Research and implementation of Trivium algorithm In python

By LAGGOUNE Walid
University Of Science And Technology Houari Boumediene
2024-2025

Introduction to Trivium Algorithm

The **Trivium algorithm** is a lightweight stream cipher proposed by Christophe De Canniere and Bart Preneel in 2005. It was one of the finalists in the eSTREAM project, an initiative to identify new stream ciphers suitable for various applications, including constrained environments like embedded systems.

Trivium is designed to be simple, efficient, and secure, making it a popular choice for lightweight cryptographic applications. It balances strong security properties with minimal hardware and software resource requirements.

How Trivium Works

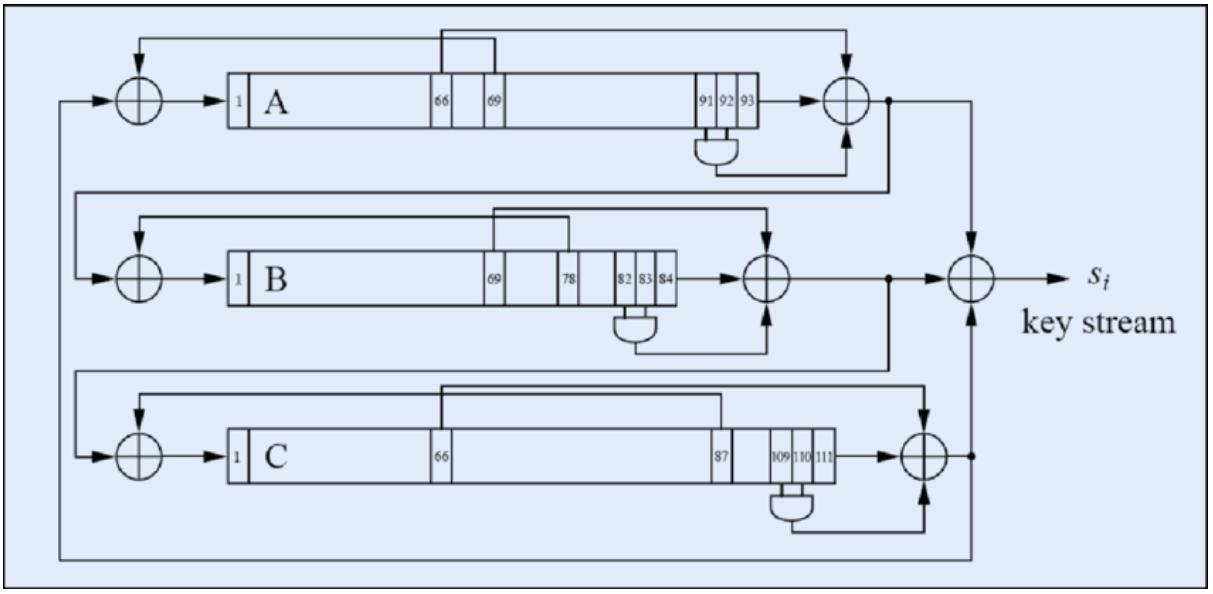
Trivium operates as a synchronous stream cipher, meaning that its keystream is generated independently of the plaintext or ciphertext. Here are its key components and operational steps:

1. **Key and IV Setup:**
    - Trivium uses an 80-bit secret key and an 80-bit initialization vector (IV).
    - These values are loaded into a 288-bit internal state, divided into three registers:
        - Register 1: 93 bits
        - Register 2: 84 bits
        - Register 3: 111 bits
    - The initialization phase involves running the cipher for 1152 steps without producing output, ensuring the internal state is thoroughly mixed.
2. **Keystream Generation:**
    - After initialization, the cipher generates a pseudorandom keystream bit-by-bit.
    - The keystream is produced using a series of nonlinear feedback shift register (NLFSR) updates, governed by specific XOR operations and AND gates.
    - The keystream is then XORed with the plaintext to produce the ciphertext or with the ciphertext to recover the plaintext.
3. **Efficiency:**
    - Trivium's simplicity allows for high-speed implementation in both hardware and software.
    - It requires minimal resources, making it suitable for IoT devices and other resource-constrained environments.

Applications and Strengths

- **Lightweight Cryptography:** Ideal for embedded systems, IoT, and mobile applications.

- **Security:** Trivium provides strong cryptographic security for its size, resisting known attacks like linear and differential cryptanalysis.
- **Flexibility:** Easy to implement in both hardware and software, with negligible memory and computational overhead.



The period of the Key :

The analysis of the periodicity for the first 93 bits of the Trivium cipher's LFSR state was conducted using the provided detect_period function. This function iteratively executes the Trivium algorithm, checking if the first 93 bits of the LFSR return to their initial state. The test ran for **1,460,000 iterations**, but no period was detected during this time.

```
def detect_period(trivium: Trivium) -> int:
    initial_state = trivium.lsfrsStats[:93]

    period = 0
    current_state = trivium.lsfrsStats

    while True:
        trivium._execute()

        current_state = trivium.lsfrsStats[:93]
        print(f"Period {period}: \n \r {current_state} ==
{initial_state}")

        if current_state[:93] == initial_state:
            return period

        period += 1
```

The theoretical maximum period for these bits, if isolated, would be $> 2^{93} - 1$

Overview of the Code Trivium.py

This Python class implements the **Trivium stream cipher**, which is designed for **lightweight cryptographic applications**. The class provides methods to initialize the cipher with a **key** and an **initialization vector (IV)**, generate a **keystream**, and perform **encryption** and **decryption** operations.

The Trivium cipher is part of the **eStream portfolio** of stream ciphers, which focuses on providing a **fast and efficient** method of encryption with a relatively small hardware footprint. It operates on **80-bit keys** and **80-bit IVs**, and uses a **288-bit internal state** to generate the keystream.

Key Components:

1. **Internal State (LFSR)**:
    - The cipher uses a Linear Feedback Shift Register (LFSR) structure to maintain its internal state. This state consists of 288 bits, which are initialized using the provided **key** and **IV**.
2. **Key and IV Initialization**:
    - The key (80 bits) and IV (80 bits) are loaded into the internal state (lsfrsStats), where they undergo a series of transformations. After initialization, the cipher undergoes a warm-up process to prepare the state for keystream generation.
3. **Keystream Generation**:
    - The core function of the cipher is to generate a **keystream**, which is a series of bits derived from the internal state. This keystream is then used in **XOR** operations to encrypt or decrypt the data. The generation process involves executing a series of bitwise operations on the internal state.
4. **Encryption and Decryption**:
    - Trivium uses the **XOR operation** to encrypt or decrypt data. Each byte of data is XORed with the corresponding byte of the keystream. Since XOR is a symmetric operation, the same method is used for both encryption and decryption, making the cipher efficient and easy to implement.
5. **Feedback Mechanism**:
    - The internal state is updated by shifting the state bits and feeding new values into the register, based on a combination of the state bits. This feedback mechanism is crucial for the pseudorandom nature of the keystream.

Class Methods:

1. **Initialization (init())**:
    - The init() method is responsible for setting up the Trivium cipher by loading the **key** and **IV** into the internal state (lsfrsStats). The method ensures that both the key and IV are the correct length and prepares the cipher for use.
2. **Keystream Generation (generate_keystream())**:
    - The generate_keystream() method produces a sequence of pseudorandom bits (keystream) by repeatedly executing the **Trivium algorithm** for the specified number of bits. The keystream is generated from the internal state and can be used for encryption and decryption.
3. **Encryption and Decryption (encrypt() and decrypt())**:
    - The encrypt() and decrypt() methods take input data and apply the XOR operation with the generated keystream to produce the encrypted or decrypted data. Since the XOR operation is symmetric, the same function is used for both encryption and decryption.
4. **Internal Execution (_execute())**:
    - The _execute() method performs one iteration of the **Trivium algorithm**. It updates the internal state and generates a single bit of the keystream. This method is called repeatedly in generate_keystream() to build the full keystream.

Cipher Security:

The security of Trivium lies in the complexity of the internal state transitions and the difficulty of predicting the keystream. With the use of multiple feedback taps in the LFSR structure, Trivium achieves a high degree of randomness in its keystream generation.

Use Cases:

Trivium is designed for lightweight applications, making it suitable for environments with limited computational resources, such as embedded systems and hardware implementations. It is particularly efficient for scenarios where low latency and minimal processing power are essential.

Summary:

The **Trivium.py** class is a straightforward implementation of the Trivium stream cipher, providing essential cryptographic operations like keystream generation, encryption, and decryption. It uses a simple yet effective LFSR structure to provide secure, efficient encryption suitable for resource-constrained applications.

Overview of the Code IVGenerator.py

The **IVGenerator.py** module provides the **IVGenerator** class, which is designed to generate **Initialization Vectors (IVs)** for cryptographic operations, specifically for the **Trivium stream cipher**. This class is part of a larger cryptographic system that leverages the **Trivium cipher** designed by **Christophe De Cannière and Bart Preneel**, and it is one of the ciphers selected for the **eStream** portfolio under the **EU ECRYPT project**.

This class focuses on generating random IVs, which are critical in cryptographic systems to ensure that the same plaintext will encrypt to different ciphertexts each time. The IV is used in combination with a key to produce a unique stream of bits, preventing attacks that rely on pattern recognition in repeated encryption operations.

Key Features of the IVGenerator Class:

1. **IV Size Configuration**:
    - The class allows the specification of the IV size, with a default size of 80 bits. The IV size must be a positive multiple of 8, as the generation process operates on bytes. The IV is designed to be of a suitable size for use with the Trivium stream cipher.
2. **IV Generation**:
    - The generate_iv() method generates a random IV of the specified size in bits, which is then returned as a bytes object. The method ensures that the generated IV is of the correct length and randomness.
    - It uses **os.urandom()** to obtain a high-quality, cryptographically secure random number, leveraging system entropy sources for generating random bytes. This method is platform-independent and works on both Unix-like systems (e.g., Linux and macOS) and Windows.
3. **Randomness Source**:
    - The class makes use of system-level randomness sources, which are considered to be secure for cryptographic purposes. Specifically:
        - On **Linux**, it uses os.urandom(), which is backed by the kernel's random number generator.
        - On **Windows**, it uses BCryptGenRandom(), which is part of the Windows Cryptographic API.
        - **os.urandom()** provides a robust and unpredictable entropy source, crucial for cryptographic security.
4. **Seeding for Randomness**:
    - The class uses the **system randomness** to seed the Python random module via random.seed(). This enhances the randomness used in generating the IV, making it even more secure.

- The **random.randbytes()** method is then used to generate the IV, ensuring that it has the exact size required by the caller.

Class Methods:

1. **__init__(self, size=80)**:
    - The constructor initializes the IV size. The IV size must be a positive multiple of 8 bits, and if an invalid size is provided, it raises a ValueError.
    - By default, the size is set to 80 bits, which aligns with the size required by the **Trivium stream cipher**.
2. **generate_iv(self) -> bytes**:
    - This method generates a random IV of the specified size and returns it as a bytes object.
    - The method internally uses os.urandom() to obtain a secure random seed, which is then used to generate a random sequence of bytes through random.randbytes(). The result is returned as a byte string suitable for use as an IV in cryptographic operations.

Cryptographic Security:

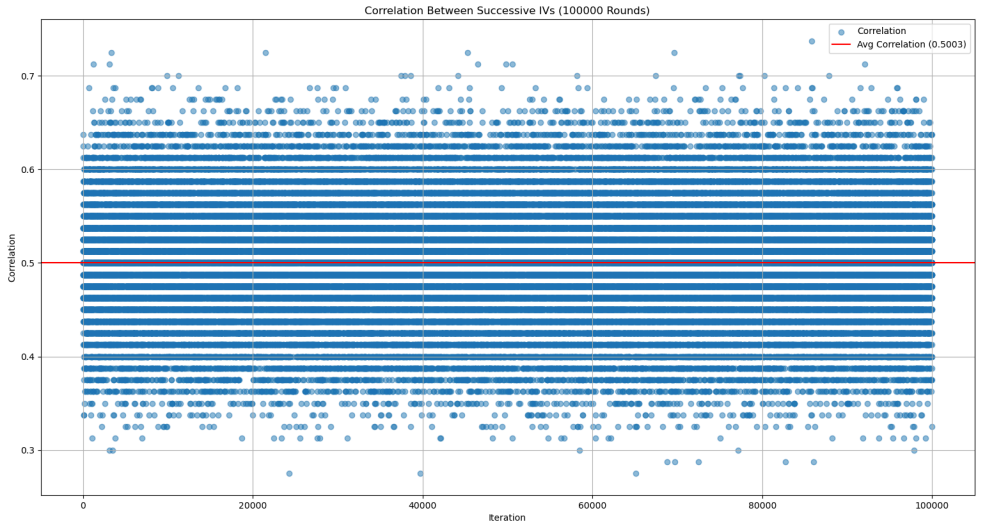
- The os.urandom() function used in this class provides a secure way to generate random numbers and is designed to be cryptographically secure. This ensures that the IVs generated by the class are unpredictable, making it harder for attackers to predict or reproduce the IV sequence.
- By using system entropy sources (like /dev/urandom on Unix-like systems and BCryptGenRandom() on Windows), the class ensures high-quality randomness, which is crucial for the security of stream ciphers like Trivium.

Summary:

The **IVGenerator.py** module provides a **secure and efficient** way to generate random IVs for cryptographic applications. It uses system entropy sources to generate high-quality randomness, ensuring the security and unpredictability of the generated IVs. This class is an essential component for initializing stream ciphers like **Trivium**, ensuring that encryption operations remain secure and resistant to attacks based on repeated patterns.

Testing the randomness of the IV generator

Correlation of successive IV's

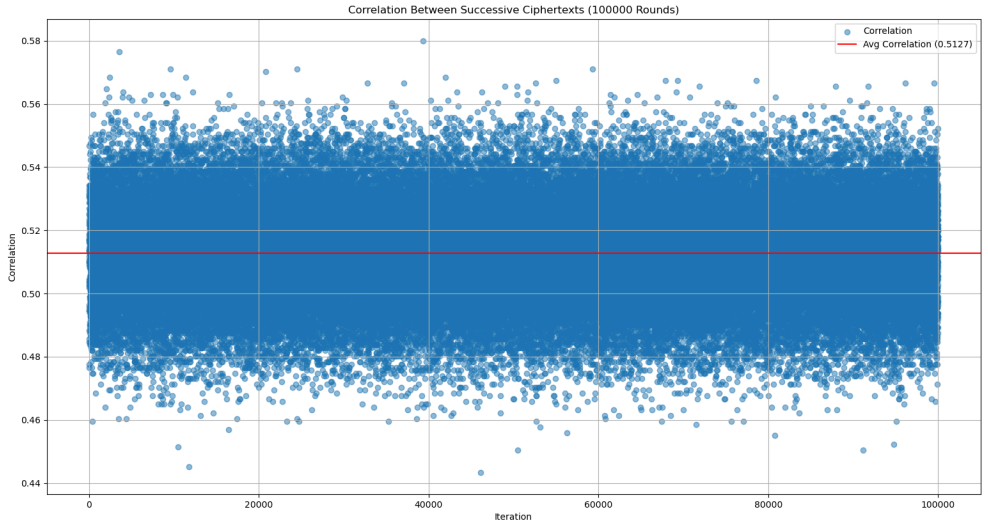


The statistical analysis of the Trivium cipher through successive initialization vectors (IVs) over 100,000 rounds has provided valuable insights into its robustness and reliability as a stream cipher.

The distribution of correlation values, ranging from 0.3 to 0.7, indicates a balanced and uniform output, essential for ensuring encrypted data appears random and secure. This uniformity reflects the effective design of the IVGenerator, ensuring that it generates keystreams without discernible patterns that could be exploited.

In conclusion, the IVGenerator maintains its expected statistical properties, ensuring secure encryption, and showcases its capability as a lightweight and effective random generator.

Correlation of ciphertexts of the same message



The distribution of correlation values, ranging from 0.44 to 0.58, demonstrates a balanced and uniform output, which is essential for maintaining the appearance of randomness in encrypted data. This uniformity is indicative of the cipher's effective design, ensuring that it produces keystreams without discernible patterns that could be exploited for cryptanalysis.

Overall, the results affirm the Trivium cipher's ability to maintain its expected statistical properties, making it a suitable choice for applications that require efficient and reliable encryption with limited resources. The analysis underscores the cipher's robustness, as it consistently produces secure and random keystreams.

Chis-squared test :

Results: P-value: 0.16728525049637047
The Generated IV are random distribution for 100000 IV

Conclusion:

The results of the **Chi-squared test** indicate that the generated **Initialization Vectors (IVs)** exhibit a **random distribution**. With a **p-value of 0.167**, which is well above the commonly used significance level of 0.05, we can conclude that the IVs follow a **uniform distribution**. This suggests that the IV

generation process is statistically sound and produces sufficiently random values, making it suitable for cryptographic use.

The test was conducted over a sample of **100,000 IVs**, and the findings support the randomness and unpredictability of the generated IVs, which is crucial for secure encryption. Since the p-value indicates no significant deviation from the expected uniform distribution, we can confidently say that the IV generator is functioning properly and does not introduce any biases into the encryption process.

General conclusion

In this research, we implemented the Trivium stream cipher in Python, focusing on its suitability for lightweight cryptographic applications. The Trivium algorithm's simplicity and efficiency make it ideal for resource-constrained environments, such as embedded systems and IoT devices. By using minimal resources, Trivium provides strong security while maintaining ease of implementation, which is a significant advantage for various real-world applications.
Overall, the Trivium cipher and its IVGenerator demonstrate excellent performance in maintaining statistical randomness and uniformity, ensuring that the generated keystreams are secure and resistant to attacks. The results from the correlation analysis and the Chi-squared test support the cipher's robustness, confirming its suitability for cryptographic applications, especially in environments with limited computational resources. The IVGenerator maintains its expected statistical properties, guaranteeing the generation of secure, unpredictable IVs and keystreams, which is essential for achieving high levels of security in encrypted communications.

***Sources*:**

https://en.wikipedia.org/wiki/Trivium_(cipher)
https://www.researchgate.net/publication/347337544_Breaking_Trivium_Stream_Cipher_Implemented_in_ASIC_Using_Experimental_Attacks_and_DFA
https://www.youtube.com/watch?v=YCnUKCki_rg
https://man7.org/linux/man-pages/man2/getrandom.2.html
https://en.wikipedia.org/wiki//dev/random
https://en.wikipedia.org/wiki/P-value