Performance Analysis of HC-128 and Salsa20/12 eSTREAM algorithms

Harsh Rupesh Shah
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada
harshrupeshshah@cmail.carleton.ca

***Abstract*— In this study, Python is used on a Windows computer to analyze the performance of the HC-128 and Salsa20/12 eSTREAM algorithms. Assessment of the algorithms' throughput and latency under a variety of circumstances, such as different message sizes and key lengths has been carried out. The networking and cryptography package, which offers a practical interface to cryptographic primitives, is used to create the algorithms. To complete the performance analysis, variables including execution time, code size, CPU utilization, and memory usage are analyzed.**

Keywords*— *HC-128, Salsa20/12, eSTREAM, stream ciphers, Python, nacl, Windows, performance analysis, throughput, latency, message sizes, code size, execution time, statistical analysis.

I. INTRODUCTION

One significant group of symmetric-key encryption algorithms, known as stream ciphers, is used extensively in several fields, including wireless communication, secure messaging, and video streaming [1]. HC-128 and Salsa20/12 are two well-known stream ciphers that are a part of the eSTREAM portfolio. These algorithms have undergone significant research and comparison in terms of performance, and they are well renowned for their ease of use, effectiveness, and high levels of security.

In this study, Python is used on a Windows PC to analyze the performance of the HC-128 and Salsa20/12 algorithms. Python is a well-known programming language that offers a high-level access to cryptographic libraries, making it the perfect tool for assessing how well cryptographic methods function. The algorithms are implemented using the nacl package, a Python implementation of different cryptographic primitives.

The performance study is done in terms of 5 important metrics: code size, CPU usage, memory consumption, and encryption and decryption times. The amount of time needed to use a cryptographic algorithm to encrypt or decrypt a message is referred to as encryption and decryption speed. Although faster encryption and decryption times are preferred, security must always come first. The amount of computing power needed to carry out a cryptographic technique is referred to as CPU consumption. For systems with constrained computing resources, lower processing power requirements are preferred. The amount of memory needed to run a cryptographic algorithm is referred to as memory utilization. For systems with constrained memory resources, lower memory utilization is preferred. The quantity of code needed to implement the algorithm is referred to as the algorithm's code size. Both the number of lines of code and the quantity of memory needed to hold the code are included in this. Generally speaking, a smaller code size can result in quicker execution times, less memory utilization, and less power usage. A cryptographic algorithm's security might be compromised if its code size is decreased, though, as lower code sizes might make it simpler for attackers to find weaknesses or carry out assaults like side-channel attacks [2].

Based on the performance requirements, the study's findings can assist developers and security experts in choosing the best stream cipher for their particular use case. The rest of this essay is structured as follows: A summary of the HC-128 and Salsa20/12 algorithms is given in Section 2. The approach used to implement and assess the algorithms is described in Section 3. The performance analysis findings are presented in Section 4, and the study's discussion and conclusion are presented in Section 5.

II. BACKGROUND INFORMATION

A. HC-128

In the eSTREAM portfolio of cryptographic algorithms, the stream cipher HC-128 was chosen. Hongjun Wu, the same person who created the HC-256 stream cipher, is the creator of this system. High security is offered by HC-128 while maintaining a manageable code size and quick execution rates. The 128-bit key length, 128-bit initialization vector, and 32-bit word size are the main characteristics of HC-128. It works by creating a keystream by combining nonlinear filtering with linear feedback shift registers (LFSRs). The ciphertext is then created by bitwise XORing the keystream and plaintext together [3]. The output feedback function, a nonlinear filtering technique used by HC-128, is one of the device's distinctive characteristics (OFF). To increase the security of the algorithm, nonlinearity is added to the keystream creation process using the OFF function. Further enhancing the algorithm's security are the LFSRs employed in HC-128, which are created to offer superior statistical features [3]. For its security and performance attributes, HC-128 has undergone significant research and testing. Several attacks, such as differential and linear cryptanalysis, have shown it to be extremely secure. HC-128 is renowned for its high performance, particularly for short message payloads, and low latency. However, especially for bigger message sizes, its throughput performance might not be as good as that of some other stream ciphers.

B. Salsa20/12

Daniel J. Bernstein created the Salsa20/12 stream cipher, which is popularly used in many applications because of its ease of use, high level of security, and superior performance [4]. It belongs to the Salsa20 family of stream ciphers, which also includes Salsa20/8 and Salsa20/20, which generate the keystream using various numbers of rounds. A 256-bit key length, a 64-bit initialization vector, and a 32-bit word size are

some of Salsa20/12's standout characteristics. It works by producing a keystream through a 20-round combination of permutation and substitution operations. The ciphertext is then created by bitwise XORing the keystream and plaintext together. Salsa20/12 has undergone thorough research and testing to ensure that it possesses good security against a range of attacks, including differential and linear cryptanalysis. Salsa20/12's usage of the "double-round" operation, which increases the algorithm's resistance to specific sorts of assaults, is one of its distinctive characteristics. Salsa20/12 has a reputation for having a fast throughput and low latency, especially for big message sizes. It is widely used in a variety of applications, such as secure communications, VPNs, and disc encryption, and has been optimized for both software and hardware implementations [4]. Several developers and security experts find Salsa20/12 to be an appealing option due to its simplicity and efficiency.

III. FRAMEWORK SET-UP AND TOOLS

A suitable development environment and relevant tools for benchmarking and profiling the algorithms have been chosen in order to examine the performance of HC-128 and Salsa20/12 stream ciphers using Python on a Windows system.

A. Development environment

Python 3.11.2 on a Windows 11 machine with an Intel Core i5-1135G7 @ 2.40GHz and 16 GB of RAM.

Visual Studio Code (VS Code) is a free and open-source code editor developed by Microsoft. It provides a modern and feature-rich environment for writing, debugging, and testing code. VS Code is available on Windows, macOS, and Linux, and supports a wide range of programming languages and frameworks [5].

HC-128 Algorithm Steps

1. Key Setup: The approach uses a key expansion algorithm to expand a key and an initialization vector (IV) into two arrays, P and Q.
2. Encryption Key Setup: By iterating over the P and Q arrays and updating their values each time to produce a fresh key stream word, the P and Q arrays are utilized to create a keystream for encryption.
3. Decryption Key Setup: In a manner similar to how the encryption key setup is carried out, the identical P and Q arrays are utilized to build a keystream for decryption.
4. Encryption: The algorithm creates a keystream with the encryption key configuration and XORs it with the plaintext to create the ciphertext to encrypt plaintext.
5. Decryption: The algorithm creates a keystream using the decryption key configuration and XORs it with the ciphertext to produce the plaintext in order to decrypt ciphertext.

SALSA20/12 Algorithm Steps

1. Key Setup: The algorithm uses a key expansion algorithm to expand a key and an initialization vector (IV) into a 16-word block.
2. Encryption: The algorithm creates a keystream by iterating over the 16-word block and uses it to modify a 64-byte block of data to encrypt plaintext. This is accomplished by utilizing a round function, which accepts a 16-word block as input, runs a number of operations on it, and then outputs a fresh 16-word block. To produce a keystream long enough for encryption, the round function is iterated several times.
3. Decryption: The algorithm creates a keystream similarly to encryption and XORs it with the ciphertext to produce the plaintext in order to decrypt ciphertext.

To create the key stream, the Salsa20/12 method combines addition, XOR, bit rotations, and modular arithmetic. The round function, which updates the values in the block with each iteration, is used to iterate over the 16-word block in order to create the key stream [6]. For the 256-bit version, the algorithm uses 12 rounds, while the 128-bit version uses a total of 20 rounds [7].

The *NaCl* library provides a number of cryptographic functions, including symmetric encryption, message authentication codes, public-key encryption, digital signatures, and key agreement. These functions are designed to be easy to use and secure, with simple and clear interfaces that minimize the risk of mistakes [8]. In addition to its cryptographic primitives, the NaCl library also includes a number of utility functions, such as random number generation, byte manipulation, and encoding and decoding functions [9].

B. Benchmarking tool

- psutil: The module provides an interface for retrieving information about processes running on the system, such as process IDs, CPU usage, memory usage, process status, and much more [10]. It also provides methods for retrieving information about system utilization, such as CPU usage, memory usage, disk usage, network statistics, and system uptime.
- time: The module is widely used in Python programs for time-related operations, such as timing how long a function takes to run, scheduling tasks to run at specific times, and generating timestamps for logging and debugging purposes.

C. Profiling tool

memory_profiler: The module will generate a report that shows the memory usage of your function, including the total memory usage, the peak memory usage, and the memory usage at various points in the function's execution [10].

D. Supporting libraries

- os: Provides a way to interact with the operating system [10]. It provides a wide range of functions that allow Python programs to access system-specific functionality, such as reading and writing files, managing processes, and interacting with the network.
- secrets: Provides a way to generate cryptographically secure random numbers and strings [10]. It is designed to be used for generating random tokens, passwords, and other

sensitive data that requires a high degree of randomness and security.

- **sys:** Provides access to some system-specific parameters and functions [10]. It is used to interact with the Python interpreter itself, and to access low-level functionality that is not provided by other Python libraries.

IV. RESULTS FOR PYTHON IMPLEMENTATION

This section provides the results based on the aforementioned performance metrics. The simulations are based on 5 different runs varying the size of plaintext. Details on the framework and tools used are provided in Section III.

A. Encryption Time

In the field of cryptography, the term "encryption time" describes how long it takes an encryption algorithm to convert a plaintext message into ciphertext. The length of the message, the complexity of the encryption algorithm, the size of the key used, and the encryption hardware all affect how long it takes to encrypt a plaintext. In order to encrypt plaintext messages, stream ciphers like HC-128 and Salsa20/12 create a keystream and XOR it with the plaintext. The keystream is produced using a nonce and a secret key (a one-time-use value) [11]. The amount of time needed to encrypt a plaintext using HC-128 and Salsa20/12 depends on the size of the key used and the length of the message.

The linear time complexity of HC-128 and Salsa20/12, where n is the message length, is O(n), according to theoretical calculations. This indicates that the length of a message has a linear effect on how long it takes to encrypt. The technology used for encryption and the algorithm's particular implementation can both affect how long it really takes to encrypt data. HC-128 and Salsa20/12 encryption can generally be performed at very fast speeds by modern CPUs; on modern technology, encryption rates of the order of terabytes per second are possible [11].

Below are the results segregated based on multiple runs of both the algorithms varying the message size:

TABLE I

ENCRYPTION TIME IN HC-128

Message Size	Encryption time
38 bytes	0.02031 seconds
45 bytes	0.01680 seconds
58 bytes	0.01926 seconds
67 bytes	0.01618 seconds
72 bytes	0.02067 seconds

TABLE II

ENCRYPTION TIME IN SALSA20/12

Message Size	Encryption time
38 bytes	0.00509 seconds
45 bytes	0.00874 seconds
58 bytes	0.00827 seconds
67 bytes	0.00924 seconds
72 bytes	0.01566 seconds

B. Decryption Time

In the field of cryptography, the term "decryption time" describes how long it takes a decryption method to convert a message from plaintext back into ciphertext. The complexity of the decryption technique, the size of the key used, the length of the ciphertext, and the decryption hardware all affect how quickly a ciphertext may be decrypted [12]. The fact that HC-128 and Salsa20/12 are both stream ciphers—which employ the same algorithm and secret key to produce a keystream that is XORed with the ciphertext to yield the original plaintext—means that they both use them to encrypt data. Since the same keystream is used for both encryption and decryption, stream ciphers often have a decryption time that is equal to their encryption time. The linear time complexity of HC-128 with Salsa20/12, where n is the length of the ciphertext, is O(n), according to theoretical calculations. This indicates that the length of a ciphertext linearly affects how long it takes to decipher.

Similar to encryption, the technology utilized for decryption as well as the precise way the algorithm is implemented might affect the actual decryption time. In general, modern Processors are capable of decrypting HC-128 and Salsa20/12 at very high rates; with current technology, decryption rates on the order of terabytes per second are possible.

Below are the results segregated based on multiple runs of both the algorithms varying the ciphertext size:

TABLE III

DECRYPTION TIME IN HC-128

Ciphertext Size	Encryption time
38 bytes	0.01409 seconds
45 bytes	0.01800 seconds
58 bytes	0.01402 seconds
67 bytes	0.01721 seconds
72 bytes	0.01309 seconds

TABLE III

DECRYPTION TIME IN SALSA20/12

Ciphertext Size	Encryption time
38 bytes	0.00601 seconds
45 bytes	0.00414 seconds
58 bytes	0.00724 seconds
67 bytes	0.00701 seconds
72 bytes	0.00601 seconds

C. CPU Usage

To conduct cryptographic operations like encryption, decryption, hashing, and key generation, a computer's central processing unit (CPU) needs a certain amount of computing power, or "CPU use," to do so [13]. While performing cryptographic operations, complicated mathematical calculations are involved, necessitating a significant amount of computer power, especially when using huge data sets and powerful cryptographic algorithms with large key sizes. As a

result, cryptographic processes may require a large amount of CPU time, which may have an impact on the system's or computer's general performance.

Several variables, including the complexity of the cryptographic method, the size of the key used, the volume of the input data, and the effectiveness of the cryptographic implementation, might have an impact on CPU consumption [13]. In general, cryptographic procedures demand greater CPU time when using more complicated algorithms and bigger key sizes.

Encryption Usage

The amount of processing power needed by a CPU to carry out encryption operations using a specific cryptographic method is referred to as CPU encryption utilization in the field of cryptography. Theoretically, the amount of CPU required to encrypt data using HC-128 and Salsa20/12 will vary depending on the hardware configuration, message length, key size, and algorithm implementation. Salsa20/12 and HC-128 are generally regarded as effective stream ciphers that can be implemented on current Processors with great performance. The message length and key size play a significant role in determining the theoretical CPU use for encrypting data using HC-128 and Salsa20/12. The linear time complexity of HC-128 and Salsa20/12 is O(n), where n is the message length. This implies that the amount of CPU time required to encrypt a message grows linearly with message length. The key size also has an impact on CPU consumption, with bigger key sizes necessitating more CPU time to complete encryption operations [14].

In practice, the actual CPU usage for encrypting data using HC-128 and Salsa20/12 will depend on the specific implementation of the algorithm, as well as the hardware and software environment used. However, on modern CPUs, the CPU usage for these algorithms is generally low, and encryption rates on the order of gigabytes per second can be achieved [14].

Below are the results segregated based on multiple runs of both the algorithms varying the message size:

TABLE IV

CPU ENCRYPTION USAGE IN HC-128

Message Size	Usage
38 bytes	14.3 %
45 bytes	11.8 %
58 bytes	7.6 %
67 bytes	2.69 %
72 bytes	9.4 %

TABLE V

CPU ENCRYPTION USAGE IN SALSA20/12

Message Size	Usage
38 bytes	12.5 %
45 bytes	15.4%
58 bytes	12.5 %
67 bytes	12.5 %
72 bytes	25 %

Decryption Usage

The size of the encrypted data, the CPU clock speed, the number of cores, and the method implementation all affect the theoretical CPU consumption for decrypting data using HC-128 and Salsa20/12. Nonetheless, in comparison to more contemporary encryption algorithms like AES, both HC-128 and Salsa20/12 have very low computational complexity.

The HC-128 stream cipher creates a keystream that is XORed with the plaintext to create the ciphertext using a 128-bit key and a 128-bit initialization vector (IV). The algorithm's very straightforward structure and usage of a 32-bit word-based design make it effective for software implementation. Because of this, utilizing HC-128 to decrypt data only uses a small amount of CPU power, which is mostly determined by the quantity of the data and the CPU clock speed [14]. Salsa20/12 is another stream cipher that creates a keystream using a 256-bit key and a 64-bit IV. The algorithm is more complicated than HC-128 because it is based on a 32-bit word-based design and has 20 rounds (or 12 rounds for Salsa20/12).

Salsa20/12 is still regarded as a rather effective algorithm for software implementation, nevertheless. Salsa20/12 uses a minimal amount of CPU power to decrypt data; this CPU use is primarily determined by the size of the data and the CPU clock speed. In conclusion, HC-128 and Salsa20/12 use relatively little CPU power compared to other contemporary encryption methods while decrypting data [14]. Nevertheless, the precise CPU consumption is dependent on a number of variables and can only be assessed by benchmarking on a particular hardware platform.

Below are the results segregated based on multiple runs of both the algorithms varying the message size:

TABLE VI

CPU DECRYPTION USAGE IN HC-128

Message Size	Usage
38 bytes	14.3 %
45 bytes	22.2 %
58 bytes	50 %
67 bytes	12.5 %
72 bytes	16.7 %

TABLE VII

CPU DECRYPTION USAGE IN SALSA20/12

Message Size	Usage
38 bytes	22.2 %
45 bytes	20 %
58 bytes	17.5 %
67 bytes	24 %
72 bytes	12.5 %

D. Memory Usage

In the context of cryptography, memory utilization refers to the amount of RAM needed to carry out cryptographic

operations like encryption and decryption. A cryptographic algorithm's memory requirements vary depending on its architecture, key size, block size, and mode of operation. Most of the time, symmetric key methods only need a minimal amount of memory [15]. Yet, as key and block sizes grow, so does the memory requirement.

In comparison to symmetric key algorithms, asymmetric key algorithms typically use more memory. A cryptographic algorithm's mode of operation can also have an impact on how much memory is required. For instance, the initialization vector (IV) and the prior ciphertext block must be stored in additional memory when using the cipher block chaining (CBC) method. The implementation of the algorithm can have an impact on the memory utilization in cryptography in addition to the design and mode of operation of the method. Due to the cost involved in running the algorithm on a general-purpose computer, a software implementation, for instance, might need more RAM than a hardware implementation [15].

Generally, reducing memory utilization is crucial for cryptographic algorithm authors, especially in situations with limited resources like embedded systems and mobile devices. Cryptographic algorithms' memory needs must be carefully considered by designers in order to balance security and speed.

Encryption Usage

The size of the message and how the algorithm is implemented determine how much memory is required to encrypt data using the HC-128 and Salsa20/12 algorithms. Due to the fact that HC-128 and Salsa20/12 are both stream ciphers, they encrypt data on a per-byte basis using a pseudorandom keystream created from a secret key and a nonce. As a result, the message size generally affects how much RAM is required to encrypt data using these ciphers [16].

The platform being utilized and how the method is implemented specifically will both affect how much memory is required. Stream ciphers, on the other hand, generally use less memory than block ciphers because they don't need to buffer a lot of data. Even when encrypting huge volumes of data, the memory utilization for HC-128 and Salsa20/12 encryption is often relatively low and shouldn't be a major concern for most applications.

Below are the results segregated based on multiple runs of both the algorithms varying the message size:

TABLE VIII

MEMORY ENCRYPTION USAGE IN HC-128

Message Size	Usage
38 bytes	1110016 bytes
45 bytes	1204224 bytes
58 bytes	626688 bytes
67 bytes	241664 bytes
72 bytes	286720 bytes

TABLE IX

MEMORY ENCRYPTION USAGE IN SALSA20/12

Message Size	Usage
38 bytes	36864 bytes
45 bytes	196608 bytes
58 bytes	286720 bytes
67 bytes	49152 bytes
72 bytes	393216 bytes

Decryption Usage

The amount of memory required to decrypt data using HC-128 and Salsa20/12 is comparable to the amount of memory required to encrypt data, and it also depends on the size of the message and how the technique is implemented. Both HC-128 and Salsa20/12 are stream ciphers that decrypt data per-byte using a pseudorandom keystream created from a secret key and a nonce, just like with encryption [16]. As a result, the message size generally affects how much RAM is required to decrypt data using these ciphers.

The necessity to temporarily keep the decrypted plaintext in memory before using it has an impact on how much memory is required during decryption. This is especially true for programs that demand that a plaintext message's whole contents be kept in memory at once. The memory usage for decrypting data with HC-128 and Salsa20/12 is typically minimal in reality and shouldn't be a major concern for the majority of applications. The exact amount of memory utilized, like with encryption, will depend on how the method is implemented and the platform being used.

Below are the results segregated based on multiple runs of both the algorithms varying the message size:

TABLE X

MEMORY DECRYPTION USAGE IN HC-128

Message Size	Usage
38 bytes	24576 bytes
45 bytes	20480 bytes
58 bytes	282624 bytes
67 bytes	524288 bytes
72 bytes	843776 bytes

TABLE XI

MEMORY DECRYPTION USAGE IN SALSA20/12

Message Size	Usage
38 bytes	69632 bytes
45 bytes	315392 bytes
58 bytes	3362816 bytes
67 bytes	65536 bytes
72 bytes	262144 bytes

E. Code Size

The quantity of programming code needed to implement a cryptographic technique is referred to as the "code size" in cryptography. It is frequently quantified in terms of the amount of code lines or the size of the binary code [16]. Both the stream ciphers Salsa20/12 and HC-128 are comparably easy to use and have compact code sizes when compared to other cipher types.

It is challenging to give a precise estimate for the theoretical value of code size since it might vary based on the programming language used, the intricacies of the implementation, and the platform on which the code is operating. However in contrast to many other encryption algorithms, HC-128 and Salsa20/12 both aim to reduce code size and have relatively tiny code footprints. Depending on the specific implementation and the needed features and functionality, the code size for HC-128 and Salsa20/12 implementations might vary from a few hundred to a few thousand lines of code [17]. Performance can also be enhanced and code size can be further decreased by using optimised and platform-specific code.

Below are the results for both the algorithm code sizes:

TABLE XI

CODE SIZE FOR HC-128 AND SALSA20/12

Algorithm	Size
HC-128	4456 bytes
SALSA20/12	2015 bytes

V. CONCLUSION AND FUTURE WORK

Through trial and error and self-knowledge, the codes for both of these algorithms—which were hand-selected from the official GitHub repository for eSTREAM ciphers—were converted from C to Python. Using the analysis, the following conclusions can be made:

- Salsa20/12 is marginally quicker in some cases, according to the performance investigation of HC-128 and Salsa20/12, however both ciphers have similar encryption and decryption throughput.
- Salsa20/12 uses less memory than HC-128, hence it is less suited to situations with limited resources.
- Salsa20/12 has a smaller code size than HC-128, which would make it less attractive in circumstances where code size is an important consideration.
- Both ciphers perform effectively in terms of CPU utilization, with Salsa20/12 occasionally being a little quicker.
- The HC-128 may be a better option if memory utilization and code size are not crucial considerations and slightly quicker encryption and decryption times are desired.
- Generally, the decision between Salsa20/12 and HC-128 depends on the particular use case and the implementer's priorities. It also relies on whether any background processes are active or not.

It is important to use an algorithm that can encrypt data securely as well. A major theme and issue throughout the entire area of cryptography has been security. Similar to this, there are various other factors and situations that should be taken into account when examining these algorithms. As a result, research to improve security issues and identify better algorithms may be done in the future.

REFERENCES

[1] Carlos Cid (RHUL) and Matt Robshaw (FTRD), The eSTREAM Portfolio in 2012.

[2] M. Robshaw and O. Billet, editors. New Stream Cipher Designs: The eSTREAM Final-ists. LNCS 4986, pp. 267293. Springer 2008.

[3] ECRYPT Network of Excellence. The eSTREAM project, available via http://www.ecrypt.eu.org/stream/.

[4] "Efficient implementation of eSTREAM ciphers on 8-bit AVR microcontrollers | IEEE Conference Publication | IEEE Xplore." https://ieeexplore.ieee.org/abstract/document/4577681

[5] "Visual Studio Code - Code Editing. Redefined." https://code.visualstudio.com/.

[6] "libestream/hc-128.c at master · lvella/libestreamGitHub." https://github.com/lvella/libestream/blob/master/hc-128.c

[7] L. C. Vella, "lvella/libestream." Jun. 01, 2021. Available: https://github.com/lvella/libestream/blob/5859bf199cbba23ff1528327aaaa970f301e986b/salsa20.c

[8] T. P. developers, "PyNaCl: Python binding to the Networking and Cryptography (NaCl) library." Available: https://github.com/pyca/pynacl/

[9] "PyNaCl: Python binding to the libsodium library." Python Cryptographic Authority, Apr. 09, 2023. [Online]. Available: https://github.com/pyca/pynacl

[10] "3.11.3 Documentation." https://docs.python.org/3/

[11] E. Biham, L.R. Knudsen, and R.J. Anderson. Serpent: A New Block Cipher Proposal. In S. Vaudenay, editors, Proceedings of FSE 1998, LNCS, volume 1372, pp. 222- 238, Springer Verlag.

[12] Hongjun Wu, "The Stream Cipher HC-128", Katholieke Universiteit Leuven, ESAT/SCD-COSIC Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgiumwu.hongjun@esat.kuleuven.be

[13] Daniel J. Bernstein, Department of Mathematics, Statistics, and Computer Science(M/C 249), "Salsa20 speci_cation", The University of Illinois at Chicago, Chicago, IL 60607-7045, snuffle@box.cr.yp.to

[14] "HC-128 Stream Cipher", A-Team Deliverables (Michael Burns, Brian Baum)

[15] H.C.A.V. Tilborg, Fundamentals of Cryptology, Kluwer Academic Publisher Boston , 1988.

[16] D. B. Budhathoki, "Performance Analysis of eSTREAM Cipher Finalists: HC-128, Salsa20/12, Rabbit & SOSEMANUK," Thesis, Department of Computer Science and Information Technology, 2016. Available: https://elibrary.tucl.edu.np/handle/123456789/9759

[17] F. M. Eljadi and I. F. Al-Shaikhli, "Statistical Analysis of the eSTREAM Competition Winners," in 2015 4th International Conference on Advanced Computer Science Applications and Technologies (ACSAT), Dec. 2015, pp. 80–85. doi: 10.1109/ACSAT.2015.43.