**KRİPTOGRAFİ VE BİLGİSAYAR GÜVENLİĞİ (BİL 470) PROJE**

**1801042103**

**Ozan GEÇKİN**

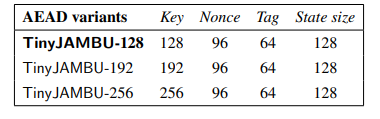
**SUBJECT OF THE ASSIGNMENT**

• Analyze and comparatively explain lightweight symmetric encryption algorithms and newly proposed algorithms.

Algorithms I have to research are TinyJAMBU and Sparkle.

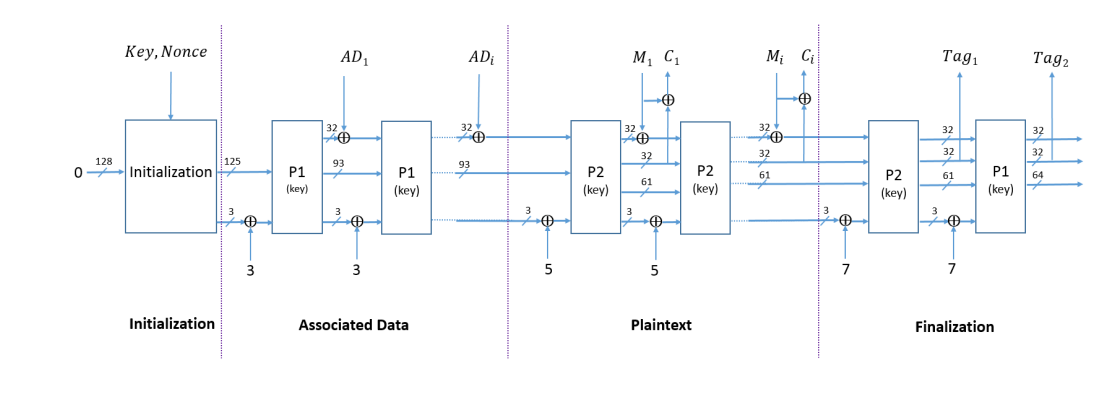
**TinyJAMBU: A Family of Lightweight Authenticated Encryption Algorithms**

TinyJAMBU, designed by Wu and Huang , is an AEAD scheme that is inspired by the third-round candidate of the CAESAR competition, JAMBU. The main component of TinyJAMBU is a 128-bit keyed permutation without a key schedule that is based on a nonlinear feedback shift register, and the nonlinearity in each round is obtained using a single NAND operation.

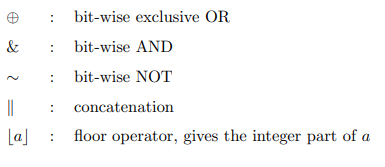


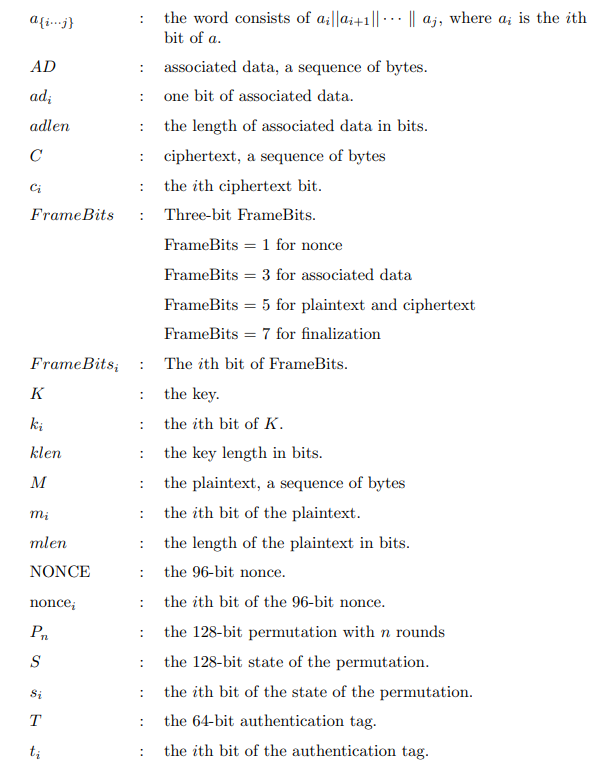
Security Analysis. In , the designers provided a security proof for the mode, analysis of the keyed-permutation, and security against forgery and key recovery attacks (including differential, linear, algebraic, and slide attacks). Saha et al. showed that the security margin of TinyJAMBU is around 12 % due to the dependencies between the outputs of multiple AND gates.

The TinyJAMBU mode is a small variant of the JAMBU mode which is a third-round candidate of the CAESAR competition. In the TinyJAMBU mode, a 128-bit keyed permutation is used, the state size is 128 its, and the message block size is 32 bits. When nonce is reused, the TinyJAMBU mode provides better authentication security than the JAMBU mode. When nonce is reused, the TinyJAMBU mode provides better authentication security than the Duplex mode (for the same permutation size and the same message block size). The reason is that the attacker can easily set part of the state to arbitrary value when nonce is reused in the Duplex mode, while it is difficult to do that in the TinyJAMBU mode. The TinyJAMBU mode is shown in Fig. 2.1. If the last block of the associated data (or plaintext) is not a full block , the length of the partial block (the number of bytes) is xored to the state.

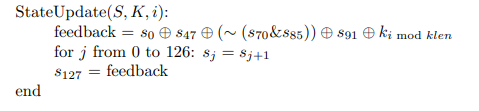
The TinyJAMBU mode for 128-bit state state and keyed-permutations

Operations, Variables and Functions

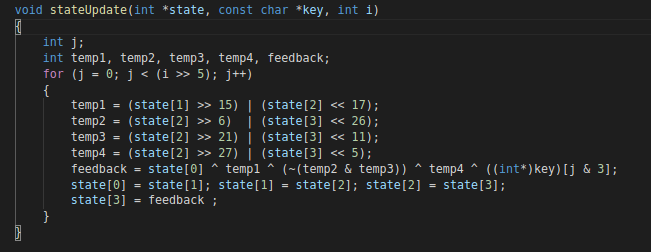
**

**

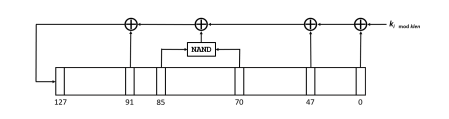
**The Keyed Permutation P**

In TinyJAMBU, a 128-bit keyed permutation is used. The permutation Pn consists of n rounds. In the ith round of the permutation, a 128-bit nonlinear feedback shift register is used to update the state as follows

**Code:**



For example, P384 means that the state of the permutation is updated using the function StateUpdate() for 384 times. 32 rounds of the permutation can be computed in parallel on 32-bit CPU



**The initialization**

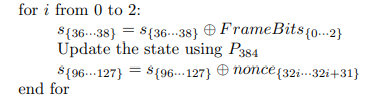
In the keyed permutation of TinyJAMBU-128, the 128-bit key of TinyJAMBU128 is used, and the klen is set to 128. The initialization of TinyJAMBU-128 consists of two stages: key setup and nonce setup.

Key Setup. The key setup is to randomize the state using the keyed permutation P1024.

1. Set the 128-bit state S as 0.

2. Update the state using P1024.

Nonce Setup. The nonce setup consists of three steps. In each step, the Framebits of nonce (the value is 1) are XORed with the state, then we update the state using the keyed permutation P384, then 32 bits of the nonce are XORed with the state.



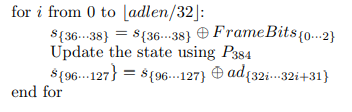
**Code**:

metin, ekran, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Processing the associated data**

After the initialization, we process the associated data AD. In each step, the Framebits of associated data (the value is 3) are XORed with the state, then we update the state using the keyed permutation P384, then 32 bits of the associated data are XORed with the state.



Processing the partial block of associated data. If the last block is not a full block (it is called a partial block), the last block is XORed to the state, and the number of bytes of associated data in the partial block is XORed to the state.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Code:**

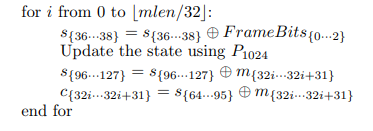
metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**The encryption**

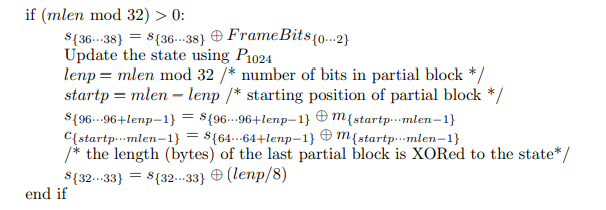
After processing the associated data, we encrypt the plaintext M. In each step, the Framebits of plaintext (the value is 5) are XORed with the state, then we update the state using the keyed permutation P1024, then 32 bits of the plaintext are XORed with the state, and we obtain 32 bits of ciphertext by XORing the plaintext with another part of the state.

**Processing the full blocks of plaintext:**



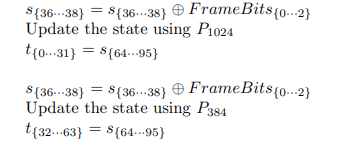
**Processing the partial block of plaintext.**

If the last block is not a full block (it is a partial block), the last block is XORed to the state, and the number of bytes in the partial block is XORed to the state



**The finalization**

After encrypting the plaintext, we generate the 64-bit authentication tag T as follows. The Framebits of finalization (the value is 7) are XORed with the state.



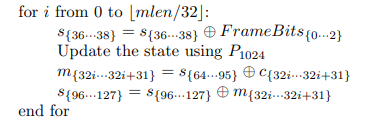
**Code (Processing the full blocks of plaintext, Processing the partial block of plaintext, The finalization):**

**metin içeren bir resim

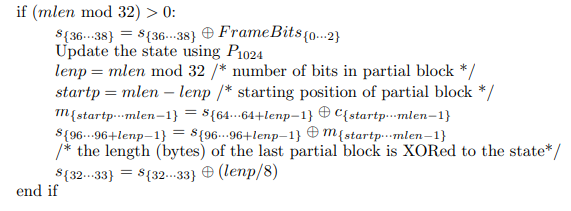
Açıklama otomatik olarak oluşturuldu**

**The decryption**

In a decryption process, the initialization and processing the associate data are the same as the encryption process. After processing the associated data, we decrypt the ciphertext C. In each step, the Framebits of plaintext (the value is 5) are XORed with the state, then we update the state using the keyed permutation P1024. We obtain 32 bits of plaintext by XORing the ciphertext with 32 state bits s{64···95}, then the plaintext is XORed with the state bits s{96···127}.

****Processing the full blocks of ciphertext:

Processing the partial block of ciphertext. If the last block is not a full block (it is a partial block), the number of bytes in the partial block is XORed to the state.



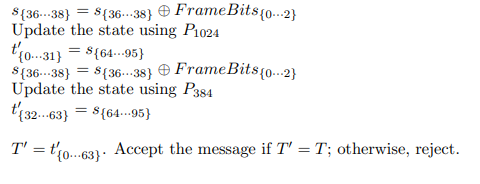
**Code:**

**metin içeren bir resim

Açıklama otomatik olarak oluşturuldu**

**The verification**

After decrypting the plaintext, we generate a 64-bit authentication tag T 0 , then compare T 0 with the received tag T. The Framebits of finalization are of value 7



**Code:**

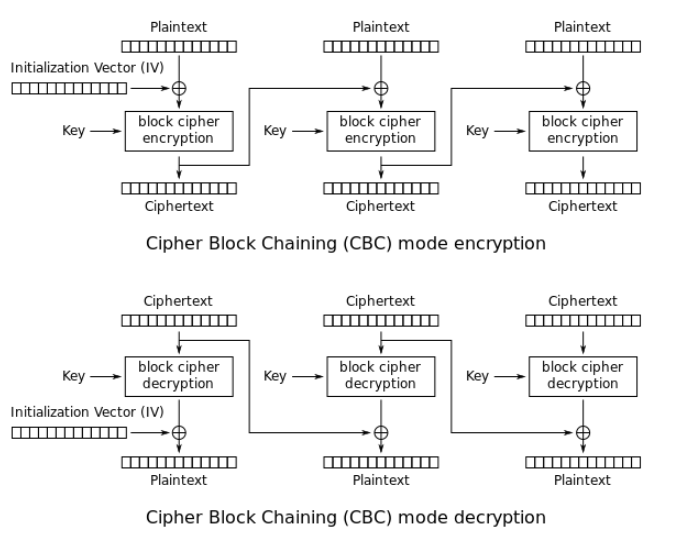
**metin içeren bir resim

Açıklama otomatik olarak oluşturuldu**

**Part B:**

**CBC :**

Cipher block chaining (CBC) is a mode of operation for a [block cipher](https://www.techtarget.com/searchsecurity/definition/block-cipher) -- one in which a sequence of bits are encrypted as a single unit, or block, with a [cipher](https://www.techtarget.com/searchsecurity/definition/cipher) key applied to the entire block. Cipher block chaining uses what is known as an initialization vector ([IV](https://whatis.techtarget.com/definition/initialization-vector-IV)) of a certain length. By using this along with a single [encryption](https://www.techtarget.com/searchsecurity/definition/encryption) key, organizations and individuals can safely encrypt and decrypt large amounts of [plaintext](https://www.techtarget.com/searchsecurity/definition/plaintext).



**Code**:

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

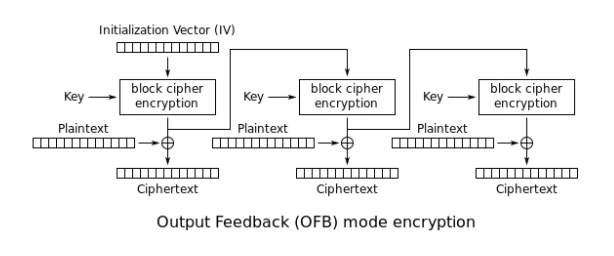
metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**OFB:**

The output feedback (OFB) mode makes a block cipher into a synchronous [stream cipher](https://en.wikipedia.org/wiki/Stream_cipher). It generates [keystream](https://en.wikipedia.org/wiki/Keystream) blocks, which are then [XORed](https://en.wikipedia.org/wiki/XOR) with the plaintext blocks to get the ciphertext. Just as with other stream ciphers, flipping a bit in the ciphertext produces a flipped bit in the plaintext at the same location. This property allows many [error-correcting codes](https://en.wikipedia.org/wiki/Error-correcting_code) to function normally even when applied before encryption.

**OFB** is an [AES](https://www.educative.io/edpresso/what-is-the-aes-algorithm) block cipher mode similar to the [CFB](https://www.educative.io/edpresso/what-is-the-cfb) mode. What mainly differs from CFB is that the OFB mode relies on XOR-ing plaintext and ciphertext blocks with expanded versions of the initialization vector.



Code:

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Test and result :**

**metin içeren bir resim

Açıklama otomatik olarak oluşturuldu**

NOTE: I implemented all the functions in the C programming language, based on the source in Teams. I implemented the cases of reading from the file and giving plaintext directly. I implemented and tested CBC and OFB models.

**Sparkle**

SPARKLE, designed by Beierle et al. comprises the SCHWAEMM family of AEAD cipher and ESCH family of hash functions, both based on the SPARKLE permutation [57]. The permutation applies multiple distinct instances of Alzette, a 4-round 64-bit block cipher, to achieve nonlinearity. Alzette is a 64-bit S-box based on an Addition-Rotation-XOR (ARX) design operating on 32-bit words, making it particularly efficient in software. The SCHWAEMM family of AEAD cipher is based on the duplexed sponge construction with a combined feedback. The ESCH family of hash functions is based on the sponge construction.

Variants. The variants of the SPARKLE family are listed in the table below. Both primary algorithms rely on the 384-bit SPARKLE permutation. In the table, each variant uses the b-bit SPARKLE permutation with r-bit rate and c-bit capacity, where b = r +c. In the # Steps column for AEAD variants, x, y indicates that the SPARKLE permutation with y steps is used (1) in the initialization, (2) between the AD processing and the message processing, and (3) in the finalization; and the SPARKLE permutation with x steps is used in (1) the AD processing and (2) the message processing. In the # Steps column for Hash variants, x, y indicates that the SPARKLE permutation with y steps is used once to generate the first half of the hash output, and the SPARKLE permutation with x steps is used (1) in the absorption phase and (2) to generate the second half of the hash output.

**tablo içeren bir resim

Açıklama otomatik olarak oluşturuldu**

Security Analysis. The security of SPARKLE against various attacks was analyzed by Beierle et al. Work has also shown that the probability of 7-round differential trails of Alzette is at most 2^−24. Liu et al. presented 4-round differential-linear and rotational differential-linear trails in Alzette that improved on previously published differential characteristics. In the designers of SPARKLE reported that the probability of the best 7-round differential trails is 2^-26.

As a result : It is a family of cryptographic permutations based on an ARX design. Its name comes from the block cipher Sparx [DPU+16], which Sparkle is closely related to. Sparkle is basically a Sparx instance with a wider block size and a fixed key, hence its name: SPARx, but Key LEss.

**The Sparkle Permutations**

Our schemes for authenticated encryption and hashing employ the permutation family Sparkle which we specify in the following. In particular, the Sparkle family consists of the permutations Sparkle256𝑛𝑠 , Sparkle384𝑛𝑠 and Sparkle512𝑛𝑠 with block sizes of 256, 384, and 512 bit, respectively. The parameter 𝑛𝑠 refers to the number of steps and a permutation can be defined for any 𝑛𝑠 ∈ N. The permutations are built using the following main components:

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Code:

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Test and Result :

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

NOTE: I worked on Sparkle a lot, but I couldn't understand enough to implement it, I was only able to implement the sparkle permutation. I couldn't find any resources to implement the other Schwaemm algorithm. Searching the internet, I found a working code directory. I coded Sparkle Schwaemm algorithms using these codes.

**Referans:**

**Sparkle :**

<https://asecuritysite.com/light/lw_Sparkle>

<https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/sparkle-spec-round2.pdf#page=86&zoom=100,113,113>

**TinyJAMBU:**

https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/TinyJAMBU-spec.pdf#page=30&zoom=100,178,166