component.

where *m* is the upper limit of the size of the keystream (e.g. 32-bit). The content of the new keystream is given by Equation (3):

$$K_{new} = K_s[i], \text{ for all } 0 < i < m \quad (3)$$

The process of setting up the length of the final keystream is also known as the pre-encryption process, since the next component will use this key for data encryption. However, the importance of this component comes from the process of standardizing the size of the keystream regardless of the selected generator, which in turn gives the platform additional flexibility.

Consequently, the Plaintext Encoder (PEn) component is carried out. The main task of this component is to encrypt a sequence of plaintext and produce a corresponding ciphertext. The input to this component is the keystream generated by the KG component, as shown in Fig. 9. The encryption process is performed as follows: One word (32-bit) of plaintext is XORed with one word of the keystream. When PEn employs all the bits of the keystream, the KG component will be invoked to generate a new round of keystreams. Equation (4) forms the condition which needs to be satisfied when calling for the KG in a new round:

$$\text{Call (KG)}: \sum \eta_{K_s} < \sum \eta_{P_t} \quad (4)$$

where η refers to the unused (available) bits.

Unlike other components, the encryption component (PEn) has a direct dependency on the keystream generator in which PEn must keep checking the number of available bits of the keystream in order to trigger the KG if the number of bits in the keystream is insufficient to perform the encryption.

The five components described above form the basic design of the proposed platform. The next step will parallelize these components and add other controllers and detectors, while applying multithreading techniques on multi-core processors for a fully parallelized platform.

Parallelizing the platform requires a detector called a Machine Cores Detector (MCD) to detect the number of cores on the corresponding machine. The MCD works at low hardware level in which it detects the total number of logical processors (cores) in a particular machine. We refer to the total number of cores as the NOC. The MCD is associated with a Thread Creation Controller (TCC) to create a specific number of threads, as many as the number of the available cores detected by the MCD. The other task of the TCC is to bind each job assigned to a thread with its correspondent core to isolate any potential concurrency issues (e.g. synchronization, system bottleneck, etc.) at the thread level. Accordingly, the TCC allows all jobs to run in parallel, in order to maximize the performance gains from the multi-core processors.

Binding each thread with one specific core is achieved by changing the scheduling policy by calling the processor's affinity routine. Processor affinity is designed to force threads to work on a specific core during the run. This is possible by using the POSIX threading library that provides developers with the one routine known by *pthread_setaffinity_np*. The default scheduling policy usually switches threads from one core to another during the run of multiple threads. Therefore, setting the processor affinity is sufficient to avoid thread switching since thread switching requires copying the thread instructions from its current L1 cache to the L2 cache of the new (switched) core.

For instance, in dual-core processors, the thread creation controller will create two threads and associate them with the two cores on that machine. The two cores share several resources and peripherals on a high-speed on-chip bus except the L1-cache, which is designated for each core. An example of an MCD and TCC implemented on a dual core processor is shown in Fig. 10.

For consistency, the keystream generation component and the plaintext encoder component are associated with an additional controller to ensure correct data encryption of each plaintext byte encrypted by its corresponding keystream. We refer to this controller as a Bit-sync Controller (BsC). This controller will ensure the synchronization between each bit in the plaintext with the corresponding keystream bit, for a correct decryption process. The encryption process $\boldsymbol{E}$ controlled by BsC is described in Equation (5):

$$C_t = E_{BsC(K_s)}(BsC(P_t)) \quad (5)$$

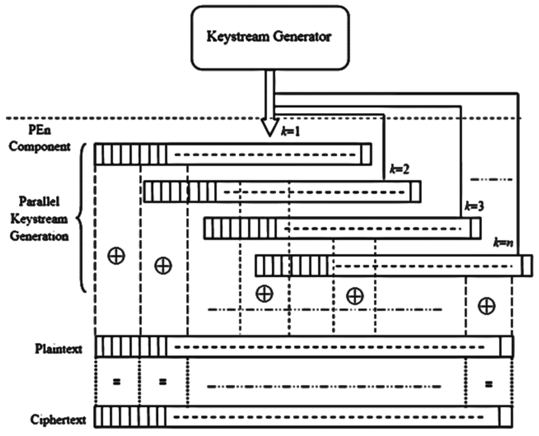


Fig. 9. Plaintext Encoder (PEn) Component.

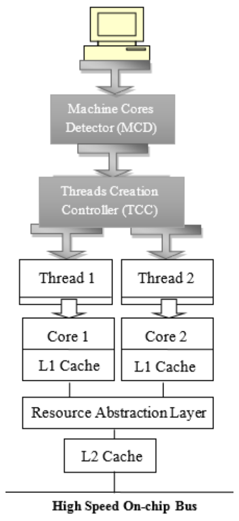

Machine Cores Detector (MCD)
Threads Creation Controller (TCC)
Thread 1
Thread 2
Core 1
L1 Cache
Core 2
L1 Cache
Resource Abstraction Layer
L2 Cache
High Speed On-chip Bus

Fig. 10. MCD and TCC in Dual-Core Processor.

where C_t and P_t are the ciphertext and plaintext bytes respectively.

The BsC controller associates each 32-bit of plaintext with each thread so that the parallel threads can run concurrently. The pseudocode of BsC is shown in Fig. 11, and visualized by Fig. 12.

This algorithm is applicable to work on *n*-core processors, which provides a platform with higher scalability with regards to the rapid growth of the processor architecture.

Associating the keystream with the running threads is conceptually controlled in the same way as associating threads with their corresponding plaintext segment. There are two important equations used to increment the counter value associated with each round in the KG component. Let $\alpha_{counter}$ be the initialized counter extracted in PEx, the new initial value of $\alpha_{counter}$ is calculated and stored in a new counter denoted by *Ctr* as shown by Equation (6):

$$\mathrm{Ctr}_0 = \alpha_{counter} + \mathrm{Thread\_ID} \qquad (6)$$

```
//Initial Parameters
NOB = 32 //number of bits
Flag = 1
bit_counter = NOB * (Thread_ID -1) +1
WHILE (more plaintext) //read plaintext for encryption
{
    IF (Flag)
        FOR bit_counter TO NOB*Thread_ID
            P[bit_counter] = plaintext[bit_counter]
    Flag = 0
    bit_counter = bit_counter + (NOB * (NOC-1)) + 1
    limit = bit_counter + (NOB – 1)
    FOR bit_counter TO limit
        P[bit_counter] = plaintext[bit_counter]
}
```

Fig. 11. The Code Snippet of a BsC Controller.

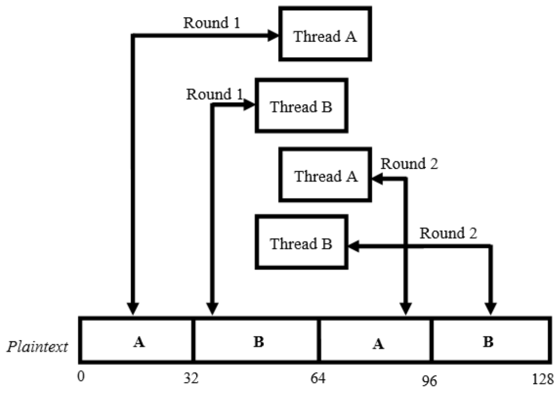


Fig. 12. Bit-Sync Controller (BsC).

The subsequent rounds will increment the value of *Ctr* as shown by Equation (7):

$$\mathrm{Ctr}_{j+1} = \mathrm{Ctr}_j + \mathrm{NOC} \qquad (7)$$

When the keystream generation component is requested to perform a new round, calculating *Ctr* by Equation (7) will be carried out. Table I shows an illustration of the associated counter values and plaintext segments with corresponding threads for three rounds of generating new keystreams in an 8-core processor. Let $\alpha_{counter}$ = 0, the incremented *Ctr* with its associated plaintext segments will be as follows:

Based on Table I, the thread with ID 1 will call KG three times and it will accordingly use the values 1, 9 and 17 as its counter values to generate three new 32-bit keystreams. The three generated keystreams by thread 1 will be associated with bits 1 to 32 (1-32), 257-288 and 513-544 of the plaintext, respectively. The functionality of the BsC controller is described in Fig. 13 where each KG component is associated to each of the existing cores.

Note that the plaintext segments are associated with a specific keystream generated by its corresponding counter.

However, the design of the platform and the plaintext encoder will ensure the correct sequence of the ciphertext segments, as shown in Fig. 14.

TABLE. I. ASSOCIATION TABLE BETWEEN THE COUNTER, PLAINTEXT SEGMENT AND THREAD ID

Thread_ID	1	2	3	4	5	6	7	8
Ctr_0	1	2	3	4	5	6	7	8
Plaintext Bits	1-32	33-64	65-96	97-128	129-160	161-192	193-224	225-256
Ctr_1	9	10	11	12	13	14	15	16
Plaintext Bits	257-288	289-320	321-352	353-384	385-416	417-448	449-480	481-512
Ctr_2	17	18	19	20	21	22	23	24
Plaintext Bits	513-544	545-576	577-608	609-640	641-672	673-704	705-736	737-768

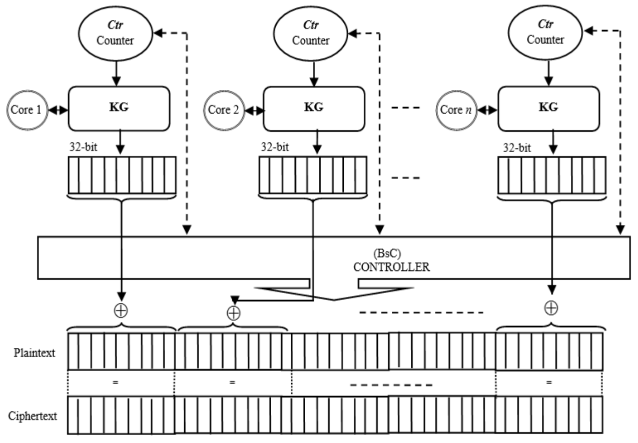


Fig. 13. Parallel Keystream Generation Controlled by BsC.

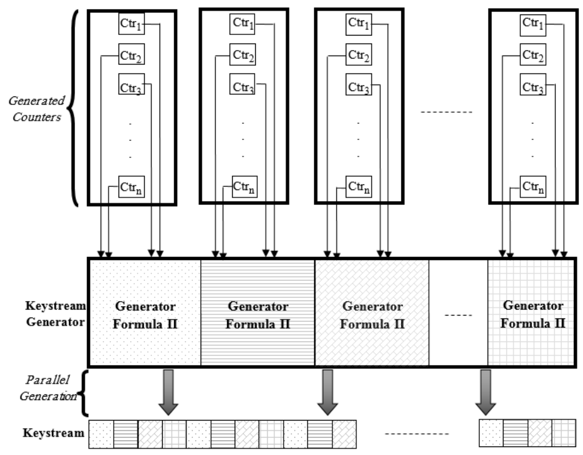


Fig. 14. Concurrency and Consistency in Parallel Keystream.

Fig. 14 illustrates the process of generating multiple independent and parallel counters by multiple threads. Technically, a copy of selected generator KG is associated to each core of the multi-core processor. Accordingly, each copy of the generator will use these independent counters to generate sequences of keystreams.

In this research, we present a high scalable platform that is capable of working on different numbers of cores on multi-core processor. The uniqueness of this parallel platform is that one can encrypt a stream of plaintext on *n*-core processors and decrypt the ciphertext on *m*-core processors (where $n \neq m$), as illustrated in Fig. 15 and 16 (for *n*=2 and *m*=4 respectively). This is due to the flexibility and the design of the platform's structure, which ensure the correctness of the encryption and decryption processes on any number of cores.

The process of matching a specific keystream sequence for a specific plaintext segment is a critical task that depends on the appropriate use of the counters. This task must be designed properly owing to its importance in allowing users to encrypt and decrypt their data on different numbers of cores. The following is an example (Example 1) of encrypting plaintext bits using keystreams generated by the keystream generator. The encryption is performed on a dual-core processor, while the decryption is performed on a quad-core processor (Example 2).

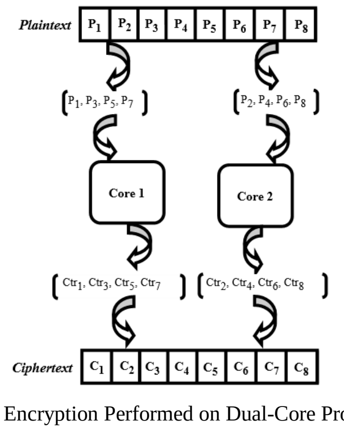


Fig. 15. Encryption Performed on Dual-Core Processor.

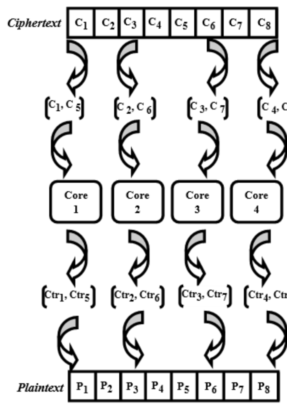


Fig. 16. Decryption Performed on Quad-Core Processor.

Example 1: Let P_t be the plaintext with length of five bytes (40 bits) such that P_t = 1101110111011011111000000111111110011111. The encryption of P_t on a dual-core processor with an 8-bit segment is performed as follows:

First, we divide the plaintext into segments of 8-bit length as follows:

	***Segment* (1)**	***Segment* (2)**	***Segment* (3)**	***Segment* (4)**	***Segment* (5)**
$P_t =$	11011101	11011011	11100000	01111111	10011111

Subsequently, each thread associated with its corresponding core will generate a unique value of the counter *C* to be later used to encrypt a specific segment of the plaintext, as shown in Table II.

The resulting ciphertext is formed such that the position of the ciphered segment in the final ciphertext is based on the order of its corresponding counter. The ciphertext C_t will be in the following form:

	***Segment* (1)**	***Segment* (2)**	***Segment* (3)**	***Segment* (4)**	***Segment* (5)**
$C_t =$	01110000	11001000	11000111	10111010	1101000

TABLE. II. GENERATING NEW COUNTER VALUE FOR ENCRYPTION ON EACH THREAD

	Core Number	**Counter C**	**KG $\mathbf{k = \Omega^C}$**	**Ciphertext Segment $\mathbf{C_t}$**	**Plaintext $\mathbf{P_t = k \oplus C_t}$**
***Thread* (1)**	1	1	10101101	Segment (1) 11011101	01110000
***Thread* (1)**	1	3	00100111	Segment (5) 11100000	11000111
***Thread* (2)**	1	5	11110111	Segment (2) 10011111	01101000
***Thread* (3)**	2	2	00010011	Segment (3) 11011011	11001000
***Thread* (4)**	2	4	11000101	Segment (4) 01111111	10111010

One can see the association between the threads and the cores, such that thread 1 with counters 1, 3 and 5 will be executed on core 1, while thread 2 with counters 2 and 4 will be executed on core 2. Another important issue is related to the association between counter *C*, keystream and plaintext segments. This association will assure the consistency and coherence between the interchangeable encryption/decryption processes on a different number of cores.

However, decrypting the ciphertext above on quad-core processor is possible, and the following example (Example 2) shows the relation between the running threads and the ciphertext segments.

Example 2: Let $\boldsymbol{C_t}$ be the ciphertext of five bytes (1 byte = 8bits) such that $\boldsymbol{C_t}$ = 01110000110010001100011110111010 1101000. The decryption of $\boldsymbol{C_t}$ on a quad-core processor with an 8-bit/segment is performed as in Table III:

Similar to the encryption process in Example 1, the plaintext is formed such that the position of the plaintext segment in the final plaintext is based on the order of its corresponding counter. The plaintext P_t will be in the following form:

	Segment (1)	*Segment* (2)	*Segment* (3)	*Segment* (4)	*Segment* (5)
P_t =	11011101	11011011	11100000	01111111	10011111

Fig. 17 visualizes the relationship between multiple threads associated with multiple cores performing data encryption and decryption. However, the previous sub-sections have discussed the functionality of each component in order to understand the connection between those components running on multiple threads, forming a parallelized platform.

The proposed platform is mainly targeted for stream ciphers based on complex mathematical problems due to their high security attributes. The platform is designed to provide the opportunity for a more secure stream cipher to be designed regardless of the speed since the platform is able to provide those stream ciphers with higher efficiency and throughput. The platform is practical and has a great impact on the field of information security systems and cryptography.

TABLE. III. GENERATING NEW COUNTER VALUE FOR DECRYPTION ON EACH THREAD

	Core Number	**Counter C**	**KG $\mathbf{k = \Omega^C}$**	**Ciphertext Segment $\mathbf{C_t}$**	**Plaintext $\mathbf{P_t = k \oplus C_t}$**
***Thread* (1)**	1	1	10101101	Segment (1) 01110000	11011101
***Thread* (1)**	1	5	11110111	Segment (5) 01101000	10011111
***Thread* (2)**	2	2	00010011	Segment (2) 11001000	11011011
***Thread* (3)**	3	3	00100111	Segment (3) 11000111	01111111
***Thread* (4)**	4	4	11000101	Segment (4) 10111010	10011111

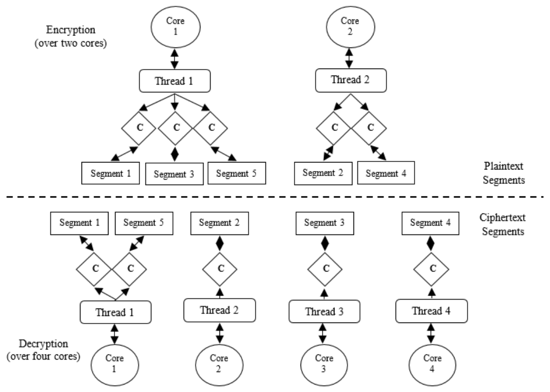


Fig. 17. Performing Encryption and Decryption on a different Number of Cores.

V. Security and Performance Analysis

Evaluating the efficiency of our parallel platform is measured against a set of stream ciphers. These stream ciphers are divided into two categories: parallelized and sequential ciphers. The chosen ciphers in our experiments are summarized in Table IV. The main reason of choosing these two categories of stream cipher is to examine the efficiency of these algorithms in utilizing the capabilities of the parallel platform.

The parallel platform is also measured from the security perspective. We analyze the impact of the parallel platform on the security attributes of the stream ciphers running over the platform. The platform is designed such that it does not affect the security attributes of the plugged-in stream ciphers, since each core is responsible for executing its own workload independently. Hence, there will be no interaction or dependency between any two or more keystream generators running over multiple cores, due to the avoidance of using global shared variables between the running threads.

The security level of the parallelized keystream generation on independent cores is similar to the security level of the sequential version of the stream ciphers. Technically, there are no shared parameters among the cores, which prevents any attempt to criticize the parallel platform on the security of the stream ciphers.

For testing purposes, we ran our model on two workstations. The first workstation (denoted by DualC) used an Intel Core 2 Duo ® E6400 processor of CPU speed 2.13 GHz, L2 cache memory of size 2MB, RAM of size 2GB. The second machine (denoted by QuadC) used an Intel Core 2 Quad ® Q6600 processor of CPU speed 2.40 GHz, L2 cache memory of size 8MB, RAM of size 2GB. The parallel platform was coded in C++ using MinGW-2.05 and tested on Microsoft Windows XP® operating system. POSIX-2.8.0 (*Pthread*) library was used to handle thread-related functions of the model.

Our testing started by examining the performance of the selected stream ciphers over DualC and QuadC machines. The results presented in Fig. 18 and 19 illustrate the performance of the stream cipher running over DualC and QuadC, respectively. Four sets of plaintext have been considered, of the sizes: 100, 500, 1000 and 2000 Mbits.

TABLE. IV. LIST OF STREAM CIPHERS CONSIDERED IN PERFORMANCE ANALYSIS

Stream ciphers	Category
Salsa20	*Parallelized*
DSP-128	*Parallelized*
ECSC-128	*Parallelized*
RC4	*Sequential*
Sosemanuk	*Sequential*

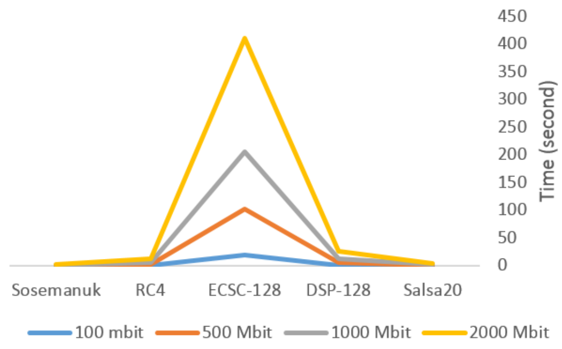


Fig. 18. Performance of Stream Ciphers over DualC Machine.

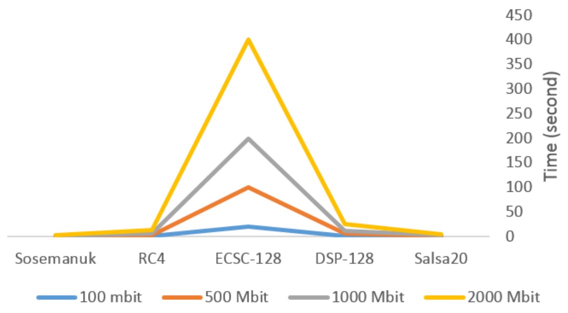


Fig. 19. Performance of Stream Ciphers over QuadC Machine.

Obviously, ECSC-128 is the slowest algorithm among the other algorithms. The results also show that little difference was found on the performance of the stream ciphers running on the quad-core machine compared to the encryption rates obtained on dual-core machines. The utilization of the two extra cores is not well identified by the selected stream ciphers. Table V presents the performance enhancements gained on the QuadC machine compared to the DualC machine.

On the next step, we plugged in the stream ciphers to our parallel platform (denoted by P(*stream cipher*)) to examine the impact of the platform on enhancing the encryption rates of these ciphers. Fig. 20 and 21 presents the results of running the five stream ciphers over the DualC and QuadC machines, respectively.

According to the performance analysis, we found that the parallel platform managed to support the parallelizable stream ciphers to utilize the two cores available on the DualC machine. Table VI shows that the parallel platform was able to enhance the encryption of the three parallelizable stream ciphers dramatically. The encryption rates of Salsa20, DSP-128 and ECSC-128 are enhanced by approximately 31%, 28% and 34%, respectively. However, the sequential stream ciphers are not capable of utilizing the support of the parallel platform.

TABLE. V. ENHANCEMENT RATIO GAINED FOR THE ORIGINAL STREAM CIPHERS RUNNING OVER QUADC MACHINE COMPARED TO DUALC MACHINE

Stream cipher	Enhancement Ratio
Salsa20	11%
DSP-128	6%
ECSC-128	2%
RC4	8%
Sosemanuk	9%

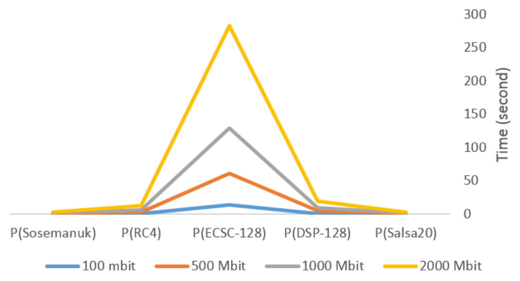


Fig. 20. Performance of Plugged Stream Ciphers on the Parallelized Platform over the DualC Machine.

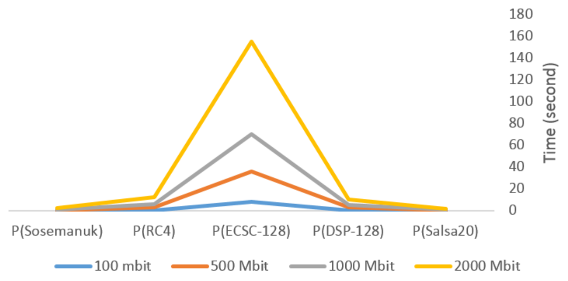


Fig. 21. Performance of Plugged Stream Ciphers on the Parallelized Platform over the QuadC Machine.

TABLE. VI. ENHANCEMENT RATIO GAINED FOR THE STREAM CIPHERS PLUGGED INTO THE PARALLEL PLATFORM AND RUNNING OVER DUALC MACHINE

Stream cipher	Enhancement Ratio
Salsa20	31%
DSP-128	28%
ECSC-128	34%
RC4	6%
Sosemanuk	10%

Similarly, the performance analysis shows that the parallel platform managed to support the parallelizable stream ciphers to utilize the four cores available on the QuadC machine. Table VII shows that the parallel platform was able to significantly enhance the encryption of the three parallelizable stream ciphers. The encryption rates of Salsa20, DSP-128 and ECSC-128 were enhanced by approximately 64%, 55% and 62%, respectively. However, the sequential stream ciphers are not capable of utilizing the support of the parallel platform.

To examine the efficiency of the parallel platform in utilizing the extra cores of QuadC compared to DualC, we compare the efficiency of the parallel platform over DualC and QuadC machines. Unlike the sequential stream ciphers (RC4 and Sosemanuk), results in Table VIII shows that an extra two cores doubled the encryption speed of the other parallelizable stream ciphers.

TABLE. VII. Enhancement Ratio Gained for the Stream Ciphers Plugged into the Parallel Platform and Running over QuadC Machine

Stream cipher	Enhancement Ratio
Salsa20	64%
DSP-128	55%
ECSC-128	62%
RC4	7%
Sosemanuk	17%

TABLE. VIII. Enhancement Ratio Gained by the Parallel Platform Running over QuadC Compared to DualC Machines

Stream cipher	Enhancement Ratio
Salsa20	33%
DSP-128	27%
ECSC-128	28%
RC4	1%
Sosemanuk	7%

We conclude that the design of the stream ciphers plays an important role in utilizing multi-core processors. The parallel platform is able to enhance the encryption rate significantly on the QuadC machine with four cores, while the sequential stream ciphers failed to utilize such computing resources. Fig. 22-26 illustrates the efficiency of the stream ciphers over different environments, where Seq-DualC and Seq-QuadC refer to running the original stream ciphers on DualC and QuadC machines, and Parallel(DualC) and Parallel(QuadC) refers to running the stream ciphers with the support of the parallel platform on DualC and QuadC machines.

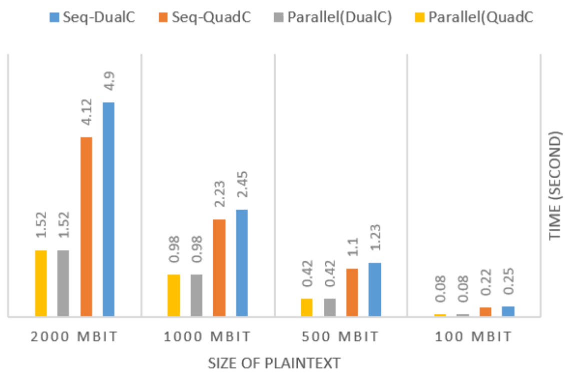


Fig. 22. Performance Efficiency of Salsa20 for different Environments.

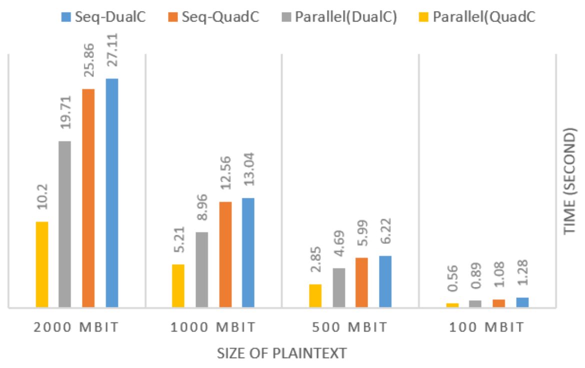


Fig. 23. Performance Efficiency of DSP-128 for different Environments.

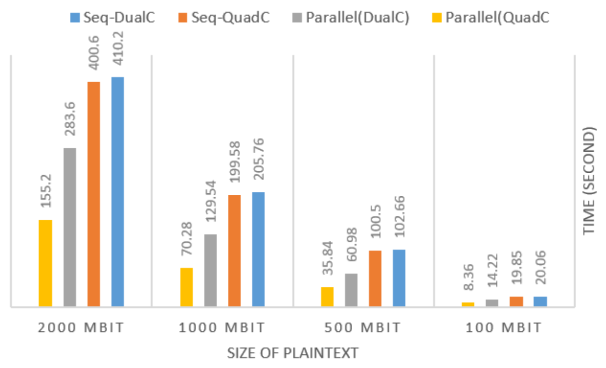


Fig. 24. Performance Efficiency of ECSC-128 for different Environments.

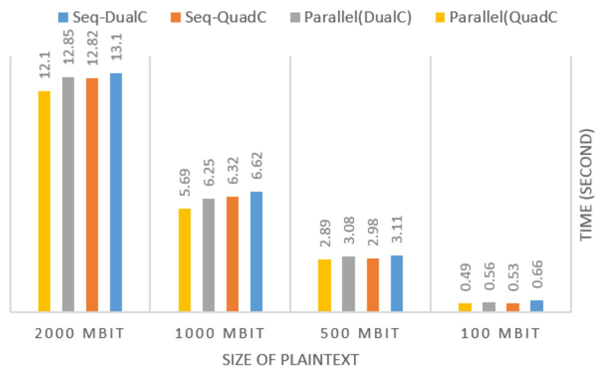


Fig. 25. Performance Efficiency of RC4 for different Environments.

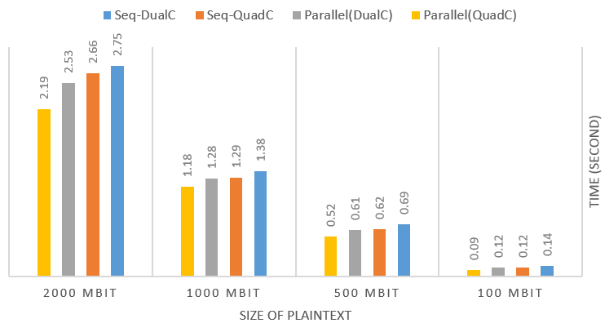


Fig. 26. Performance Efficiency of Sosemanuk for different Environments.

VI. Conclusion

In this paper we present a novel parallel platform to enhance the performance of stream ciphers. The underlying architecture of the platform relies on the use of multithreading technology. The platform is designed to be scalable and adaptable to the increasing number of cores in the future. Parallelism on our platform is implemented at two levels: task and data parallelism. Task parallelism is achieved by dividing the workload among the available cores in the corresponding machine, where each core will have its own components and parameters set. On the other hand, data parallelism is achieved by encrypting smaller sets of plaintext in multiple cores, concurrently.

The experiments' results show that parallel stream ciphers (Salsa20, DSP-128, ECSC-128) are capable of achieving higher performance on the parallel platform. The results also show that increasing the number of cores from two to four cores has doubled the performance of these three algorithms. This is due to the parallelizable design of these ciphers. However, sequential stream ciphers (RC4, Sosemanuk) are not able to utilize the support of the parallel platform running over the quad-core machine.

From the security perspective, the underlying architecture of the parallel platform is constructed to avoid the existence of shared global of local attributes between the running keystream generators. Each core is associated with one independent set of data and operates over separate input keys and counters. Accordingly, the parallel platform does not affect the security of the plugged-in stream ciphers.

ACKNOWLEDGMENT

The author would like to express her thanks to Arab Open University, Saudi Arabia for supporting this study.

REFERENCES

[1] Nutaro, J., and B. Zeigler. "How to apply Amdahl's law to multithreaded multicore processors." Journal of Parallel and Distributed Computing 107: 1-2, 2017.

[2] Dang, H., M. Snir, and W. Gropp. "Eliminating contention bottlenecks in multithreaded MPI." Parallel Computing 69: 1-23. 2017.

[3] Thébault, L., and E. Petit. "Asynchronous and multithreaded communications on irregular applications using vectorized divide and conquer approach." Journal of Parallel and Distributed Computing 16-27: 16-27. 2018.

[4] Soni, V., A. Hadjadj, O. Roussel, and G. Moebs. "Parallel multi-core and multi-processor methods on point-value multiresolution algorithms for hyperbolic conservation laws." Journal of Parallel and Distributed Computing 123: 192-203. 2019.

[5] Ogiela, M. "Cognitive solutions for security and cryptography." Cognitive Systems Research 55: 258-261. 2019.

[6] Hiscock, T., O. Savry, and L. Goubin. "Lightweight instruction-level encryption for embedded processors using stream ciphers." Microprocessors and Microsystems 64: 43-52. 2019.

[7] CharlesLai, B., K. Li, and C. Chiang. "Self adaptable multithreaded object detection on embedded multicore systems." Journal of Parallel and Distributed Computing 78: 25-38. 2015.

[8] Crainicu, B. "Unified Formal Model for Synchronous and Self-Synchronizing Stream Ciphers." Procedia Engineering 181: 620-625. 2017.

[9] Khelifi, F. "On the security of a stream cipher in reversible data hiding schemes operating in the encrypted domain." Signal Processing 143: 336-345. 2018.

[10] Rivest, R. The RC4 Encryption Algorithm. RSA. Document No. 003-013005-100-000000, USA: Data Security Inc. 1992.

[11] Berbain, C., O. Billet, A. Canteaut, N. Courtois, H. Gilbert, L. Goubin, A. Gouget, et al. "Sosemanuk, A Fast Software-oriented Stream Cipher." Accessed June 22, 2019. http://www.ecrypt.eu.org/stream/p3ciphers/sosemanuk/sosemanuk_p3.pdf. 2005.

[12] Bernstein, D. "The Salsa20 Family of Stream Ciphers." In New Stream Cipher Designs, 84-97. Berlin: Springer-Verlag. 2008.

[13] Bernstein, D. "The ChaCha family of stream ciphers." D. J. Bernstein's webpage. Accessed June 20, 2019. http://cr.yp.to/chacha.html. 2005.

[14] Suwais, K., and A. Samsudin. "DSP-128: Stream Cipher Based On Discrete Log Problem And Polynomial Arithmetic." American Journal of Applied Sciences 5 (7): 896-904. 2008.

[15] Suwais, K., and A. Samsudin. "ECSC-128: New Stream Cipher Based on Elliptic Curve Discrete Logarithm Problem." First International Conference on Security of Information and Networks. Famagusta. 13-23. 2007.