CSE470

An Analysis of NIST's Lightweight Symmetric Encryption Algorithm Evaluation and Selection Process, ASCON and Report of 2nd Homework

Şiyar Tarık Özcaner

200104004021

NIST’s Lightweight Symmetric Encryption Algorithm Evaluation and Selection Process

The analysis and comparison of lightweight symmetric encryption algorithms involve evaluating various factors to assess their suitability for constrained environments. These factors typically include performance metrics, security considerations, and resource efficiency.

1. Performance Metrics: When comparing lightweight symmetric encryption algorithms, performance metrics such as speed, code size, and energy consumption are crucial. Algorithms that offer faster encryption and decryption, smaller code size, and lower energy consumption are generally preferred for resource-constrained devices.

2. Security Considerations: The resistance of algorithms to various types of attacks is a critical aspect of the analysis. This includes evaluating their resilience against known cryptographic attacks, as well as their ability to withstand side-channel attacks and fault injection attacks. Algorithms with strong security guarantees are essential for protecting sensitive data in constrained environments.

3. Resource Efficiency: The efficient utilization of resources, including memory and processing power, is a key consideration for lightweight algorithms. Algorithms that can operate effectively within the constraints of embedded systems or IoT devices, without imposing excessive computational or memory overhead, are highly desirable.

4. Flexibility and Adaptability: The ability of algorithms to adapt to different application needs and requirements is also important. This includes support for different key sizes, nonce sizes, and tag sizes, as well as the ability to accommodate various use cases without significant modifications.

5. Post-Quantum Security: With the rise of quantum computing, the evaluation of algorithms for their resistance to quantum threats is becoming increasingly important. While not a primary concern for lightweight cryptography, it is a factor that may be considered in the evaluation of newly proposed algorithms.

In summary, the analysis and comparison of lightweight symmetric encryption algorithms involve a comprehensive assessment of their performance, security, resource efficiency, and

adaptability to different application scenarios. These factors collectively contribute to the selection of algorithms that are well-suited for deployment in resource-constrained environments.

Analysis of NIST's Lightweight Symmetric Encryption Competition's Winner ASCON

The finalist algorithm ASCON is a permutation-based Authenticated Encryption with Associated Data (AEAD) and hashing scheme. It is designed to provide flexibility and functionality while maintaining strong security guarantees. The ASCON family includes different variants with varying key sizes, nonce sizes, tag sizes, and block sizes, making it adaptable to different application needs.

The design principles of ASCON are based on a permutation instantiated with different constants and rounds for different variants. The AEAD variants of ASCON use the monkeyDuplex construction, which includes additional key additions during initialization and finalization. The hash function and extendable output function (XOF) variants of ASCON provide additional flexibility and functionality.

During the evaluation process, ASCON demonstrated strong performance in software and hardware benchmarking, outperforming current NIST standards such as AES-GCM and SHA-2 in various platforms. Its resistance to side-channel and fault attacks was also evaluated, and ASCON showed strong performance in this aspect as well.

One of the key strengths of ASCON is its maturity, having been previously presented and analyzed as part of the CAESAR competition. The AEAD variants of ASCON were selected as the primary choice for lightweight authenticated encryption in the final portfolio of the competition. Additionally, ASCON has undergone extensive third-party analysis and implementations, further demonstrating its strong security and reliability.

Overall, the ASCON algorithm stood out due to its flexibility, strong security, and efficient resource utilization. Its ability to adapt to different application needs, combined with its strong performance and extensive analysis, make it a suitable choice for lightweight symmetric encryption in constrained environments.

Analysis of Lightweight Symmetric Encryption Algorithm Finalist SPARKLE

Part 1: Overview of SPARKLE and the Document

SPARKLE is a family of cryptographic permutations designed for lightweight authenticated encryption and hashing. The document "Lightweight Authenticated Encryption and Hashing using the SPARKLE Permutation Family" provides a comprehensive specification of the SPARKLE permutation family, as well as its applications in hash functions and authenticated encryption algorithms. The document covers various aspects of SPARKLE, including its design rationale, implementation aspects, security analysis, and C implementation details.

The specification document begins with an introduction to SPARKLE, highlighting its key features and its role in the context of lightweight cryptography. It outlines the motivation behind the development of SPARKLE and its applications in hash functions and authenticated ciphers. The document also introduces the related algorithms Esch and Schwaemm, which are built upon the SPARKLE permutation family.

Definition 2.1.1 (Left/Right branches). *We call* left branches *those that correspond to the state inputs* $(x_0, y_0), (x_1, y_1), \ldots, (x_{n_b/2-1}, y_{n_b/2-1})$, *and we call* right branches *those corresponding to* $(x_{n_b/2}, y_{n_b/2}), \ldots, (x_{n_b-2}, y_{n_b-2}), (x_{n_b-1}, y_{n_b-1})$.

Algorithm 2.1 SPARKLE256$_{n_s}$
In/Out: $\big((x_0, y_0), ..., (x_3, y_3)\big), x_i, y_i \in \mathbb{F}_2^{32}$

$(c_0, c_1) \leftarrow$ (0xB7E15162,0xBF715880)
$(c_2, c_3) \leftarrow$ (0x38B4DA56,0x324E7738)
$(c_4, c_5) \leftarrow$ (0xBB1185EB,0x4F7C7B57)
$(c_6, c_7) \leftarrow$ (0xCFBFA1C8,0xC2B3293D)
for all $s \in [0, n_s - 1]$ **do**
  $y_0 \leftarrow y_0 \oplus c_{(s \bmod 8)}$
  $y_1 \leftarrow y_1 \oplus (s \bmod 2^{32})$
  for all $i \in [0, 3]$ **do**
    $(x_i, y_i) \leftarrow A_{c_i}(x_i, y_i)$
  end for
  $\big((x_0, y_0), ..., (x_3, y_3)\big) \leftarrow \mathcal{L}_4\big((x_0, y_0), ..., (x_3, y_3)\big)$
end for
return $\big((x_0, y_0), ..., (x_3, y_3)\big)$

Algorithm 2.2 SPARKLE384$_{n_s}$
In/Out: $\big((x_0, y_0), ..., (x_5, y_5)\big), x_i, y_i \in \mathbb{F}_2^{32}$

$(c_0, c_1) \leftarrow$ (0xB7E15162,0xBF715880)
$(c_2, c_3) \leftarrow$ (0x38B4DA56,0x324E7738)
$(c_4, c_5) \leftarrow$ (0xBB1185EB,0x4F7C7B57)
$(c_6, c_7) \leftarrow$ (0xCFBFA1C8,0xC2B3293D)
for all $s \in [0, n_s - 1]$ **do**
  $y_0 \leftarrow y_0 \oplus c_{(s \bmod 8)}$
  $y_1 \leftarrow y_1 \oplus (s \bmod 2^{32})$
  for all $i \in [0, 5]$ **do**
    $(x_i, y_i) \leftarrow A_{c_i}(x_i, y_i)$
  end for
  $\big((x_0, y_0), ..., (x_5, y_5)\big) \leftarrow \mathcal{L}_6\big((x_0, y_0), ..., (x_5, y_5)\big)$
end for
return $\big((x_0, y_0), ..., (x_5, y_5)\big)$

Algorithm 2.3 SPARKLE512$_{n_s}$
In/Out: $\big((x_0, y_0), ..., (x_7, y_7)\big), x_i \in \mathbb{F}_2^{32}, y_i \in \mathbb{F}_2^{32}$

$(c_0, c_1) \leftarrow$ (0xB7E15162,0xBF715880)
$(c_2, c_3) \leftarrow$ (0x38B4DA56,0x324E7738)
$(c_4, c_5) \leftarrow$ (0xBB1185EB,0x4F7C7B57)
$(c_6, c_7) \leftarrow$ (0xCFBFA1C8,0xC2B3293D)
for all $s \in [0, n_s - 1]$ **do**
  $y_0 \leftarrow y_0 \oplus c_{(s \bmod 8)}$
  $y_1 \leftarrow y_1 \oplus (s \bmod 2^{32})$
  for all $i \in [0, 7]$ **do**
    $(x_i, y_i) \leftarrow A_{c_i}(x_i, y_i)$
  end for
  $\big((x_0, y_0), ..., (x_7, y_7)\big) \leftarrow \mathcal{L}_8\big((x_0, y_0), ..., (x_7, y_7)\big)$
end for
return $\big((x_0, y_0), ..., (x_7, y_7)\big)$

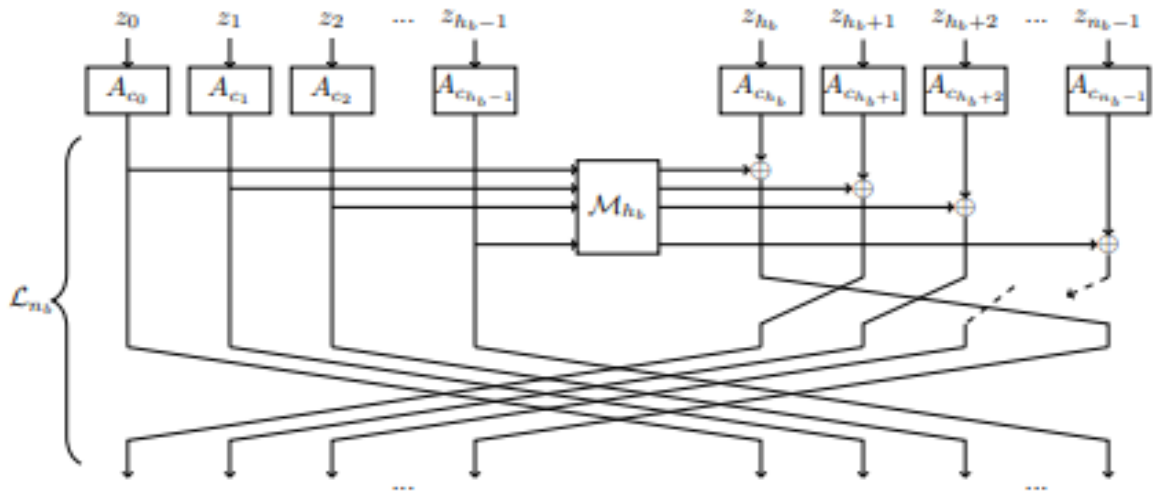


Figure 2.1: The overall structure of a step of SPARKLE. z_i denotes the 64-bit input (x_i, y_i) to the corresponding Alzette instance.

The document then delves into the specification of the SPARKLE permutations, providing detailed insights into the internal structure, operations, and characteristics of the permutation family. It discusses the sponge structure, permutation structure, ARX-box Alzette, linear layer, and the number of steps used in the permutations. Additionally, it presents the design rationale behind the choice of various components of the algorithms, emphasizing the security guarantees and efficiency considerations. the document addresses the implementation aspects of SPARKLE, focusing on software and hardware implementations. It describes the Alzette ARX-box and its efficient implementation on different architectures, highlighting its performance and security characteristics. The document also discusses protection against side-channel attacks and presents implementation results, shedding light on the practical considerations of deploying SPARKLE-based algorithms in real-world systems.

Table 2.3: The instances we provide for authenticated encryption together with their (joint) security level in bit with regard to confidentiality and integrity and the limitation in the data (in bytes) to be processed. The first line refers to our primary member, i.e. SCHWAEMM256-128.

	n	r	c	$\|K\|$	$\|N\|$	$\|T\|$	security	data limit (in bytes)
SCHWAEMM256-128	384	256	128	128	256	128	120	2^{68}
SCHWAEMM192-192	384	192	192	192	192	192	184	2^{68}
SCHWAEMM128-128	256	128	128	128	128	128	120	2^{68}
SCHWAEMM256-256	512	256	256	256	256	256	248	2^{133}

In summary, the document provides a comprehensive overview of SPARKLE, covering its design rationale, implementation aspects, and security analysis. It serves as a valuable resource for researchers, cryptographers, and developers interested in understanding the intricacies of the SPARKLE permutation family and its applications in lightweight authenticated encryption and hashing.

Part 2: Design Rationale and Security Analysis of SPARKLE

The design rationale of SPARKLE is rooted in the selection of components and structures that prioritize security, efficiency, and practical implementation. The document provides a detailed explanation of the design choices, emphasizing the motivation behind the selection of the sponge construction, permutation structure, ARX-box Alzette, and the linear layer.

The sponge construction, chosen for its flexibility and security properties, forms the basis of the SPARKLE permutation family. The document justifies this choice by highlighting the security guarantees and adaptability of sponges to different cryptographic applications. It also addresses

the Long Trail Strategy (LTS) and its adaptation to design sponges with strong security guarantees, providing insights into the hermetic nature of sponges and their resistance to structural distinguishers .

Moreover, the document elaborates on the permutation structure, emphasizing the Long Trail Strategy (LTS) and its role in providing security guarantees against differential and linear cryptanalysis. The LTS, originally introduced in the design of Sparx, is adapted to SPARKLE to ensure robust security against various attacks, including integral, impossible differential, and meet-in-the-middle attacks. This strategic approach to permutation design enables rigorous security arguments and bolsters confidence in the resilience of SPARKLE-based algorithms .

Additionally, the design rationale behind the ARX-box Alzette and the linear layer is elucidated, shedding light on the cryptographic properties and efficiency considerations that influenced their selection as key subcomponents of SPARKLE. The document emphasizes the security guarantees provided by the ARX construction and its suitability for lightweight cryptographic applications, aligning with the overarching goal of achieving fast software encryption for all platforms . The document provides a detailed security analysis of SPARKLE, addressing state-of-the-art attacks and beyond. It emphasizes the cryptographic security notions of preimage resistance, second preimage resistance, and collision resistance, highlighting the security levels offered by the hash functions Esch256 and Esch384. The analysis also extends to the resilience of SPARKLE-based algorithms against differential and linear cryptanalysis, integral attacks, and other potential threats, demonstrating the robustness of the permutation family and its derived algorithms .

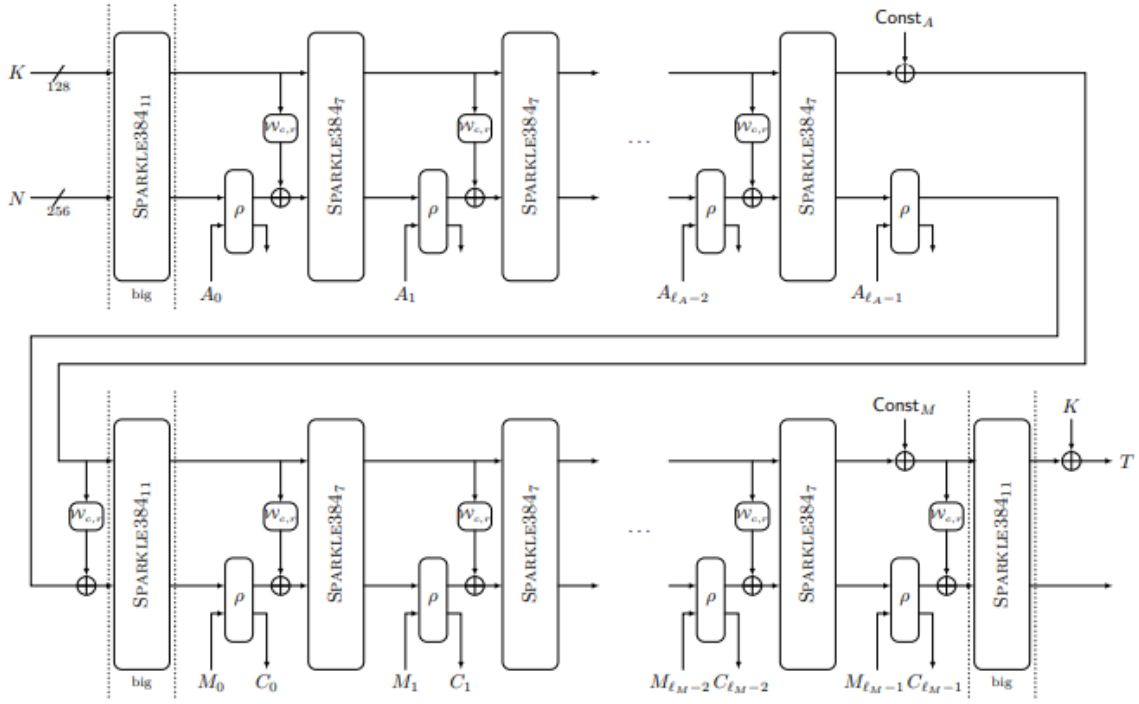


Figure 2.5: The Authenticated Encryption Algorithm SCHWAEMM256-128 with rate $r = 256$ and capacity $c = 128$.

Attack	**Ref.**	SPARKLE256	SPARKLE384	SPARKLE512
Differential cryptanalysis	[BS91]	4	5	6
Linear cryptanalysis	[Mat94]	5	6	6
Boomerang attacks	[Wag99]	3	4	5
Truncated differentials	[Knu95]	2	2	3
Yoyo games	[BBD+99]	4	4	4
Impossible differentials	[Knu98, BBS99]	4	4	4
Zero-correlation	[BR14]	4	4	4
Integral and Division property	[DKR97, KW02, Tod15]	4	4	4
	# steps slim	7	7	8
	# steps big	10	11	12

In conclusion, the design rationale and security analysis presented in the document underscore the meticulous consideration of cryptographic principles, security guarantees, and efficiency requirements in the development of SPARKLE. The comprehensive analysis provides a solid foundation for understanding the resilience and practical applicability of SPARKLE-based algorithms in lightweight authenticated encryption and hashing scenarios.

Part 3: Implementation Aspects and Practical Considerations of SPARKLE

The document delves into the implementation aspects of SPARKLE, providing insights into software and hardware implementations, performance considerations, and practical deployment scenarios. It emphasizes the efficient implementation of the ARX-box Alzette and its suitability for various architectures, highlighting the balance between security bounds and performance considerations.

In the context of software implementations, the document elucidates the characteristics of Alzette, emphasizing its efficient execution on 8 or 16-bit architectures and ARM processors. The carefully chosen rotation amounts enable efficient implementation using move, swap, and 1-bit rotate instructions, contributing to the performance of the permutation. Furthermore, the document highlights the in-register computation of Alzette on AVR, MSP, and ARM architectures, reducing load-store overheads and enhancing performance . The document addresses the consistency of operations across branches and its implications for code size, register utilization, and instruction pipelining. It emphasizes the potential for exploiting instruction-level parallelism using Single Instruction Multiple Data (SIMD) instructions, highlighting the adaptability of SPARKLE-based algorithms to diverse software environments and architectural features .

In the context of hardware implementations, the document underscores the potential for performance gains on processor platforms with vector engines, such as ARM NEON and Intel SSE/AVX. It emphasizes the exploitation of SIMD-level parallelism, enabled by the consistent operations across 32-bit words of the state, and highlights the trade-offs between performance and silicon area in hardware implementations. This insight into hardware-level parallelism underscores the adaptability of SPARKLE-based algorithms to diverse computing platforms and their potential for efficient execution in resource-constrained environments .

Furthermore, the document addresses the practical considerations of deploying SPARKLE-based algorithms in real-world systems, emphasizing the balance between security, performance, and resource constraints. It provides a nuanced understanding of the trade-offs involved in software and hardware implementations, shedding light on the practical considerations that influence the adoption and integration of SPARKLE-based algorithms in lightweight cryptographic applications.

In summary, the implementation aspects and practical considerations presented in the document provide valuable insights into the adaptability, performance characteristics, and deployment considerations of SPARKLE-based algorithms. The comprehensive analysis

underscores the versatility of SPARKLE in diverse computing environments and its potential for efficient and secure implementation in lightweight authenticated encryption and hashing scenarios.

Analysis of Lightweight Symmetric Encryption Algorithm Finalist Romulus

Part 1: Overview of Romulus and its Design Goals

Romulus is a family of authenticated encryption (AE) schemes and a hash function based on a tweakable block cipher (TBC) called Skinny. The primary goal of Romulus is to provide lightweight, highly-secure, and efficient AE schemes based on a TBC. The Romulus submission document introduces three AE variants: Romulus-N, Romulus-M, and Romulus-T, along with the hash function Romulus-H. These variants are designed to cater to different security and efficiency requirements.

The Romulus family of algorithms is characterized by several key features:

1. Security Margin: The underlying TBC, Skinny, offers a comfortable security margin, with the version used in Romulus still maintaining more than 30% security margin in the related-key related-tweakey model. This security margin is further bolstered by the fact that Skinny is being considered for ISO standardization and has been deployed in real-world applications, such as the French Covid tracing application, following a recommendation from the French Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI) .

2. Efficiency: Romulus is designed to have minimal overhead on top of the underlying primitive, resulting in a small area compared to TBC-based designs of similar parameters. The algorithms aim for high efficiency by using a small number of TBC calls and leveraging the performance trade-offs offered by the TBC .

3. Security: The Romulus variants are designed to be secure against different types of adversaries. Romulus-N is a nonce-based AE (NAE) scheme, Romulus-M is a nonce misuse-resistant AE (MRAE) scheme, and Romulus-T is a leakage-resilient AE. Each variant addresses specific security requirements, ensuring a comprehensive approach to security .

Romulus-N, the primary AEAD member of the Romulus family, is particularly noteworthy for its efficiency in handling small messages. It achieves this efficiency without compromising security, making it a compelling choice for lightweight cryptography applications. Romulus-M and Romulus-T cater to scenarios where nonce misuse resistance and leakage resilience are critical, respectively.

In summary, the Romulus family of algorithms is designed to strike a balance between security, efficiency, and lightweight implementation. The variants are tailored to address different security requirements while maintaining a focus on practicality and real-world deployment. This emphasis on versatility and practicality positions Romulus as a strong contender in the field of lightweight cryptography.

Part 2: Security and Efficiency Analysis of Romulus

The Romulus family of algorithms is designed to provide high security while maintaining efficiency and lightweight implementation. In this section, we will analyze the security and efficiency of the Romulus variants in detail.

Security Analysis:

The security of the Romulus family of algorithms is based on the security of the underlying TBC, Skinny. Skinny has been extensively studied and has a comfortable security margin, making it a suitable choice for lightweight cryptography applications. The security of Skinny is based on the standard model security of block ciphers, and the TBC is used as a black-box primitive in the Romulus variants. The main variant, Romulus-N, achieves standard model security .

The security of the Romulus variants is further bolstered by the use of nonce-based and leakage-resilient designs. Romulus-N is a nonce-based AE scheme that provides security against nonce misuse attacks. Romulus-M is a nonce misuse-resistant AE scheme that provides security against nonce misuse attacks. Romulus-T is a leakage-resilient AE scheme that provides security against side-channel attacks .

Efficiency Analysis:

The Romulus family of algorithms is designed to be efficient and lightweight. Romulus-N achieves high efficiency by using a small number of TBC calls and faster MAC computation for associated data. Romulus-M achieves efficiency by using a reduced state size and inverse-freeness, which eliminates the need for TBC inverse computation. Romulus-T achieves efficiency by using a leakage-resilient design that eliminates the need for masking and other countermeasures . The efficiency of the Romulus variants is further enhanced by the use of the TBC, Skinny. Skinny is a highly efficient TBC that offers a range of performance trade-offs. The use of Skinny in the Romulus variants results in a significant boost to software and hardware implementations .

Overall, the Romulus family of algorithms strikes a balance between security and efficiency. The use of Skinny as the underlying TBC and the design choices made in the Romulus variants result in highly efficient and lightweight implementations without compromising security.

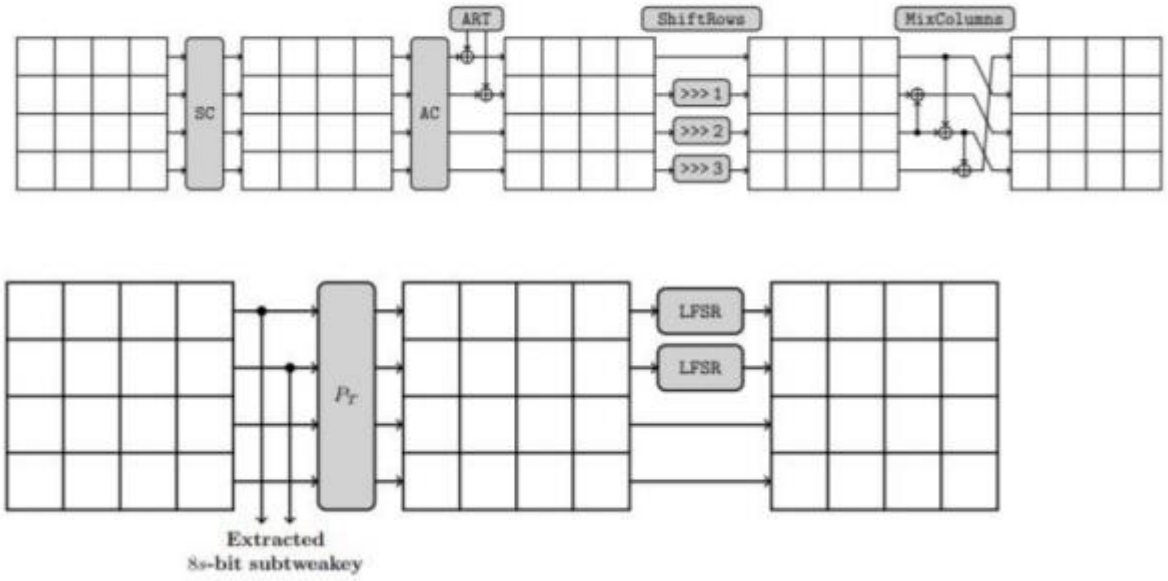
Design Rationale:

The design rationale of the Romulus family of algorithms is based on the goal of achieving lightweight, efficient, and highly-secure AE schemes. The design choices made in the Romulus variants are aimed at achieving this goal while addressing specific security requirements.

Romulus-N is designed to be highly efficient in handling small messages while maintaining security against nonce misuse attacks. The design achieves this efficiency by using a small number of TBC calls and faster MAC computation for associated data. Romulus-M is designed to be efficient while providing security against nonce misuse attacks. The design achieves this efficiency by using a reduced state size and inverse-freeness. Romulus-T is designed to be leakage-resilient while maintaining efficiency. The design achieves this by using a leakage-resilient design that eliminates the need for masking and other countermeasures .

The design choices made in the Romulus variants are based on the use of the TBC, Skinny. Skinny is a highly efficient TBC that offers a range of performance trade-offs. The use of Skinny in the Romulus variants results in highly efficient and lightweight implementations without compromising security. The design choices also take into account the specific security requirements of each variant, resulting in a comprehensive approach to security .

In summary, the Romulus family of algorithms is designed to achieve lightweight, efficient, and highly-secure AE schemes. The design choices made in the Romulus variants are based on the use of the TBC, Skinny, and take into account specific security requirements. The resulting designs strike a balance between security and efficiency, making Romulus a compelling choice for lightweight cryptography applications.



Part 3: Implementation Aspects and Real-World Impact of Romulus

Implementation Aspects:

The Romulus family of algorithms is designed with a focus on practical implementation considerations. The design choices made in the Romulus variants aim to minimize overhead and maximize efficiency, making them well-suited for real-world deployment in lightweight hardware and software environments.

Romulus-N, Romulus-M, and Romulus-T are designed to have minimal overhead on top of the underlying TBC, Skinny. This results in a small area compared to TBC-based designs of similar parameters, making Romulus well-suited for lightweight hardware implementations. The use of Skinny as the underlying TBC allows for efficient and compact implementations, further enhancing the practicality of Romulus .

Real-World Impact:

The Romulus family of algorithms has the potential to make a significant impact in real-world applications, particularly in the context of lightweight cryptography for constrained environments such as IoT devices and embedded systems. The efficiency, security, and practical implementation aspects of Romulus position it as a compelling choice for a wide range of applications. The security margin and efficiency of the Romulus variants make them suitable for deployment in scenarios where lightweight cryptography is essential, such as IoT devices, smart sensors, and other resource-constrained systems. The use of Skinny as the underlying TBC further enhances the practicality of Romulus, making it well-suited for real-world deployment in a variety of applications.

Furthermore, the Romulus family of algorithms has the potential to contribute to the standardization of lightweight cryptographic primitives. The consideration of Skinny for ISO standardization and its deployment in real-world applications, such as the French Covid tracing application, underscore the real-world impact and practical relevance of Romulus .

In conclusion, the Romulus family of algorithms is not only designed with a focus on practical implementation aspects but also has the potential to make a significant impact in real-world applications. The efficiency, security, and practicality of Romulus position it as a compelling choice for lightweight cryptography in a wide range of scenarios, making it a noteworthy contribution to the field of lightweight cryptographic primitives.

Implementations of Romulus, SPARKLE and File Authenticity and Integrity Control Application

I Implemented the SPARKLE encryption algoritihm in Python and Romulus in C to get to use different platforms and see the differences how cryptography and cybersecurity is achieved through different programming languages.

For further details see SPA.py and Romulus.c

```python
# SPARKLE-256 constants
ROUND_CONSTANTS = [
    0x06, 0x0B, 0x0F, 0x0D, 0x05, 0x08, 0x03, 0x0C,
    0x0E, 0x07, 0x01, 0x09, 0x00, 0x02, 0x0A, 0x04
]
ROTATION_CONSTANTS = [
    16, 12, 8, 7, 16, 12, 8, 7, 16, 12, 8, 7, 16, 12, 8, 7
]

# SPARKLE-256 permutation function
def sparkle256(state):
    for i in range(16):
        x, y = state[i % 12], state[(i + 16) % 12]  # Use modulo to cycle through the state
        x ^= ROUND_CONSTANTS[i]
        y = (y + x) & 0xFFFFFFFF
        y = ((y << ROTATION_CONSTANTS[i]) | (y >> (32 - ROTATION_CONSTANTS[i]))) & 0xFFFFFFFF
        x ^= y
        x = (x - y) & 0xFFFFFFFF
        state[i % 12], state[(i + 16) % 12] = x, y
```

SPARKLE's constants and SPARKLE256 permutation mechanism

```python
def sparkle256_encrypt_and_decrypt_demo(mode, key, nonce, plaintext):
    if mode == 'CBC':
        ciphertext = sparkle256_cbc_encrypt(key, nonce, plaintext)
        decrypted_plaintext = sparkle256_decrypt(key, nonce, ciphertext, mode)
    elif mode == 'OFB':
        ciphertext = sparkle256_ofb_encrypt(key, nonce, plaintext)
        decrypted_plaintext = sparkle256_decrypt(key, nonce, ciphertext, mode)
    else:
        raise ValueError("Invalid mode")

    print(f"SPARKLE in {mode} mode")
    print("Plaintext:", plaintext.decode('utf-8', 'replace'))  # Display as UTF-8, replace non-printable characters
    print("Key:", key.hex())
    print("Nonce:", nonce.hex())
    print("Ciphertext:", ciphertext.hex())
    print("Decrypted Plaintext:", decrypted_plaintext.decode('utf-8', 'replace'))  # Display as UTF-8, replace non-printable characters
```

Encrypts and then decrypts plaintext

```
SPARKLE in CBC mode
Plaintext: Hello, SPARKLE!
Key: 3e496bdbd621e5dcb86c1a28c5db067bc5d9590d26a137c209d26f38ca26b03a
Nonce: b164b7caa910a1fa9e2398f3ba26da7f
Ciphertext: 52b587a200000000bb2bb4130000000f60f0109000000004fb9efe00000000
Decrypted Plaintext: Hello, SPARKLE!

SPARKLE in OFB mode
Plaintext: Hello, SPARKLE!
Key: 3e496bdbd621e5dcb86c1a28c5db067bc5d9590d26a137c209d26f38ca26b03a
Nonce: b164b7caa910a1fa9e2398f3ba26da7f
Ciphertext: 69525c896f2c2053a36ca5b34c452111e71e10181111111115ea8fef11111111
Decrypted Plaintext: Hello, SPARKLE!
```

Outputs this

Now for Romulus the application uses Romulus to check for integrity and authenticity

```
int romulusTypes[6][3]={{64,64,32},{64,128,36},{64,192,40},{128,128,40},{128,256,48},{128,384,56}};

// Sbox for 4 bit
const unsigned char sbox_4[16] = {12,6,9,0,1,10,2,11,3,8,5,13,4,14,7,15};

// Sbox for 8-bit
const unsigned char sbox_8[256] = {0x65 , 0x4c , 0x6a , 0x42 , 0x4b , 0x63 , 0x43 , 0x6b , 0x55 , 0x75 , 0x5a , 0x7a , 0x53 , 0x73 , 0x5b , 0x7b ,0x35 , 0x8c , 0x3a

// ShiftAndSwitchRows permutation
const unsigned char P[16] = {0,1,2,3,7,4,5,6,10,11,8,9,13,14,15,12};

// Tweakey permutation
const unsigned char TWEAKEY_P[16] = {9,15,8,13,10,14,12,11,0,1,2,3,4,5,6,7};


// round constants
const unsigned char roundArray[62] = {
    0x01, 0x03, 0x07, 0x0F, 0x1F, 0x3E, 0x3D, 0x3B, 0x37, 0x2F,
    0x1E, 0x3C, 0x39, 0x33, 0x27, 0x0E, 0x1D, 0x3A, 0x35, 0x2B,
    0x16, 0x2C, 0x18, 0x30, 0x21, 0x02, 0x05, 0x0B, 0x17, 0x2E,
    0x1C, 0x38, 0x31, 0x23, 0x06, 0x0D, 0x1B, 0x36, 0x2D, 0x1A,
    0x34, 0x29, 0x12, 0x24, 0x08, 0x11, 0x22, 0x04, 0x09, 0x13,
    0x26, 0x0c, 0x19, 0x32, 0x25, 0x0a, 0x15, 0x2a, 0x14, 0x28,
    0x10, 0x20};
```

Parameters for operation

```
// encryption function of Skinny
void enc(unsigned char* input, const unsigned char* userkey, int ver)
{
  unsigned char state[4][4];
  unsigned char keyCells[3][4][4];
  int i;

  memset(keyCells, 0, 48);
  for(i = 0; i < 16; i++) {
        if (romulusTypes[ver][0]==64){
            if(i&1)
            {
                state[i>>2][i&0x3] = input[i>>1]&0xF;
                keyCells[0][i>>2][i&0x3] = userkey[i>>1]&0xF;
                if (romulusTypes[ver][1]>=128)
                    keyCells[1][i>>2][i&0x3] = userkey[(i+16)>>1]&0xF;
                if (romulusTypes[ver][1]>=192)
                    keyCells[2][i>>2][i&0x3] = userkey[(i+32)>>1]&0xF;
            }
            else
            {
                state[i>>2][i&0x3] = (input[i>>1]>>4)&0xF;
                keyCells[0][i>>2][i&0x3] = (userkey[i>>1]>>4)&0xF;
                if (romulusTypes[ver][1]>=128)
                    keyCells[1][i>>2][i&0x3] = (userkey[(i+16)>>1]>>4)&0xF;
                if (romulusTypes[ver][1]>=192)
                    keyCells[2][i>>2][i&0x3] = (userkey[(i+32)>>1]>>4)&0xF;
            }
        }
        else if (romulusTypes[ver][0]==128){
            state[i>>2][i&0x3] = input[i]&0xFF;
            keyCells[0][i>>2][i&0x3] = userkey[i]&0xFF;
            if (romulusTypes[ver][1]>=256)
                keyCells[1][i>>2][i&0x3] = userkey[i+16]&0xFF;
            if (romulusTypes[ver][1]>=384)
                keyCells[2][i>>2][i&0x3] = userkey[i+32]&0xFF;
        }
  }
```

Encryption function of Romulus

```
// generate test vectors for all the romulusTypes of Skinny
void TestVectors(int ver)
{
  unsigned char p[16];
  unsigned char c[16];
  unsigned char k[48];
  int n;

  for(n = 1; n < 10; n++)
  {
    int i;
    for(i = 0; i < (romulusTypes[ver][0]>>3); i++) c[i] = p[i] = rand() & 0xff;
    for(i = 0; i < (romulusTypes[ver][1]>>3); i++) k[i] = rand() & 0xff;
    enc(c,k,ver);
    dec(c,k,ver);
  }
}
```

Outputs of the integrity and authenticity check

```
Plaintext: GTU
Cipher: 786D68C88A53B29CF64C, Len: 19
Plaintext: GTU, Len: 3
Romulus Cipher passed from the test successfully!
Test of the Romulus Algorithm with Cipher Block Chaining (CBC) mode
Plaintext: GizliBilgi
Key: 0123456789ABCDEF0123456789ABCDEF
Nonce: 000000000000111111111111
Additional Information: GizliBilgi

Plaintext: GizliBilgi
Text Encrypted CBC Mode : zR

Text Decrypted CBC Mode : GizliBilgi
Success, test passed
Test of the Romulus Algorithm with Cipher Block Chaining (OFB) mode
Plaintext: TarikOzcaner
Key: 0123456789ABCDEF0123456789ABCDEF
Nonce: 000000000000111111111111
Additional Information: GizliBilgi
```