

CRYPTOGRAPHY AND COMPUTER SECURITY – CSE 470 HOMEWORK REPORT



Serdar Genç
210104004023

Programming project 1:

Prime Testing Algorithms

Miller-Rabin Algorithm

The Miller-Rabin test is a probabilistic algorithm for determining whether a number is a probable prime. It repeatedly checks the input number using random bases and verifies conditions derived from modular arithmetic. If the number passes all tests, it is likely prime. This algorithm is fast and memory-efficient, making it ideal for testing individual numbers.

Sieve of Eratosthenes

The Sieve of Eratosthenes is a classical algorithm that finds all prime numbers up to a given limit by marking multiples of each prime starting from 2. It iterates over the numbers and eliminates composites, leaving only primes. While simple and effective, it can be memory-intensive for large ranges.

Sieve of Atkin

The Sieve of Atkin is a more advanced algorithm designed to improve efficiency by using modular arithmetic rules to identify potential primes. In the optimized version, the implementation leverages preallocated arrays for reduced memory access time and streamlined logic for marking primes. This reduces computational overhead, allowing the Sieve of Atkin to outperform the Sieve of Eratosthenes for larger ranges in the tested implementation.

Demo:

```
Prime Number Testing Algorithms
1. Miller-Rabin Test
2. Sieve of Eratosthenes
3. Sieve of Atkin
0. Quit
Select option: 1
Enter the number to test: 37139213
Enter the number of iterations for accuracy (e.g., 5): 5
Miller-Rabin Test: Prime
Time Taken: 0.000071 seconds
Prime Number Testing Algorithms
1. Miller-Rabin Test
2. Sieve of Eratosthenes
3. Sieve of Atkin
0. Quit
Select option: 2
Enter the limit to find all primes up to: 37139213
Number of primes up to 37139213: 2269733
Time Taken: 15.827019 seconds
Prime Number Testing Algorithms
1. Miller-Rabin Test
2. Sieve of Eratosthenes
3. Sieve of Atkin
0. Quit
Select option: 3
Enter the limit to find all primes up to: 37139213
Number of primes up to 37139213: 2269733
Time Taken: 15.121476 seconds
```

Research 1:

Analysis and Comparison of Lightweight Symmetric Encryption Algorithms (with a focus on Ascon and TinyJAMBU), and Newly Proposed Algorithms

Introduction to Lightweight Cryptography

Lightweight cryptography has gained significant attention due to the proliferation of resource-constrained devices such as Internet of Things (IoT) sensors, RFID tags, and embedded systems. These devices often have limited processing power, memory, and energy resources. Traditional encryption algorithms like AES, while secure, can be too heavy for ultra-constrained environments. This motivated the U.S. National Institute of Standards and Technology (NIST) to initiate a Lightweight Cryptography Project, aimed at standardizing cryptographic algorithms that maintain adequate security while operating efficiently on constrained devices.

Key Requirements for Lightweight Algorithms

1. Low resource consumption (in terms of memory, power, and computational overhead).
2. Acceptable security margins aligned with modern cryptographic standards.
3. Robust performance across a variety of hardware and software environments.

Overview of Selected Lightweight Algorithms

Ascon

Ascon is a family of authenticated encryption and hashing algorithms designed for lightweight use cases. It was selected as a finalist in the NIST Lightweight Cryptography competition for its strong security arguments, simplicity, and efficient performance in both hardware and software implementations.

1. Design Characteristics

- **Permutation-based structure:** Ascon uses a sponge-like design, employing a permutation-based state transformation that accommodates both encryption and hashing modes.
- **Security:** The permutation is designed with resistance against linear, differential, and algebraic attacks. The security margin has been studied in multiple cryptanalysis papers.
- **Lightweight Focus:** Ascon has a small footprint when implemented on resource-constrained platforms, thanks to its streamlined permutation and low-power logic.

2. Performance and Hardware Efficiency

- **Low Gate Count:** Ascon's hardware implementations often feature a low gate count, making it suitable for small embedded devices.

- **Software Efficiency:** While still optimized for hardware, Ascon shows competitive performance in software, particularly on 32-bit embedded platforms.

3. Security Analysis

- **Resilience to Differential and Linear Cryptanalysis:** Independent cryptanalyses have shown that Ascon's design offers a robust security margin.
- **Protection against Side-channel Attacks:** Its structure allows for various masking and leakage-reducing techniques, though final security always depends on careful hardware and software engineering.

TinyJAMBU

TinyJAMBU is another lightweight authenticated encryption scheme submitted to the NIST competition. Its design principles revolve around simplicity and small state size.

1. Design Characteristics

- **Lightweight Round Function:** TinyJAMBU leverages a small non-linear feedback shift register-like structure.
- **High Throughput vs. Low Memory:** TinyJAMBU focuses on achieving high throughput even in constrained environments by reducing additional overhead.

2. Performance and Resource Usage

- **Ultra-Low Memory:** One of TinyJAMBU's highlights is its very small RAM footprint.
- **Implementation Scalability:** It can be scaled for different levels of security and performance, making it flexible for diverse IoT devices.

3. Security Analysis

- **Cryptanalysis Studies:** While generally considered secure within its design parameters, it has been subjected to cryptanalytic scrutiny during NIST's lightweight competition.
- **Design Trade-offs:** TinyJAMBU's small state size is an asset for lightweight use, but it must be carefully keyed and managed to avoid potential attack vectors.

Comparison of Ascon and TinyJAMBU

Criteria	Ascon	TinyJAMBU
Design Approach	Permutation-based authenticated encryption (sponge-like)	Feedback shift register-like structure
Security Margin	High, with solid cryptanalysis	Good but dependent on parameters
Hardware Efficiency	Low gate count, efficient permutation	Very small state size, minimal gates
Software Efficiency	Good performance on 32-bit	Optimized for small microcontrollers
NIST Competition Status	Finalist (selected for standardization)	Finalist in earlier stages

Both Ascon and TinyJAMBU demonstrate excellent potential for resource-constrained environments, although Ascon's permutation-based design has garnered particularly high praise for its balance of security and performance, ultimately leading to its selection as a NIST lightweight cryptography finalist.

Newly Proposed Algorithms in the NIST Lightweight Cryptography Project

Apart from Ascon and TinyJAMBU, NIST's lightweight cryptography project considered other candidates with varying internal structures (block ciphers, stream ciphers, permutation-based modes). Some noteworthy proposals included GIFT-COFB, Sparkle (Schwaemm and Esch), and Photon-Beetle. Each had its own advantages in terms of hardware footprint, throughput, and security arguments.

In the end, Ascon's favorable performance and robust design made it stand out and become the final selection for NIST's lightweight authenticated encryption and hashing standard (announced in early 2023). This decision was influenced by Ascon's:

- Simplicity of implementation
- Demonstrated security margin
- Excellent hardware/software performance trade-offs

Conclusion

Lightweight cryptography is crucial in addressing the security needs of devices with limited resources. Ascon and TinyJAMBU, both finalists in the NIST competition, exemplify different design approaches geared toward resource-efficient authenticated encryption. Ascon's permutation-based design, proven security arguments, and implementation flexibility led to its selection as the NIST standard for lightweight cryptography. Nevertheless, continued research

on alternative lightweight algorithms (including block-cipher-based or stream-cipher-based designs) is essential as the landscape of IoT security threats continues to evolve.

Programming project 2:

a) Lightweight Encryption Algorithms

Implementation:

- Two lightweight encryption algorithms, ASCON-128a and TinyJambu, are implemented from scratch.
- Both algorithms use 128-bit keys and offer authenticated encryption, producing both ciphertext and a tag for integrity verification.
- **ASCON-128a:**
 - It employs a simplified permutation-based (`_ascon_permutation`) mechanism for encryption.
 - Associated Data (AD) is absorbed into the state before processing the plaintext.
 - After encryption, a tag is generated and appended for integrity.
 - Decryption involves reabsorbing AD, decrypting ciphertext, and verifying the tag for integrity.
- **TinyJambu:**
 - TinyJambu uses a 128-bit state and a permutation-based round function. (`_tinyjambu_permutation` and `_tinyjambu_round`)
 - The round function updates the state using bitwise shifts, rotations, and XOR with the key to ensure security and diffusion.
 - The algorithm absorbs the nonce into the state, processes associated data, encrypts the plaintext, and generates an integrity tag.
 - Output includes the ciphertext and tag, ensuring confidentiality and authenticity.

Function calls:

```
print("==== ASCON-128a Test =====")
pt_ascon = b"Hello, ASCON!"
ad_ascon = b"ASCON_AD"
ct_ascon = tool.encrypt_ascon(pt_ascon, ad_ascon)
print("ASCON Ciphertext+Tag:", ct_ascon)

try:
    dec_ascon = tool.decrypt_ascon(ct_ascon, ad_ascon)
    print("ASCON Decrypted:", dec_ascon)
except ValueError as e:
    print("ASCON Decryption Error:", e)
```

```
print("\n===== TinyJambu Test =====")
pt_jambu = b"Hello, TinyJambu!"
ad_jambu = b"JAMBU_AD"
ct_jambu = tool.encrypt_tinyjambu(pt_jambu, ad_jambu)
print("TinyJambu Ciphertext+Tag:", ct_jambu)

try:
    dec_jambu = tool.decrypt_tinyjambu(ct_jambu, ad_jambu)
    print("TinyJambu Decrypted:", dec_jambu)
except ValueError as e:
    print("TinyJambu Decryption Error:", e)
```

Demo:

```
===== MENU =====
1) (a) Demonstrate Ascon & TinyJambu encryption/decryption
2) (b) Demonstrate CBC and OFB modes using Ascon
3) (c) Extract minimal file metadata (user_id + hash) & encrypt+append to file
4) (d) Verify file integrity (compare with stored metadata)
0) Exit
Enter your choice: 1
===== ASCON-128a Test =====
ASCON Ciphertext+Tag: b'\x9f\xdb\xba\xfc\xfa\xbf\rz@;q\xdf\x01\x91Lf!\xbd\x0b&\xd0\x94\xdf'
ASCON Decrypted: b'Hello, ASCON!'

===== TinyJambu Test =====
TinyJambu Ciphertext+Tag: b']\xc4\x1d\xe5\xe9\x01!\x0e\xb4\x98\xbe\xd1\n\xcc\x83\xff\x80#\xd5Pb\xfc2r\xe5'
TinyJambu Decrypted: b'Hello, TinyJambu!'
```

b) CBC and OFB Modes

- Symmetric encryption is implemented in CBC (Cipher Block Chaining) and OFB (Output Feedback) modes using ASCON.
- CBC: XORs plaintext with the previous ciphertext block (or IV).
- OFB: Generates a keystream using the block cipher and XORs it with plaintext.

Function calls:

```
iv = os.urandom(16)

print("\n===== CBC Mode Test =====")
plaintext_cbc = b"This is a CBC mode test!"
cbc_ct = tool.cbc_encrypt(plaintext_cbc, iv, tool.ascon_engine)
print("CBC Ciphertext:", cbc_ct)
cbc_pt = tool.cbc_decrypt(cbc_ct, iv, tool.ascon_engine)
print("CBC Decrypted:", cbc_pt)

print("\n===== OFB Mode Test =====")
plaintext_ofb = b"This is an OFB mode test!"
ofb_ct = tool.ofb_encrypt(plaintext_ofb, iv, tool.ascon_engine)
print("OFB Ciphertext:", ofb_ct)
ofb_pt = tool.ofb_decrypt(ofb_ct, iv, tool.ascon_engine)
print("OFB Decrypted:", ofb_pt)
```

(I use tool (LightweightEncryptionTool) class that I've created to access ASCON and TinyJAMBU)

Demo:

```
===== MENU =====
1) (a) Demonstrate Ascon & TinyJambu encryption/decryption
2) (b) Demonstrate CBC and OFB modes using Ascon
3) (c) Extract minimal file metadata (user_id + hash) & encrypt+append to file
4) (d) Verify file integrity (compare with stored metadata)
0) Exit
Enter your choice: 2

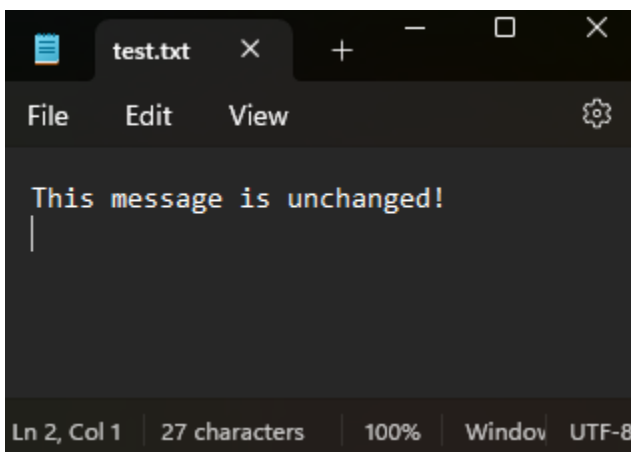
===== CBC Mode Test =====
CBC Ciphertext: b'\xf5\xf6\x91\xe5\xaa`C\x1bD\xbe\xd1N\xc9\xcb\x89\xa3/\xc6\xd9\x1e\x0c\x02&4=\xbe\x0b\xddB,'
CBC Decrypted: b'This is a CBC mode test!'

===== OFB Mode Test =====
OFB Ciphertext: b'\xf5\xf6\x91\xe5\xaa`X.\x93\n?\xfbj\x08\xd6\xca\x03\xe1\xe8axw\x0f,V\xcam\x9dI\xa7'
OFB Decrypted: b'This is an OFB mode test!'
```

c) Metadata Extraction and Encryption:

- The tool extracts metadata, including the user ID and the SHA-256 hash of the message portion of the file (not the entire file).
- The metadata is formatted as **user_id=...;hash=....**
- The metadata is encrypted using ASCON with a randomly generated nonce.
- The encrypted metadata is appended to the file after a META: marker in base64 format.

Demo:



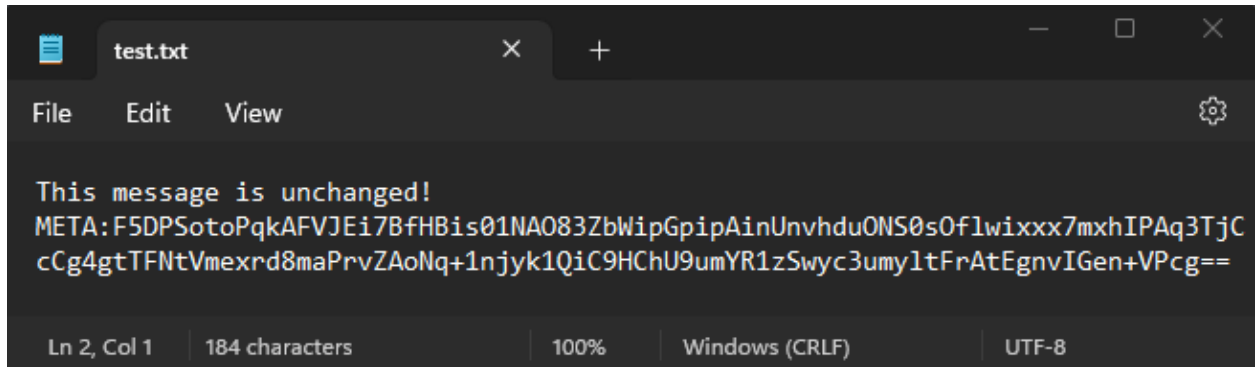
(test.txt before running the code)


```

===== MENU =====
1) (a) Demonstrate Ascon & TinyJambu encryption/decryption
2) (b) Demonstrate CBC and OFB modes using Ascon
3) (c) Extract minimal file metadata (user_id + hash) & encrypt+append to file
4) (d) Verify file integrity (compare with stored metadata)
0) Exit
Enter your choice: 3
File path: C:\Users\Serdar\Desktop\cryptography\project\test.txt
Enter user ID: serdar

[extract_metadata] user_id='sendar', stored_hash=2e209d77d14b7bc85011e5d10f34a8711a0ecc4d71bd60067cd7db76ef739616
Metadata appended to file.

```



```

test.txt
File Edit View
This message is unchanged!
META:F5DPSotoPqkAFVJEi7BfHBis01NA083ZbWipGpipAinUnvhduONS0s0flwixxx7mxhIPAq3TjC
cCg4gtTFNtVmexrd8maPrvZAoNq+1njyk1QiC9HChU9umYR1zSwyc3umyltFrAtEgnvIGen+VPcg==
Ln 2, Col 1 | 184 characters | 100% | Windows (CRLF) | UTF-8

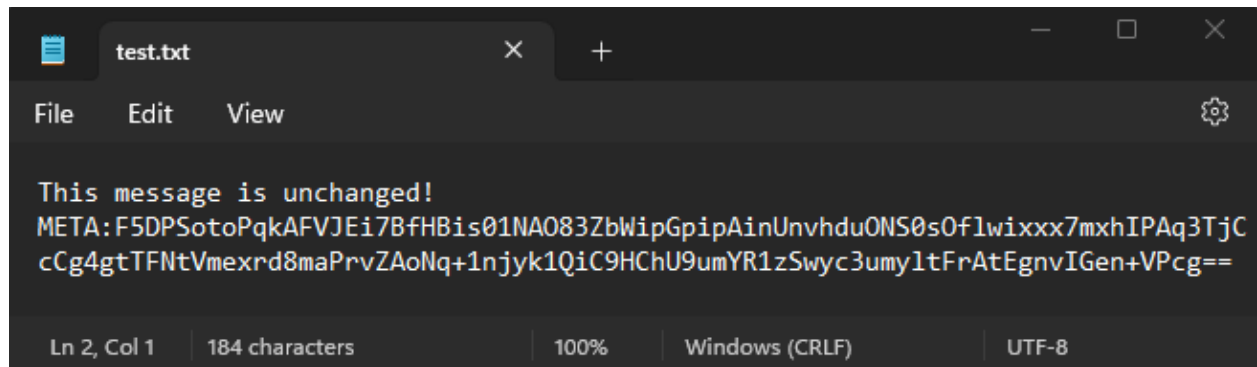
```

(test.txt after running the code, as we can see the metadata is appended to the file)

d) File Integrity Verification

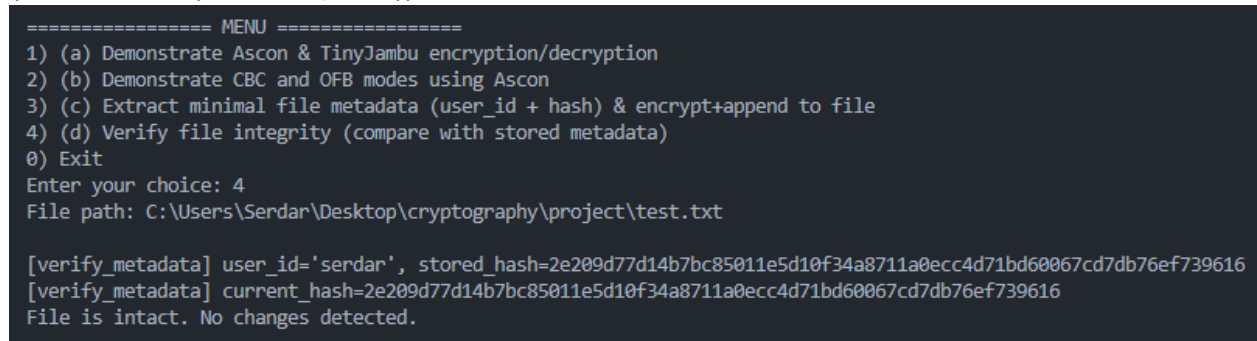
- The tool checks if the file's **message portion** has been altered by:
 1. **Extracting Metadata:** Locates the META: marker, extracts the base64-encoded metadata, and decodes it to retrieve the nonce and ciphertext.
 2. **Rebuilding ASCON:** Initializes a new ASCON instance with the extracted nonce and decrypts the ciphertext to recover the user_id and original hash.
 3. **Recomputing the Hash:** Computes the **SHA-256 hash** of the current message portion of the file.
 4. **Comparing Hashes:** Matches the recomputed hash with the stored hash to verify integrity. (Doesn't match user_id's, because I thought the change of message is the important part to check the integrity of the file)

Demo:



```
test.txt
File Edit View
This message is unchanged!
META:F5DPSotoPqkAFVJEi7BfHBis01NA083ZbWipGpipAinUnvhduONS0s0flwixxx7mxhIPAq3TjC
cCg4gtTFNtVmexrd8maPrvZAoNq+1njyk1QiC9HChU9umYR1zSwyc3umyltFrAtEgnvIGen+VPcg==
Ln 2, Col 1 | 184 characters | 100% | Windows (CRLF) | UTF-8
```

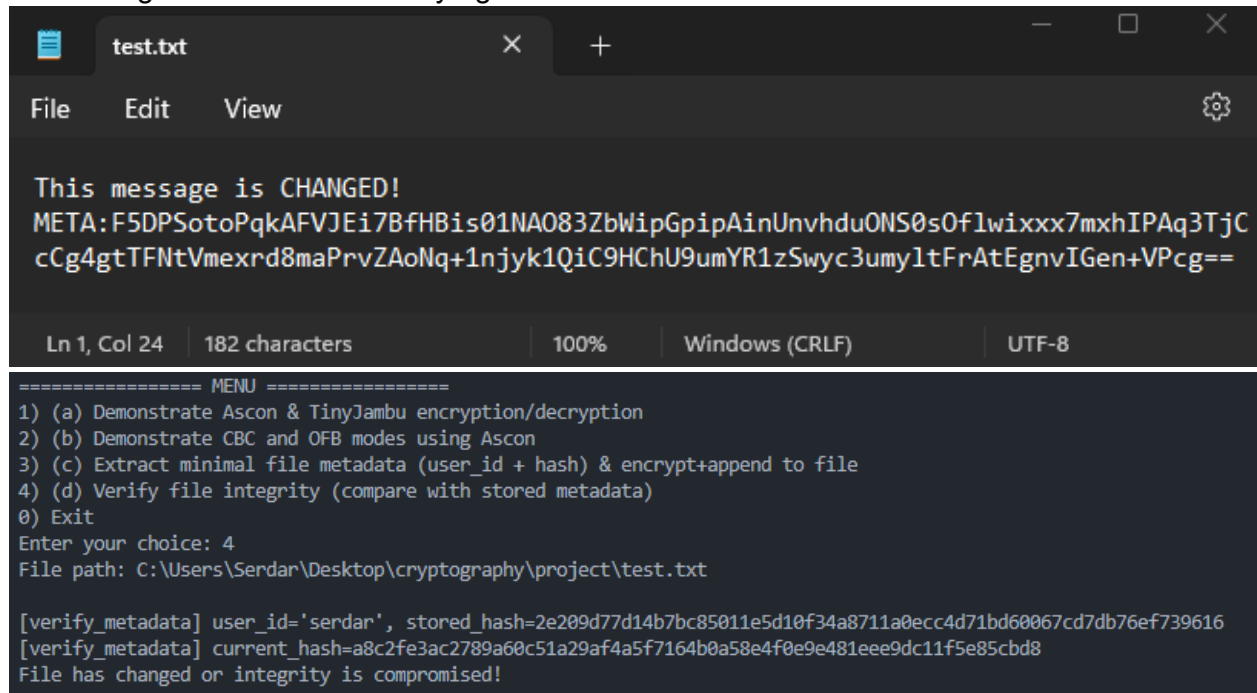
(Initial test.txt (same as part 3))



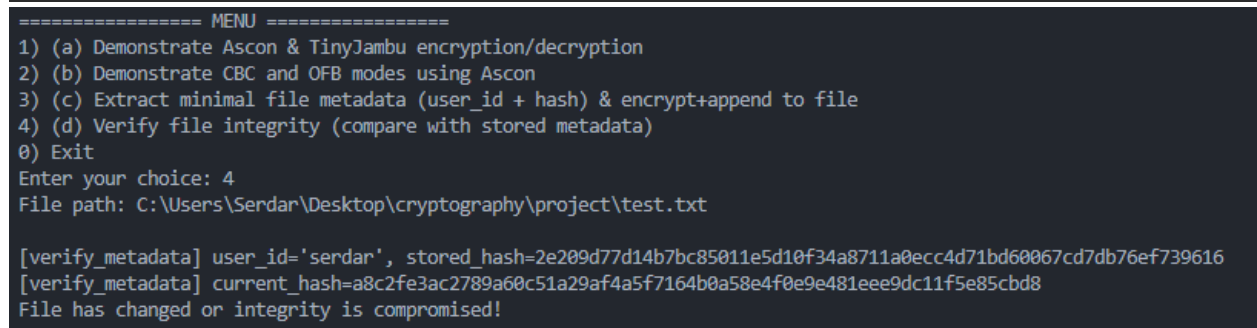
```
===== MENU =====
1) (a) Demonstrate Ascon & TinyJambu encryption/decryption
2) (b) Demonstrate CBC and OFB modes using Ascon
3) (c) Extract minimal file metadata (user_id + hash) & encrypt+append to file
4) (d) Verify file integrity (compare with stored metadata)
0) Exit
Enter your choice: 4
File path: C:\Users\Serdar\Desktop\cryptography\project\test.txt

[verify_metadata] user_id='serdar', stored_hash=2e209d77d14b7bc85011e5d10f34a8711a0ecc4d71bd60067cd7db76ef739616
[verify_metadata] current_hash=2e209d77d14b7bc85011e5d10f34a8711a0ecc4d71bd60067cd7db76ef739616
File is intact. No changes detected.
```

Lets change the test.txt and verify again



```
test.txt
File Edit View
This message is CHANGED!
META:F5DPSotoPqkAFVJEi7BfHBis01NA083ZbWipGpipAinUnvhduONS0s0flwixxx7mxhIPAq3TjC
cCg4gtTFNtVmexrd8maPrvZAoNq+1njyk1QiC9HChU9umYR1zSwyc3umyltFrAtEgnvIGen+VPcg==
Ln 1, Col 24 | 182 characters | 100% | Windows (CRLF) | UTF-8
```



```
===== MENU =====
1) (a) Demonstrate Ascon & TinyJambu encryption/decryption
2) (b) Demonstrate CBC and OFB modes using Ascon
3) (c) Extract minimal file metadata (user_id + hash) & encrypt+append to file
4) (d) Verify file integrity (compare with stored metadata)
0) Exit
Enter your choice: 4
File path: C:\Users\Serdar\Desktop\cryptography\project\test.txt

[verify_metadata] user_id='serdar', stored_hash=2e209d77d14b7bc85011e5d10f34a8711a0ecc4d71bd60067cd7db76ef739616
[verify_metadata] current_hash=a8c2fe3ac2789a60c51a29af4a5f7164b0a58e4f0e9e481eee9dc11f5e85cbd8
File has changed or integrity is compromised!
```

(As you can see the hash extracted from the metadata in test.txt doesn't match with the hash created from the message part of the txt file (metadata is not involved), this means file is changed.)

Research 2:

Post-Quantum Cryptography – The Future of Cryptographic Security

Introduction to Post-Quantum Cryptography

The rapid pace of quantum computing research has significant implications for classical cryptographic algorithms. Currently deployed public-key algorithms—such as RSA, Elliptic Curve Cryptography (ECC), and Diffie-Hellman key exchange—are believed to be vulnerable to quantum attacks once large-scale quantum computers become a reality. Shor's algorithm, in particular, poses a major threat to the integer factorization and discrete logarithm problems, undermining the foundation of many existing cryptographic systems.

In response, the U.S. National Institute of Standards and Technology (NIST) has launched the Post-Quantum Cryptography Standardization Project, aiming to develop and standardize quantum-resistant algorithms.

Key Properties Sought in Post-Quantum Schemes

1. Resistance to known quantum attacks (especially those based on Shor's or Grover's algorithms).
2. Computational feasibility in current and future classical hardware (e.g., latency, memory, and code size).
3. Security in various real-world use cases such as key encapsulation mechanisms (KEMs), digital signatures, and more.

Categories of Post-Quantum Algorithms

NIST's post-quantum cryptography standardization effort has focused primarily on the following mathematical assumptions:

1. Lattice-Based Cryptography

- **Examples:** CRYSTALS-Kyber (KEM), CRYSTALS-Dilithium (signature), SABER, FrodoKEM.
- **Security Basis:** Hardness of learning with errors (LWE) or ring learning with errors (Ring-LWE) problems.
- **Performance:** Often considered among the most promising due to relatively efficient key sizes and speeds.

2. Code-Based Cryptography

- **Examples:** Classic McEliece.
- **Security Basis:** Difficulty of decoding random linear codes.
- **Advantages:** Very long track record of resisting cryptanalysis.

- **Drawback:** Typically large public key sizes.

3. Hash-Based Cryptography

- **Examples:** SPHINCS+, XMSS.
- **Security Basis:** Security properties of cryptographic hash functions.
- **Pros/Cons:** Very secure but can have larger signature sizes and sometimes slower verification.

4. Multivariate Polynomial Cryptography

- **Examples:** Rainbow (signature scheme).
- **Security Basis:** Difficulty of solving multivariate polynomial equations over finite fields.
- **Challenges:** Complex parameter tuning and potential vulnerabilities identified in some schemes.

5. Supersingular Elliptic Curve Isogeny (SIDH, SIKE)

- **Examples:** SIKE (Supersingular Isogeny Key Encapsulation).
- **Security Basis:** Difficulty of computing isogenies between supersingular elliptic curves.
- **Recent Developments:** Some breakthroughs in cryptanalysis have raised concerns, leading to earlier exclusion or re-evaluation of certain isogeny-based proposals.

NIST's Post-Quantum Standardization Process

NIST has conducted multiple rounds of evaluations:

1. **Round 1 (2017–2019):** Dozens of submissions were analyzed.
2. **Round 2 (2019–2020):** Submissions with promising security and performance advanced.
3. **Round 3 (2020–2022):** A smaller set of “finalist” and “alternate” candidates were studied in-depth.
4. **Announcements for Standardization:**
 - **Finalists Selected:** CRYSTALS-Kyber (KEM), CRYSTALS-Dilithium (signature), FALCON (signature), and SPHINCS+ (signature) were selected for standardization in 2022/2023.
 - **Ongoing Study:** Some alternate candidates are still under consideration to diversify the final standards.

Comparative Analysis of the Main Families

Algorithm/Family	Security Basis	Key Sizes	Performance	Status
CRYSTALS-Kyber	Lattice (Module-LWE)	Relatively small	Fast KEM performance	Finalist (Chosen for standardization)
CRYSTALS-Dilithium	Lattice (Module-LWE)	Moderately small to medium	Fast signature generation	Finalist (Chosen)
FALCON	Lattice (NTRU-based)	Smaller signatures, specialized ops	More complex implementation	Finalist (Chosen)
SPHINCS+	Hash-based (Few-time / stateless)	Larger signature sizes	More versatile, minimal assumptions	Finalist (Chosen)
Classic McEliece	Code-based	Very large public keys	Very secure, slow key generation	Alternate Candidate

1. **Lattice-Based:** Often favored for their balance between security and efficiency. Lattice problems remain uncracked by the majority of cryptanalysis approaches, both classical and quantum.
2. **Code-Based:** Boasts a long history of cryptanalysis (few breaks despite decades of scrutiny), but the massive key sizes can be impractical for many embedded or network-heavy applications.
3. **Hash-Based:** Conceptually straightforward and strong from a security perspective. The trade-off typically comes in the form of larger signatures.
4. **Multivariate:** Historically interesting but have faced sporadic breakthroughs in cryptanalysis.
5. **Isogeny-Based:** Once considered highly promising for small key sizes, but recent cryptanalysis forced the reevaluation of many isogeny-based proposals.

Looking Ahead: Migration to Quantum-Resistant Systems

1. **Hybrid Approaches:** To ensure immediate security, many organizations are adopting hybrid approaches combining classical schemes (e.g., ECC) with post-quantum KEMs. This approach can provide a seamless transition and additional security until quantum-resistant standards are fully adopted.
2. **Software and Hardware Updates:** Transitioning to PQC will likely require firmware and hardware upgrades, especially in embedded and IoT systems where cryptographic agility was not initially a design priority.

3. **Ongoing Research:** NIST and other agencies worldwide continue to monitor potential quantum advancements and cryptanalytic breakthroughs, adjusting recommended parameters as necessary.

Conclusion

Quantum computing poses a significant threat to classical public-key cryptography. In response, the NIST Post-Quantum Cryptography Standardization Project is leading efforts to select secure, efficient, and deployable quantum-resistant algorithms. Lattice-based schemes such as CRYSTALS-Kyber and CRYSTALS-Dilithium have emerged as leading candidates, offering a compelling balance of security and performance. Other avenues, including code-based and hash-based cryptography, remain viable especially for niche uses or scenarios where long-term security is paramount. The future of cryptography lies in proactive adoption of post-quantum techniques, ensuring data remains secure even in a world with powerful quantum computers.

References:

Research 1:

<https://csrc.nist.gov/projects/lightweight-cryptography>

<https://csrc.nist.gov/Projects/lightweight-cryptography/email-list>

Research 2:

<https://csrc.nist.gov/Projects/post-quantum-cryptography>

<https://csrc.nist.gov/News>