

**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

*"Jnana Sangama", Belgaum - 590014, Karnataka*



**CRYPTOGRAPHY  
AAT REPORT**

On  
**ChaCha20**

Submitted by

Sarthaka Mitra GB (1BM23CS305)  
Shaikh Uzair Ahmed (1BM23CS307)  
Shashank U (1BM23CS314)  
Subramanya J (1BM23CS343)

Under the Guidance of

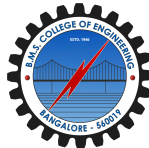
Dr. Nandhini Vineeth  
Associate Professor  
Department of CSE  
B. M. S. College Of Engineering

in partial fulfillment for the award of the degree of

**BACHELOR OF ENGINEERING**

in

**COMPUTER SCIENCE AND ENGINEERING**



**B. M. S. College Of Engineering**  
(Autonomous Institution under VTU)

**BENGALURU - 560019**

**February 2025 to June 2025**

**B. M. S. College Of Engineering,**

Bull Temple Road, Bengaluru 560019

*(Affiliated To Visvesvaraya Technological University, Belgaum)*

**Department of Computer Science and Engineering**



## **CERTIFICATE**

This is to certify that the project work entitled **ChaCha20**, carried out by **Sarthaka Mitra GB (1BM23CS305)**, **Shaikh Uzair Ahmed (1BM23CS307)**, **Shashank U (1BM23CS314)**, **Subramanya J (1BM23CS343)**, who are bonafide students of **B. M. S. College of Engineering**, is in partial fulfillment for the award of Bachelor of Engineering in Computer Science and Engineering of Visvesvaraya Technological University, Belgaum during the year 2024-2025. The AAT has been approved as it satisfies the academic requirements of the course CRYPTOGRAPHY (23CS4ESCRP) work prescribed for the said degree.

Signature of the Guide

**Dr. Nandhini Vineeth**

Associate Professor

Department of CSE

B. M. S. College Of Engineering

Signature of the HOD

**Dr. Kavita Sooda**

Professor and Head

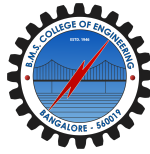
Department of CSE

B. M. S. College Of Engineering

**Name of the Examiner**

**Signature with Date**

**B. M. S. College Of Engineering,  
Department of Computer Science and Engineering**



## Declaration

We, **Sarthaka Mitra GB (1BM23CS305)**, **Shaikh Uzair Ahmed (1BM23CS307)**, **Shashank U (1BM23CS314)**, **Subramanya J (1BM23CS343)**, students of 4th Semester, B.E, Department of Computer Science and Engineering, **BMS College of Engineering, Bengaluru**, hereby declare that this AAT entitled **ChaCha20** has been carried out by us under the guidance of **Dr. Nandhini Vineeth**, Associate Professor, Department of CSE, B. M. S. College Of Engineering, Bangalore, during the academic semester **April 2025 - August 2025**.

We also declare that, to the best of our knowledge and belief, the development reported here is not a part of any other report by any other students.

**Signatures**

**Sarthaka Mitra GB (1BM23CS305)**

**Shaikh Uzair Ahmed (1BM23CS307)**

**Shashank U (1BM23CS314)**

**Subramanya J (1BM23CS343)**

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Cryptanalysis . . . . .	6
1.1.1	Boolean Encoding . . . . .	6
1.1.2	Physical Side - Channel attacks . . . . .	7
<b>2</b>	<b>Methodology</b>	<b>8</b>
2.1	Key and Nonce Generation . . . . .	8
2.2	Initialization . . . . .	9
2.3	Block Processing . . . . .	9
2.4	Message Authentication . . . . .	11
2.5	Decryption Process . . . . .	11
<b>3</b>	<b>Results and Discussion</b>	<b>13</b>
3.1	Compilation . . . . .	13
3.2	Encryption . . . . .	14
3.3	Decryption . . . . .	14
3.4	Validation . . . . .	15
<b>4</b>	<b>Conclusion and Future Work</b>	<b>16</b>

# List of Figures

3.1	Automated compilation using GNU Make . . . . .	13
3.2	Usage help message and encryption . . . . .	14
3.3	Decryption with authenticity validation . . . . .	14
3.4	Invalid MAC signature detection and invalidation of data . . . . .	15

# List of Tables

# Chapter 1

## Introduction

ChaCha20 is a stream cipher designed by Daniel J. Bernstein in 2008 as a variant of the Salsa20 cipher [1]. The primary motivation behind the development of ChaCha20 was to create a cryptographic algorithm that is both secure and efficient, particularly in software implementations. Bernstein aimed to address some of the limitations and vulnerabilities found in existing ciphers, particularly in the context of high-performance applications.

The development of ChaCha20 was primarily motivated by the need for a cryptographic algorithm that could achieve high security while maintaining efficiency, particularly in software implementations [2]. Prior to ChaCha20, many cryptographic algorithms, such as RC4 and even Salsa20, were often optimized for hardware execution. This hardware-centric design approach leveraged the parallel processing capabilities of dedicated cryptographic hardware, which could perform operations at high speeds.

Software implementations of cryptographic algorithms faced several challenges. Many existing algorithms were not designed with the constraints of software execution in mind, leading to inefficiencies such as high computational overhead and increased memory usage. For instance, algorithms that relied heavily on complex mathematical operations or extensive key schedules could result in slower performance on general-purpose processors. Additionally, software implementations were more susceptible to side-channel attacks, such as timing attacks, where an attacker could infer information based on the time taken to execute cryptographic operations.

ChaCha20 was specifically designed to address these shortcomings. It employs a simple and efficient structure that minimizes the number of operations required to generate the keystream, making it suitable for software execution. The algorithm's design allows it to perform well even on devices with limited processing power, such as smartphones and IoT devices. Furthermore, ChaCha20's key schedule is designed to be resistant to related-key attacks, enhancing its security profile.

By focusing on both security and efficiency, ChaCha20 has become a widely adopted algorithm in modern cryptographic protocols, including TLS and SSH, ensuring robust encryption in a software-driven environment. Its ability to deliver high performance without compromising security has made it a preferred choice for developers seeking to implement secure communication in a variety of applications.

Before the introduction of ChaCha20, several stream ciphers were widely used, including RC4 variants [3] and Salsa20. RC4, designed by Ron Rivest in 1987, was popular due to its simplicity and speed. However, it was later found to have significant vulnerabilities, such as biases in the keystream that could be exploited in certain contexts, leading to potential security breaches [4]. RC4's vulnerabilities

were primarily due to inherent biases in its keystream, resulting in a non-uniform distribution of output bytes [5]. This meant that certain bytes were more likely to appear than others, creating predictable patterns. Attackers could exploit these biases, especially in scenarios where the same key was reused, such as in HTTPS sessions. For instance, in the "Fluhrer, Mantin, and Shamir" attack, attackers could recover portions of plaintext by analyzing the keystream. This predictability undermined the confidentiality of encrypted data, leading to significant security breaches and prompting the cryptographic community to recommend against the use of RC4 in favor of more secure alternatives.

Salsa20, developed by Bernstein himself, was an improvement over RC4, offering better security and performance [6]. However, it still faced scrutiny regarding its resistance to certain cryptanalytic attacks [7]. Specifically, concerns arose about its key schedule and the potential for related-key attacks. Salsa20, designed by Bernstein as an enhancement over RC4, provided improved security and performance through its innovative design and efficient operations. However, it was not without vulnerabilities. Cryptanalysts raised concerns about its key schedule, which could potentially allow attackers to exploit related-key attacks, where knowledge of one key could lead to the compromise of another. Additionally, while Salsa20 demonstrated resilience against many known attacks, the possibility of future cryptanalytic advancements necessitated ongoing scrutiny and evaluation of its security in various contexts.

ChaCha20 was created to enhance the security and performance of Salsa20 while addressing these weaknesses. It features a more robust key schedule and improved diffusion properties, making it resistant to known attacks and suitable for high-performance applications, particularly in software environments where efficiency is crucial.

ChaCha20 operates on 64-byte blocks and utilizes a 256-bit key along with a 12-byte nonce (number used once). The algorithm employs a series of operations, including addition, bitwise XOR, and rotation, to produce a keystream that is combined with plaintext to generate ciphertext. The design of ChaCha20 emphasizes simplicity and speed, making it suitable for a wide range of platforms, from low-power devices to high-performance servers.

The security of ChaCha20 has been extensively analyzed, and it has been adopted in various cryptographic protocols, including TLS (Transport Layer Security) and SSH (Secure Shell). Its resilience against known cryptanalytic attacks and its performance characteristics have made it a preferred choice for modern encryption needs.

**Why This is Relevant Today** In an era where data breaches and cyberattacks are increasingly common, the need for robust encryption solutions has never been more critical. ChaCha20's relevance today stems from several factors:

**Increased Data Sensitivity:** With the proliferation of digital data, including personal, financial, and health information, there is a heightened need to protect sensitive data from unauthorized access. **Performance Requirements:** As applications become more demanding, the need for fast and efficient encryption algorithms is paramount. ChaCha20's design allows it to perform well on a variety of hardware, including mobile devices and embedded systems, where computational resources may be limited. **Regulatory Compliance:** Organizations are often required to comply with data protection regulations, such as GDPR (General Data Protection Regulation) and HIPAA (Health Insurance Portability and Accountability Act). Implementing strong encryption methods like ChaCha20 helps organizations meet these legal requirements and protect user privacy.



The implementation of the ChaCha20 file encryption and decryption utility specifically addresses several weaknesses associated with traditional encryption methods:

**Nonce Reuse:** Many encryption algorithms are vulnerable to attacks if the same nonce is reused for multiple encryption operations. ChaCha20 mitigates this risk by generating a unique nonce for each encryption session, ensuring that even identical plaintexts yield different ciphertexts. **Key Management:** The utility emphasizes secure key management by deriving encryption keys from user-provided passwords, which can be further enhanced with key stretching techniques. This approach helps protect against brute-force attacks on weak passwords. **Data Integrity:** The utility incorporates the Poly1305 message authentication code (MAC) to ensure the integrity and authenticity of the encrypted data [8]. This addresses the risk of data tampering and provides assurance that the data has not been altered during storage or transmission [9]. By leveraging the strengths of ChaCha20 and addressing these critical weaknesses, the utility provides a secure and efficient solution for file encryption and decryption in today's digital landscape.

## 1.1 Cryptanalysis

### 1.1.1 Boolean Encoding

Boolean encoding is a prominent method for analyzing various algorithms and computer systems, including the cryptanalysis of symmetric ciphers.

Boolean encoding is a method of representing a complex problem (like a cipher algorithm) as a Boolean propositional formula.

A Boolean propositional formula is an equation or set of equations composed of Boolean variables (true/false or 0/1) combined using logical operators (AND, OR, NOT).

This encoding is essential because it allows the problem to be processed by SAT solvers, which are specialized programs that solve Boolean formulas.

The focus of Sobon and Stachowiak's research was on using Boolean encoding to analyze ChaCha20 [10]. The process begins by encoding the entire ChaCha20 algorithm as a Boolean propositional formula, transforming all its mathematical operations and logical transformations into Boolean logic. Randomly selected bits of plaintext and the cryptographic key are also encoded in Boolean form.

Using this encoded model, the corresponding ciphertext is calculated through SAT solvers, specialized programs capable of solving Boolean formulas. The core of the cryptanalysis involves attempting to compute the secret key from the known plaintext and ciphertext using these SAT solvers. Multiple SAT solvers, including both older yet efficient versions and modern, popular ones, are tested for performance and effectiveness in this task. Additionally, the study explores ChaCha20's properties by analyzing its standard configuration and weaker variants, providing insights into how SAT techniques perform against different levels of cipher strength.

The process begins with:

**Encoding the ChaCha20 algorithm as a Boolean formula:** This means expressing all the mathematical operations and logical transformations of ChaCha20 in Boolean logic.

**Encoding plaintext and cryptographic keys:** Randomly chosen bits of plaintext (the message) and the key (the secret code) are also converted into Boolean formulas.

Calculating ciphertext: By using SAT solvers on these formulas, the corresponding ciphertext is calculated.

### 1.1.2 Physical Side - Channel attacks

Physical side-channel attacks are a class of cryptanalytic techniques that exploit unintended information leakage from cryptographic devices during their operation. Unlike traditional cryptanalysis, which focuses on breaking the mathematical structure of a cipher, side-channel attacks target the physical properties of a device running the cryptographic algorithm.

Common forms of side-channel leakage include timing information, power consumption, electromagnetic (EM) emissions, and acoustic signatures. For instance, timing attacks analyze the time taken by a device to perform cryptographic operations, while power analysis examines fluctuations in power usage to infer secret keys. EM side-channel attacks, a particularly sophisticated variant, involve capturing electromagnetic radiation emitted during cryptographic computations. This leakage can reveal sensitive information, such as key-dependent values manipulated within the cipher, making it a powerful tool for attackers with physical proximity to the target device.

Modern cryptographic algorithms like ChaCha20, which rely on addition, rotation, and XOR (ARX) operations, are considered secure against timing attacks due to their constant-time design. However, they remain vulnerable to EM side-channel attacks because these attacks bypass algorithmic defenses and directly exploit physical characteristics. This makes understanding and mitigating side-channel vulnerabilities essential for the secure deployment of cryptographic systems in real-world environments.

In their paper, Jungk and Bhasin [11] investigated EM-based side-channel attacks on ChaCha20. Two attacks were presented, one assuming a strong attacker with full control over the nonce while the other assumes a weak attacker having only partial control. The proposed attack is also applicable on Poly1305 based key generation which itself uses ChaCha20. Finally the potential of using a shuffling countermeasure to mitigate side-channel leakage is explored. Although shuffling can significantly reduce the SNR of the leakage, the countermeasure still remains vulnerable and can be potentially exploited with added effort. Future works could explore alternative lightweight countermeasures for ARX constructions.

# Chapter 2

## Methodology

Steps in the Complete ChaCha20 Implementation :

### 2.1 Key and Nonce Generation

1. Key derivation : Transforms a user-provided password into a secure cryptographic key suitable for ChaCha20, which requires a 256-bit key. This transformation is achieved using a Key Derivation Function (KDF), which ensures that the derived key is both strong and resistant to attacks.

A KDF takes the input password and processes it with additional parameters, such as a salt and an iteration count. The salt is a random value added to the password before hashing, ensuring that identical passwords yield different keys. This prevents attackers from using pre-computed tables, like rainbow tables, to crack passwords efficiently.

The iteration count specifies how many times the KDF applies its hashing algorithm, increasing the computational effort required to derive the key. This makes brute-force attacks more challenging, as attackers must perform the KDF for each password guess multiple times.

Common KDFs include PBKDF2, bcrypt, and Argon2, each designed to enhance security through these mechanisms. By deriving a secure key from a password, the ChaCha20 encryption process can effectively protect sensitive data against unauthorized access. Proper key management practices are essential to maintain the confidentiality and integrity of the derived key throughout its lifecycle.

2. Nonce generation : A nonce, or "number used once," is a unique value that is generated for each encryption operation. In the case of ChaCha20, the nonce is 12 bytes (96 bits) in length, providing a sufficiently large space to ensure uniqueness across multiple encryption sessions.

The primary purpose of the nonce is to ensure that the same plaintext encrypted multiple times with the same key produces different ciphertexts. This is crucial for maintaining the security of the encryption scheme. If a nonce is reused with the same key, it can lead to vulnerabilities, such as revealing patterns in the ciphertext and potentially allowing attackers to recover the plaintext through various cryptanalytic techniques.

To generate a secure nonce, it is essential to use a reliable source of randomness. Cryptographic random number generators (CSPRNGs) are typically employed

to produce nonces that are unpredictable and uniformly distributed. This randomness is critical, as predictable nonces can compromise the security of the encryption.

In practice, the nonce is often stored alongside the ciphertext, allowing the decryption process to retrieve it easily. By ensuring that each nonce is unique and never reused with the same key, the integrity of the ChaCha20 encryption process is upheld, significantly enhancing the overall security of the encrypted data. Proper nonce management is essential to prevent accidental reuse, which could lead to severe security vulnerabilities. This nonce should never be reused with the same key to maintain security.

## 2.2 Initialization

1. The ChaCha20 state consists of 16 32-bit words, organized into a specific structure that combines fixed constants, the encryption key, the nonce, and a counter.
  - (a) Fixed Constants: The first four words of the state are fixed constants that serve as identifiers for the ChaCha20 algorithm. These constants are predefined values that help distinguish the ChaCha20 cipher from other algorithms, ensuring that the initialization process is consistent and recognizable.
  - (b) Key Integration: The next eight words of the state are derived directly from the 256-bit key. Each 32-bit segment of the key is loaded into the state, providing the necessary cryptographic strength for the encryption process. This integration ensures that the key influences the keystream generation, making it unique to the specific encryption session.
  - (c) Nonce and Counter: The final four words of the state are derived from the nonce and a counter. The nonce, which is unique for each encryption operation, is split into three 32-bit segments, while the counter, initialized to zero, is included as the last word. The counter is incremented with each block processed, ensuring that the state changes with each encryption operation and preventing the reuse of the same keystream. By combining these elements, the ChaCha20 state is initialized in a way that maximizes security and efficiency. This structured approach allows the algorithm to produce a high-quality keystream that is resistant to cryptanalytic attacks. The proper setup of the state is essential for the subsequent keystream generation, which directly impacts the confidentiality of the encrypted data.

## 2.3 Block Processing

1. Here the initialized state undergoes a series of transformations to produce a pseudorandom keystream. This keystream is then used to encrypt plaintext by performing an XOR operation with the data.
  - (a) Quarter-Round Operations: The core of the keystream generation involves a series of quarter-round operations, which are designed to mix

the state variables effectively. Each quarter-round operation takes four 32-bit words from the state and applies a sequence of arithmetic and bitwise operations, including addition, XOR, and rotation. This mixing process ensures that the output is highly dependent on all input values, enhancing the diffusion properties of the algorithm.

- (b) Loop Iterations: The ChaCha20 algorithm performs a total of 20 rounds of these quarter-round operations, divided into 10 iterations. In each iteration, the state is processed in a specific order, with the quarter-round operations applied to different combinations of state variables. This repeated mixing significantly increases the complexity of the output, making it resistant to cryptanalysis.
  - (c) Keystream Output: After completing the quarter-round operations, the final state is added to the original state used for initialization. This addition ensures that the keystream is unique to the specific key, nonce, and counter combination. The result is a 64-byte keystream, which is then used in the encryption process. The keystream is generated on-the-fly, allowing for efficient encryption of data in blocks, without the need to precompute large amounts of data. This is done in a loop to produce a keystream of 64 bytes.
2. The XOR operation is a crucial step in the ChaCha20 encryption process, where the generated keystream is combined with the plaintext data to produce the final ciphertext. This operation leverages the properties of the XOR (exclusive OR) function, which is fundamental to the security of stream ciphers.
- (a) Combining Keystream and Plaintext: During encryption, the 64-byte keystream generated by the ChaCha20 algorithm is XORed with the plaintext data in 64-byte blocks. The XOR operation is performed on each corresponding byte of the keystream and plaintext, resulting in a ciphertext that is a direct transformation of the original data. Mathematically, this can be expressed as:  $\text{ciphertext}[i] = \text{plaintext}[i] \oplus \text{keystream}[i]$  where  $\oplus$  denotes the XOR operation.
  - (b) Security Through Randomness: One of the key advantages of using the XOR operation in this context is that it ensures that the same plaintext encrypted multiple times with the same key and nonce will yield different ciphertexts. This is due to the unique nature of the keystream generated for each encryption session. As a result, even if the same plaintext is processed, the output ciphertext will differ, thereby enhancing security and preventing attackers from identifying patterns in the encrypted data.
  - (c) Reversibility for Decryption: The XOR operation is inherently reversible, which is a critical property for decryption. To recover the original plaintext, the same keystream used for encryption is XORed with the ciphertext:  $\text{plaintext}[i] = \text{ciphertext}[i] \oplus \text{keystream}[i]$  This property allows for efficient and straightforward decryption, making the ChaCha20 algorithm both secure and practical for real-world applications.

## 2.4 Message Authentication

This provides a mechanism to ensure the integrity and authenticity of the encrypted data. This step is required for verifying that the ciphertext has not been altered and that it originates from a legitimate source.

1. **MAC Generation:** After the encryption process is complete, the Poly1305 algorithm is employed to generate a MAC. This MAC is computed over the ciphertext and any associated data (AAD) that may be included in the encryption operation. The use of AAD allows for additional context to be authenticated without being encrypted, which is useful in various applications where metadata needs to be verified.
2. **Key Usage:** The same key used for the ChaCha20 encryption is also utilized in the Poly1305 MAC computation.

$$\text{MAC} = \left( \sum_{i=0}^{n-1} m_i \times r^i \right) \mod (2^{130} - 5)$$

- (a)  $m_i$  are blocks of the message, treated as 128-bit integers.
- (b)  $r$  is a 128-bit value derived from the Poly1305 key, but it is not directly used as a secret.
- (c)  $n$  is the number of blocks in the message.
- (d) The modulo  $2^{130} - 5$  is a large prime number, ensuring security and preventing overflow.

This ensures that the MAC is tightly coupled with the encryption process, enhancing security. The MAC is derived from the ciphertext and the associated data, producing a fixed-length output (typically 16 bytes) that serves as a cryptographic signature for the data. **Integrity and Authenticity:** The primary purpose of the Poly1305 MAC is to provide assurance that the ciphertext has not been tampered with during transmission or storage. When the recipient receives the ciphertext, they can compute the MAC using the same key and compare it to the received MAC. If the two MACs match, it confirms that the data is intact and authentic. If they do not match, it indicates potential tampering or corruption, prompting the recipient to reject the data.

3. **Appending the MAC:** The computed MAC is appended to the end of the ciphertext, allowing the recipient to verify the authenticity of the message upon decryption.

## 2.5 Decryption Process

1. **Nonce Retrieval:** For decryption, the nonce is read from the beginning of the ciphertext.
2. **State Reinitialization:** The ChaCha20 state is reinitialized using the same key, nonce, and counter as during encryption.
3. **Keystream Generation:** The keystream is generated again using the same process as in encryption.

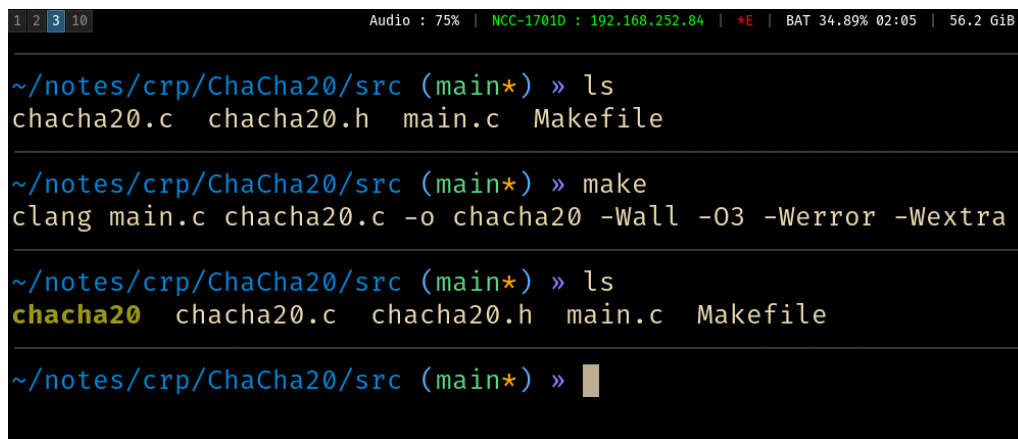
4. XOR Operation: The ciphertext is XORed with the keystream to recover the original plaintext.
5. MAC Verification: The MAC is extracted from the end of the ciphertext, and the Poly1305 MAC is computed over the decrypted plaintext and associated data. If the computed MAC does not match the extracted MAC, the decryption fails, indicating potential tampering.

# Chapter 3

## Results and Discussion

Images relevant to the operational features of the program have been attached below

### 3.1 Compilation

A terminal window with a dark background and light-colored text. The window title bar at the top shows system status: 'Audio : 75% | NCC-1701D : 192.168.252.84 | \*E | BAT 34.89% 02:05 | 56.2 GiB'. The terminal content shows a series of commands and their outputs in a directory '~/.notes/crp/ChaCha20/src (main\*)'. The commands are: 'ls', 'make', 'ls', and a prompt. The outputs are: 'chacha20.c chacha20.h main.c Makefile', 'clang main.c chacha20.c -o chacha20 -Wall -O3 -Werror -Wextra', and 'chacha20 chacha20.c chacha20.h main.c Makefile'. The final prompt is followed by a cursor.

```
1 2 3 10 Audio : 75% | NCC-1701D : 192.168.252.84 | *E | BAT 34.89% 02:05 | 56.2 GiB

~/.notes/crp/ChaCha20/src (main*) » ls
chacha20.c  chacha20.h  main.c  Makefile

~/.notes/crp/ChaCha20/src (main*) » make
clang main.c chacha20.c -o chacha20 -Wall -O3 -Werror -Wextra

~/.notes/crp/ChaCha20/src (main*) » ls
chacha20  chacha20.c  chacha20.h  main.c  Makefile

~/.notes/crp/ChaCha20/src (main*) » █
```

Figure 3.1: Automated compilation using GNU Make

A suitable Makefile has been used for speeding up recompilation time.

GNU Make is a build automation tool that reads Makefile instructions to compile code efficiently, tracking dependencies for minimal recompilation [12]. It uses rules of the form target: dependencies followed by commands to build the target. It supports variables, conditionals, and functions for flexible build configurations.

The clang compiler has been utilized, with the address sanitizer in development, and with level 3 optimization in release mode. [13]

AddressSanitizer (ASan) is a memory error detector for C/C++ programs, part of GCC and Clang. It detects out-of-bounds access, use-after-free, use-after-scope, and memory leaks. ASan adds runtime checks, making programs slightly slower and more memory-intensive. It provides clear error messages with stack traces for quick debugging [14].

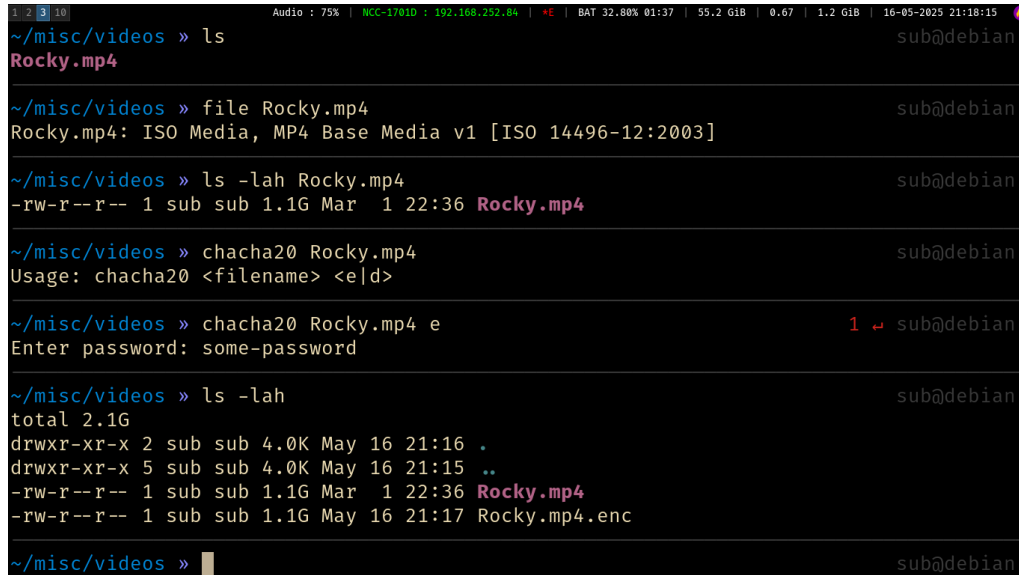
-O3 in GCC is designed for performance-intensive applications where speed is a priority. In addition to the optimizations at the -O2 level, -O3 includes optimizations like aggressive inlining, loop unrolling, and vectorization.

These optimizations can result in significant performance gains for programs that benefit from highly parallelized execution or intensive computational workloads. However, -O3 can sometimes lead to larger binary sizes, as more functions may be



inlined, and loops are unrolled to reduce loop control overhead. While this typically improves performance, the increased code size can potentially lead to cache misses and slower performance on certain systems, especially those with limited memory or cache.

## 3.2 Encryption



```

1 2 3 10 Audio : 75% | MCC-1701D : 192.168.252.84 | +E | BAT 32.80% 01:37 | 55.2 GiB | 0.67 | 1.2 GiB | 16-05-2025 21:18:15
~/misc/videos » ls sub@debian
Rocky.mp4

~/misc/videos » file Rocky.mp4 sub@debian
Rocky.mp4: ISO Media, MP4 Base Media v1 [ISO 14496-12:2003]

~/misc/videos » ls -lah Rocky.mp4 sub@debian
-rw-r--r-- 1 sub sub 1.1G Mar  1 22:36 Rocky.mp4

~/misc/videos » chacha20 Rocky.mp4 sub@debian
Usage: chacha20 <filename> <e|d>

~/misc/videos » chacha20 Rocky.mp4 e 1 ↵ sub@debian
Enter password: some-password

~/misc/videos » ls -lah sub@debian
total 2.1G
drwxr-xr-x 2 sub sub 4.0K May 16 21:16 .
drwxr-xr-x 5 sub sub 4.0K May 16 21:15 ..
-rw-r--r-- 1 sub sub 1.1G Mar  1 22:36 Rocky.mp4
-rw-r--r-- 1 sub sub 1.1G May 16 21:17 Rocky.mp4.enc

~/misc/videos »

```

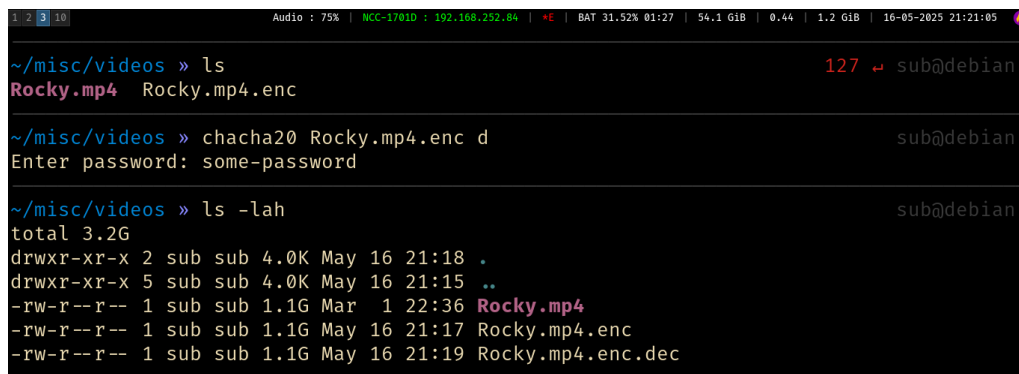
Figure 3.2: Usage help message and encryption

We demonstrate usage of the binary on a large file (1.1GiB).

This also demonstrates the capability of modern stream ciphers to act not only on ASCII text data, but data at the binary level encoded in a multitude of formats.

The program delivers an appropriate message when flags are not correctly provided.

## 3.3 Decryption



```

1 2 3 10 Audio : 75% | MCC-1701D : 192.168.252.84 | +E | BAT 31.52% 01:27 | 54.1 GiB | 0.44 | 1.2 GiB | 16-05-2025 21:21:05
~/misc/videos » ls 127 ↵ sub@debian
Rocky.mp4 Rocky.mp4.enc

~/misc/videos » chacha20 Rocky.mp4.enc d sub@debian
Enter password: some-password

~/misc/videos » ls -lah sub@debian
total 3.2G
drwxr-xr-x 2 sub sub 4.0K May 16 21:18 .
drwxr-xr-x 5 sub sub 4.0K May 16 21:15 ..
-rw-r--r-- 1 sub sub 1.1G Mar  1 22:36 Rocky.mp4
-rw-r--r-- 1 sub sub 1.1G May 16 21:17 Rocky.mp4.enc
-rw-r--r-- 1 sub sub 1.1G May 16 21:19 Rocky.mp4.enc.dec

~/misc/videos »

```

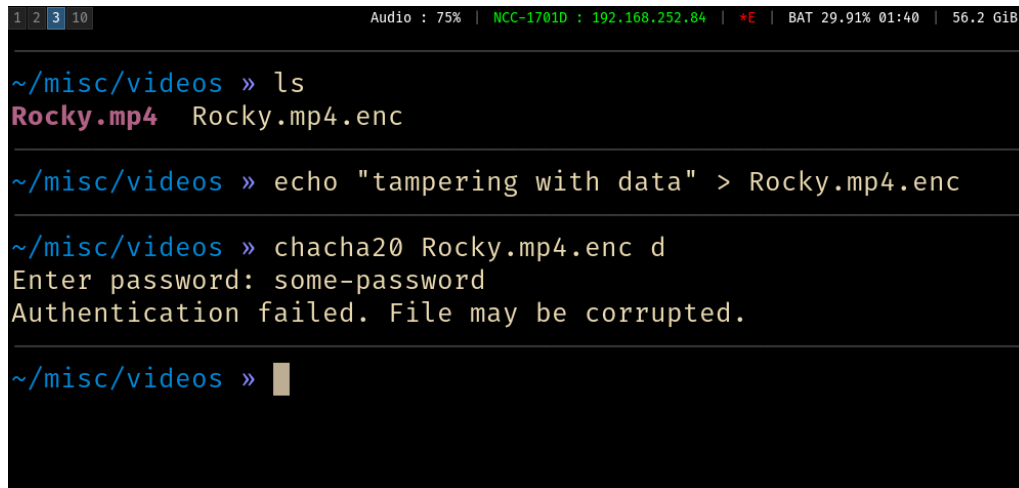
Figure 3.3: Decryption with authenticity validation

This process shows the symmetric-key nature of the algorithm, while also demonstrating the decryption process.

We note that the file size of the original file, encrypted file, and the decrypted file all remain the same.

This consistency in file size is a characteristic of stream ciphers like ChaCha20, where each byte of plaintext is directly transformed into a corresponding byte of ciphertext without altering the overall length. The decrypted file is an exact byte-for-byte replica of the original, confirming the integrity of the encryption and decryption process.

## 3.4 Validation

A terminal window with a dark background and light-colored text. The window title bar at the top shows system status: '1 2 3 10' on the left, and 'Audio : 75% | NCC-1701D : 192.168.252.84 | \*E | BAT 29.91% 01:40 | 56.2 GiB' on the right. The terminal content shows a user in the directory ~/misc/videos. They run 'ls' and see 'Rocky.mp4' and 'Rocky.mp4.enc'. Then they run 'echo "tampering with data" > Rocky.mp4.enc' to modify the file. Next, they run 'chacha20 Rocky.mp4.enc d' to decrypt. The terminal prompts for a password: 'Enter password: some-password'. After entering the password, it shows the error message: 'Authentication failed. File may be corrupted.' The prompt '~ /misc/videos >' is shown again with a cursor, indicating the process has ended.

```
1 2 3 10 Audio : 75% | NCC-1701D : 192.168.252.84 | *E | BAT 29.91% 01:40 | 56.2 GiB

~/misc/videos » ls
Rocky.mp4 Rocky.mp4.enc

~/misc/videos » echo "tampering with data" > Rocky.mp4.enc

~/misc/videos » chacha20 Rocky.mp4.enc d
Enter password: some-password
Authentication failed. File may be corrupted.

~/misc/videos »
```

Figure 3.4: Invalid MAC signature detection and invalidation of data

This test demonstrates the integrity protection provided by ChaCha20 with Poly1305 authentication mechanism.

By modifying even a single byte of the encrypted file, the decryption process fails with an authentication error. This behavior ensures that any unauthorized modification of the ciphertext is immediately detected, protecting the integrity and authenticity of the data.

# Chapter 4

## Conclusion and Future Work

In conclusion, this AAT examined the ChaCha20 stream cipher, focusing on its design, cryptographic security, and practical application when paired with Poly1305 for message authentication. ChaCha20, designed as a secure, high-performance alternative to traditional stream ciphers like RC4 and block ciphers such as AES, exhibits a unique approach based on non-cryptanalytic techniques. This methodology has proven to provide strong security guarantees, making ChaCha20 an effective solution for mitigating common attacks such as those targeting older ciphers. Furthermore, the careful choice of rotation and quarter-round operations within ChaCha20 ensures that it remains resistant to differential and linear cryptanalysis, solidifying its trustworthiness in cryptographic systems.

Then we also emphasized the significance of Poly1305 as a message authentication code (MAC), which, when combined with ChaCha20, enables authenticated encryption. Poly1305's ability to provide both confidentiality and integrity through a single operation is crucial, as it mitigates the risk of potential man-in-the-middle attacks and guarantees that the integrity of transmitted data remains intact. This combination of ChaCha20 with Poly1305 has garnered widespread adoption in protocols like TLS and is featured in popular systems such as Google's QUIC protocol, where both speed and security are paramount.

Performance evaluation results presented in this work further underscore the practical advantages of ChaCha20. Unlike AES, which can be dependent on hardware acceleration for optimal performance, ChaCha20 is highly efficient in software implementations, making it particularly well-suited for environments with limited hardware support or resource constraints.

Ultimately, this work reinforces the credibility and utility of ChaCha20 as a highly secure and efficient cryptographic solution. The combination of ChaCha20 and Poly1305 offers a versatile and reliable approach to authenticated encryption, making it an attractive alternative to established cryptographic schemes. As cryptographic needs evolve, ChaCha20-Poly1305 remains a strong candidate for future developments in cryptographic systems, providing a balanced approach to both speed and security[15].

# Bibliography

- [1] Daniel J Bernstein et al. Chacha, a variant of salsa20. In *Workshop record of SASC*, volume 8, pages 3–5. Citeseer, 2008.
- [2] Wikipedia Contributors. Chacha20-poly1305, 2025. Accessed: 2025-05-16.
- [3] Goutam Paul and Subhamoy Maitra. *RC4 stream cipher and its variants*. CRC press, 2011.
- [4] Andreas Klein. Attacks on the rc4 stream cipher. *Designs, codes and cryptography*, 48:269–286, 2008.
- [5] Scott Fluhrer, Itsik Mantin, and Adi Shamir. Weaknesses in the key scheduling algorithm of rc4. In *Selected Areas in Cryptography: 8th Annual International Workshop, SAC 2001 Toronto, Ontario, Canada, August 16–17, 2001 Revised Papers 8*, pages 1–24. Springer, 2001.
- [6] Daniel Bernstein. Salsa20 security. WWW <http://cr.yp.to/snuffle/security.pdf>, 2005.
- [7] Yukiyasu Tsunoo, Teruo Saito, Hiroyasu Kubo, Tomoyasu Suzaki, and Hiroki Nakashima. Differential cryptanalysis of salsa20/8. In *Workshop Record of SASC*, volume 28. Citeseer, 2007.
- [8] Fabrizio De Santis, Andreas Schauer, and Georg Sigl. Chacha20-poly1305 authenticated encryption for high-speed embedded iot applications. In *Design, automation & test in europe conference & exhibition (DATE), 2017*, pages 692–697. IEEE, 2017.
- [9] Jean Paul Degabriele, Jérôme Govinden, Felix Gü nther, and Kenneth G Paterson. The security of chacha20-poly1305 in the multi-user setting. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1981–2003, 2021.
- [10] Artur Soboń and Sylwia Stachowiak. Chacha20 cipher cryptanalysis through sat problem solving. In *2024 IEEE 17th International Scientific Conference on Informatics (Informatics)*, pages 355–361. IEEE, 2024.
- [11] Bernhard Jungk and Shivam Bhasin. Don’t fall into a trap: Physical side-channel analysis of chacha20-poly1305. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1110–1115. IEEE, 2017.
- [12] Free Software Foundation. Gnu make manual, n.d. Accessed: 2025-05-16.
- [13] LLVM Project. *Clang Compiler User’s Manual*, 2025. Accessed: 2025-05-16.
- [14] LLVM Team. Addresssanitizer, n.d. Accessed: 2025-05-16.
- [15] DuckDuckGo. Duck.ai, 2025. Accessed: 2025-05-16.