1. Introduction

This report analyzes a modified implementation of the ChaCha stream cipher, named "ChaC." The goal is to understand the changes made to the original design, and apply cryptanalytic methods to evaluate its security.

2. The Original ChaCha Cipher

ChaCha is a stream cipher designed by Daniel J. Bernstein, known for its speed and security. It operates on a 512-bit state matrix (16 x 32-bit words) composed of constants, a 256-bit key, a 64-bit nonce, and a 64-bit block counter. The cipher uses a series of rounds (typically 20), each applying a quarter-round function that mixes the state using addition, XOR, and rotation operations. The key schedule ensures that the key and nonce are properly integrated, and unique nonces are critical for security. The output keystream is XORed with plaintext to produce ciphertext.

Diagram: ChaCha State Matrix

```
| constant | constant | constant | constant |
| key      | key      | key      | key      |
| key      | key      | key      | key      |
| counter  | nonce    | nonce    | nonce    |
```

Quarter-Round Function:

- The quarter-round function operates on four 32-bit words:

$$a = a + b;\ d = d \oplus a;\ d = \mathrm{ROTL}(d, 16)$$

$$c = c + d;\ b = b \oplus c;\ b = \mathrm{ROTL}(b, 12)$$

$$a = a + b;\ d = d \oplus a;\ d = \mathrm{ROTL}(d, 8)$$

$$c = c + d;\ b = b \oplus c;\ b = \mathrm{ROTL}(b, 7)$$

3. The Modified Version

The provided implementation (`chac.c`), described as "ChaC," introduces several changes compared to the standard ChaCha design. Key differences include:

- **Reduced State Size:** ChaC uses a 256-bit state (8 x 32-bit words) instead of ChaCha's 512-bit state (16 x 32-bit words). The state consists of a 128-bit key (4 x 32-bit), a 32-bit counter, a 64-bit nonce, and a 32-bit constant (binary for "ChaC").

- **Modified Key Schedule:** The key schedule is simplified. The block is initialized as follows:
  - block[0-3]: key
  - block[4]: counter
  - block[5-6]: nonce
  - block[7]: constant
- **Quarter-Round Function:** The QR macro is similar to ChaCha, but operates on only 8 words (instead of 16). The rotations used are 16, 12, 8, and 7 bits, matching ChaCha's quarter-round.
- **Keystream Generation:** The keystream function first runs 20 rounds (like ChaCha20), but only on the 8-word state. Then, it performs three additional sets of 2 rounds each ("cheap rotations") to generate more output blocks from the same state, aiming for efficiency.
- **Output Expansion:** Each block generates 4 x 128-bit keystreams (total 512 bits), instead of generating a new 128-bit keystream per block. This is done by reusing the state and applying extra rounds.

Diagram: ChaC Block Structure

```
+-----+-----+-----+-----+-----+-----+-----+------+
| k1  | k2  | k3  | k4  | ct  | n1  | n2  |  C   |
+-----+-----+-----+-----+-----+-----+-----+------+
```

> Where k1-k4 are key words, ct is the counter, n1/n2 are nonce, and C is the constant.

Implications:

- Reducing the state size may decrease diffusion and security, as fewer words are mixed per round.
- Reusing the state for multiple keystream outputs could introduce correlations between outputs, potentially weakening security.
- The simplified key schedule and output expansion may make the cipher more efficient, but at the cost of cryptographic strength.

4. Cryptanalytic Methods Applied

4.1 Differential Cryptanalysis

Differential cryptanalysis examines how differences in plaintext input affect the differences in ciphertext output. For this analysis, pairs of plaintexts with controlled differences were encrypted using the modified cipher. The output differences were recorded and analyzed for patterns that could indicate weaknesses.

Experimental Setup:

- Generate plaintext pairs with specific bit differences (e.g., flipping one bit).
- Encrypt using the ChaC implementation, keeping key and nonce constant.
- Compare ciphertexts and analyze the distribution of output differences.

Example Python Code for Differential Cryptanalysis

```python
import os
import subprocess
import numpy as np

def xor_bytes(a, b):
  return bytes(x ^ y for x, y in zip(a, b))

key = b'\x00' * 16  # Example key
nonce = '0x12345678'
num_tests = 100
bit_positions = [0, 1, 2, 3, 4, 5, 6, 7]
diff_counts = []

for bit in bit_positions:
  diffs = []
  for _ in range(num_tests):
    pt1 = os.urandom(32)
    pt2 = bytearray(pt1)
    pt2[0] ^= (1 << bit)  # Flip one bit in first byte
    with open('pt1.bin', 'wb') as f: f.write(pt1)
    with open('pt2.bin', 'wb') as f: f.write(pt2)
    subprocess.run(['./chac', '-n', nonce, 'pt1.bin', 'ct1.bin'])
    subprocess.run(['./chac', '-n', nonce, 'pt2.bin', 'ct2.bin'])
    with open('ct1.bin', 'rb') as f: ct1 = f.read()
    with open('ct2.bin', 'rb') as f: ct2 = f.read()
    diff = sum(b1 != b2 for b1, b2 in zip(ct1, ct2))
    diffs.append(diff)
  diff_counts.append((bit, np.mean(diffs), np.std(diffs)))

for bit, mean, std in diff_counts:
  print(f'Bit {bit}: Mean diff = {mean:.2f}, Std = {std:.2f}')
```

This code generates random plaintext pairs differing by one bit, encrypts them with ChaC, and computes the average number of differing bytes in the ciphertexts. You can adjust the key, nonce, and number of tests as needed.

Experimental Results:

The following results were obtained by flipping each bit in the first byte of the plaintext and measuring the average number of differing bytes in the ciphertexts:

```
Bit 0: Mean diff = 1.00, Std = 0.00
Bit 1: Mean diff = 1.00, Std = 0.00
Bit 2: Mean diff = 1.00, Std = 0.00
Bit 3: Mean diff = 1.00, Std = 0.00
Bit 4: Mean diff = 1.00, Std = 0.00
Bit 5: Mean diff = 1.00, Std = 0.00
Bit 6: Mean diff = 1.00, Std = 0.00
Bit 7: Mean diff = 1.00, Std = 0.00
```

Interpretation:

These results indicate that flipping a single bit in the plaintext only affects a single byte in the ciphertext, with no diffusion to other bytes. This suggests that the modified ChaC cipher has very poor diffusion and avalanche properties compared to the original ChaCha, where a single bit change should affect many output bytes. This is a significant weakness and could make the cipher vulnerable to differential attacks.

4.2 Statistical/Randomness Testing

Randomness tests were performed to ensure the keystream output is indistinguishable from random data. Tools such as Dieharder or NIST STS were used to analyze large samples of keystream output.

Experimental Setup:

- Generate a large keystream using the ChaC cipher (e.g., 1 million bytes).
- Run statistical tests (Dieharder, NIST STS) on the output.

Sample Command

```
./chac -n 0x12345678 input.txt keystream.bin
dieharder -a -f keystream.bin
```

Results

Dieharder Statistical Test Results (Full):

The full Dieharder output for the ChaC keystream is attached and summarized below. Most tests were PASSED, indicating good statistical randomness. However, a few tests were marked as WEAK:

- `rgb_bitdist` (ntup=8, p-value=0.99836114)
- `rgb_bitdist` (ntup=10, p-value=0.00356960)
- `dab_filltree2` (ntup=1, p-value=0.99918205)

All other tests were PASSED with p-values well within the expected range.

Interpretation:

The ChaC keystream passes the majority of Dieharder randomness tests, suggesting it is statistically close to random. The presence of a few WEAK results (very high or very low p-values) may indicate subtle non-randomness or correlations in the output, but these are not catastrophic failures. Combined with the poor diffusion and avalanche properties observed in the differential cryptanalysis, these results reinforce the recommendation to use ChaC with caution in cryptographic applications. For high-security use, further analysis and improvements to diffusion are advised.

5. Critical Discussion

Strengths

- ChaC retains the basic structure and round function of ChaCha, which is well-studied and robust against many attacks.
- The cipher is efficient and produces a large keystream per block, which may be useful for certain applications where performance is prioritized over security.

Weaknesses and Experimental Findings

- **Poor Diffusion and Avalanche:** Differential cryptanalysis shows that flipping a single bit in the plaintext only affects a single byte in the ciphertext, with no diffusion to other bytes. This is a major weakness compared to the original ChaCha, where a single bit change should affect many output bytes. Such poor avalanche properties make the cipher vulnerable to differential attacks and may allow attackers to infer relationships between plaintext and ciphertext.
- **Statistical Randomness:** Dieharder tests on the ChaC keystream show that most statistical tests are passed, indicating the output is close to random. However, a few tests (e.g., rgb_bitdist, dab_filltree2) are marked as WEAK, suggesting subtle non-randomness or correlations in the output. While not catastrophic, these results reinforce concerns about the cipher's robustness.
- **Key Schedule and Output Expansion:** The simplified key schedule and reuse of state for multiple keystream outputs may introduce correlations and reduce security. This design choice, while efficient, further weakens the cipher's resistance to advanced cryptanalysis.

Literature Comparison

Academic studies of ChaCha (Bernstein, RFC 8439) emphasize the importance of strong diffusion and avalanche properties for security. Reducing the state size or reusing it for multiple outputs is generally discouraged, as it can introduce weaknesses and make the cipher more susceptible to attacks. Similar stream ciphers (e.g., Salsa20) also highlight the need for a large state and sufficient rounds to ensure security.

Recommendations

- The current ChaC design is not recommended for security-critical applications due to its poor diffusion and some statistical weaknesses. If used, it should be limited to scenarios where performance is more important than security, and where the risk of cryptanalysis is low.
- For cryptographic use, consider reverting to the original ChaCha design or increasing the state size and round count to improve diffusion and avalanche properties.
- Further analysis and testing (including more advanced cryptanalytic techniques) are advised before deploying ChaC in any real-world system.

6. Role of AI Tools

AI tools (mainly Copilot) were used to:

- Summarize cryptanalytic methods and relevant literature.
- Suggest experiment setups and analysis techniques.

AI accelerated the research process, provided broad ideas, and helped clarify concepts. However, human expertise was essential for critical analysis and interpretation of results. AI tools were especially useful for quickly reviewing the differences between ChaCha and ChaC, generating experimental code, and summarizing academic literature. The final analysis and recommendations were made based on experimental evidence and cryptographic principles.

7. Conclusion

The modified ChaCha cipher (ChaC) was evaluated using differential cryptanalysis and statistical randomness tests. While the cipher retains some strengths from the original design, the experimental results reveal significant weaknesses in diffusion and avalanche properties, as well as subtle statistical anomalies. These findings indicate that ChaC is not suitable for security-critical applications without further improvements. AI tools were valuable for supporting the analysis, but final conclusions were based on experimental data and cryptographic best practices. Further cryptanalysis and caution are strongly recommended before deploying ChaC in any real-world cryptographic system.