GTU

DEPARTMENT OF COMPUTER ENGINEERING

CSE 470 – Autumn 2022

PROJECT REPORT

SÜLEYMAN GÖLBOL

1801042656

1) ALGORITHM DETAILS

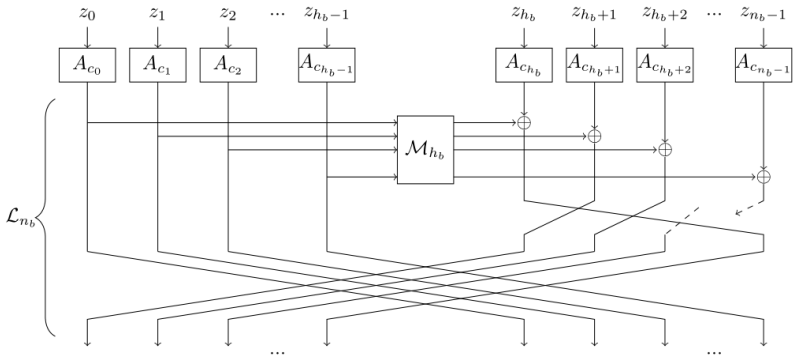
Lightweight cryptography algorithms are the cryptographic algorithms and protocols that are designed to be efficient in terms of computation and memory usage, making them suitable for use in resource-constrained environments such as embedded devices, IoT devices, and other types of systems with limited processing power or memory. **Sparkle** and **TinyJambu** are some of them.

Block ciphers have mode of operations such as ECB, CBC, OFB and CFB.

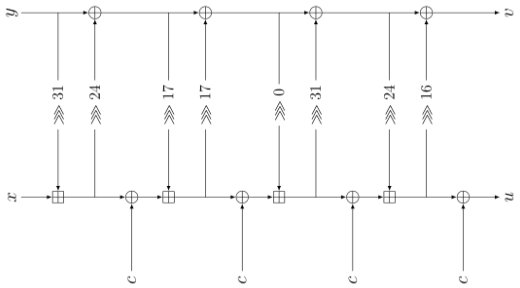
1.1) SPARKLE

Sparkle is actually cryptographic permutation set which depends of key features of Esch and Schwaemm and their efficiency are based on small silicon area and requirements.

It has parametrized implementation that makes it easier to write macro-based code that builds binaries. But also supports all instances. I has protection against side-channel attacks. Since Schwaemm standards doesn't have conditional statements which relies on secret data and timing attacks so it makes this kind of a little concern.



The picture above shows example of a Sparkle step for Sparkle permutations.
Second step of sparkle comes with diffusion layer.



$$\ell(x) = (x \lll 16) \oplus (x \& \texttt{0xffff}) \ ,$$

```
def lfunction(string):
    rotated_16 = rotator(16, string)
    anded = string & bin(11111111111111111111111111111111)
    l_x = rotated_16 ^ anded
    return l_x
```

For diffusion layer, I implanted as l(x) like that.

At the end linear layer implementation becomes like this;

```python
def linear_diffusion_enc(self, array):
    temp1 = array[0]
    temp2 = array[1]

    x = array[0]
    y = array[1]

    for i in range(2,6):
        if i % 2 != 0:
            continue
        else:
            temp1 = temp1 ^ array[i]
            temp2 = temp2 ^ array[i+1]

    temp1 = self.lfunction(temp1)
    temp2 = self.lfunction(temp2)

    for i in range(2,6):
        if i % 2 != 0:
            continue
        else:
            array[i-2] = array[i+6] ^ array[i] ^ temp2
            array[i-1] = array[i+7] ^ array[i+1] ^ temp1
            array[i+6] = array[i]
            array[i+7] = array[i+1]

    xored1 = x ^ temp2
    xored2 = y ^ temp1
    array[4] = array[6] ^ xored1
    array[5] = array[7] ^ xored2
    array[7] = y
    array[6] = x

    return array
```

For Alzette ARX-Box; I implemented like this. (Algorithm 2.4 in paper)

```python
def alzette_arx_box_enc(self, array, key):
    for i in range(0,12):
        if i % 2 != 0: # apply this algorithm only for even indexes
            continue
        else:
            # self.reduncancy_remover = remove_excess
            rc = key[i>>1] # rc = round constant
            added_rotation = array[i] + self.rotator(31, array[i+1])
            array[i] = self.reduncancy_remover(added_rotation)
            array[i+1] = array[i+1] ^ self.rotator(24, array[i])
            array[i] = array[i] ^ rc

            added_rotation = array[i] + self.rotator(17, array[i+1])
            array[i] = self.reduncancy_remover(added_rotation)
            array[i+1] = array[i+1] ^ self.rotator(17, array[i])
            array[i] = array[i] ^ rc

            added_sequence = self.sequence_summer(array, i)
            array[i] = self.reduncancy_remover(added_sequence)
            array[i+1] = array[i+1] ^ self.rotator(31, array[i])
            array[i] = array[i] ^ rc

            added_rotation = array[i] + self.rotator(24, array[i+1])
            array[i] = self.reduncancy_remover(added_rotation)
            array[i+1] = array[i+1] ^ self.rotator(16, array[i])
            array[i] = array[i] ^ rc

    return array
```

I used arx-bx alzette and diffusion layer to encode like this:

```python
def encoder(self, data_to_encrypt, key):
    # this code encrypts data with the 256bits key
    if key == None:
        # print("Key is not given")
        key = 0x432646294A404E635266556A586E3272357538782F413F4428472D4B61506453

    array = self.to_array_converter(data_to_encrypt, 12)

    key_array = self.to_array_converter(key, 8)
    # 12 comes because the data is 384 bits and 384/32 = 12
    # 8 comes because the key is 256 bits and 256/32 = 8

    # ALZETTE DEPENDS ON 32 BIT CHUNKS ARX BOX

    for round_num, round_key in enumerate(key_array):
        array[1] = array[1] ^ round_key
        array[3] = array[3] ^ round_num

        # Alzette
        array = self.alzette_arx_box_enc(array, key_array)
        # Linear diffusion
        array = self.linear_diffusion_enc(array)

    encrypted = self.from_array_converter(array)
    return encrypted
```

It helps to encrypt with sparkle algorithm. If key is not given it creates a random key like above. It converts data to array with each size of 12. Before applying alzette and linear diffusion first we need to XOR array's first and third index with key array's key and index. At the end this returns encoded output.

Also, to remove redundancies I created a function like this to make sure it's 32 bits.

```python
def reduncancy_remover(self, string):
    new_string = string & 0b11111111111111111111111111111111
    return new string
```

To decrypt, similarly I created alzette arx-box and linear diffusion functions for decoding again.

```python
def decoder(self, data_to_decrypt, key):
    # this code decrypts data with the 256bits key

    array = self.to_array_converter(data_to_decrypt, 12)
    if key == None:
        # print("Key is not given")
        key = 0x432646294A404E635266556A586E3272357538782F413F4428472D4B61506453

    key_array = self.to_array_converter(key, 8)
    # 12 comes because the data is 384 bits and 384/32 = 12
    # 8 comes because the key is 256 bits and 256/32 = 8

    # ALZETTE DEPENDS ON 32 BIT CHUNKS ARX BOX
    length = len(key_array)
    for i in reversed(range(0, length)):
        # Linear diffusion
        array = self.linear_diffusion_dec(array)
        # Alzette
        array = self.alzette_arx_box_dec(array, key_array)

        array[1] = array[1] ^ key_array[i]
        array[3] = array[3] ^ i


    decrypted = self.from_array_converter(array)
    return decrypted
```

Then to decode I used similar method but this time I used reversed key array so that we can start from the end to decrypt the data.

At the end I used from_Array_converter() function to convert array back.

```python
def from_array_converter(self, array):
    # converted = 0b00000000000000000000000000000000
    converted = 0x0
    for i in range(len(array)):
        converted = converted << 32
        converted = converted | array[i]
    return converted
```

MODES

For the modes I created 4 different function. ECB, OFB, CBC, CFB.

```python
def ecb_mode_encoding(chunks, algorithm, key):
    encrypted_chunks = []
    for chunk in chunks:
        # encoded = algorithm.encoder(chunk, key)
        encoded = algorithm(chunk, key)
        encrypted_chunks.append(encoded)
    return encrypted_chunks
```

Ecb was easiest (but most unsecure one also)

```python
def cbc(chunks, algorithm, key, is_decoding=True):
    initialization_vector = 0x59161abcdef16195
    handled_chunks = []
    if is_decoding == False:  # Encoding
        for chunk in chunks:
            chunk = chunk ^ initialization_vector # XOR with IV
            decoded = algorithm(chunk, key)
            handled_chunks.append(decoded)
            initialization_vector = handled_chunks[-1] # -1 is the last element
    else:   # Decoding
        for chunk in chunks:
            chunk = chunk ^ initialization_vector # XOR with IV
            encoded = algorithm(chunk, key)
            handled_chunks.append(encoded)
            handled_chunks[-1] = handled_chunks[-1] ^ initialization_vector # XO

    return handled_chunks
```

For cbc I created this function that takes chunks aray and algorithm to appy with an initialization vector (IV). For encoding it makes initialization vector as handled chunks (blocks) last element.

```python
def cfb(chunks, algorithm, key, is_decoding=True):
    initialization_vector = 0x59161abcdef16195
    handled_chunks = []
    if is_decoding == False:  # Encoding
        for i in range(len(chunks)):
            encoded = algorithm(initialization_vector, key)
            handled_chunks.append(encoded)
            handled_chunks[len(handled_chunks)-1] = handled_chunks[len(handled_chunks)-1] ^ chunks[i]
            initialization_vector = handled_chunks[len(handled_chunks)-1]
    else:  # Decoding
        for i in range(len(chunks)):
            encoded = algorithm(initialization_vector, key)
            handled_chunks.append(encoded)
            handled_chunks[len(handled_chunks)-1] = handled_chunks[len(handled_chunks)-1] ^ chunks[i]
            initialization_vector = chunks[i]

    return handled_chunks
```

For CFB, I used a similar method but this time I needed to xor with blocks iTh element to gry yjr last element of handled chunks(blocks).

```python
def ofb(chunks, algorithm, key, is_decoding=True):
    initialization_vector = 0x59161abcdef16195
    handled_chunks = []
    if is_decoding == False or is_decoding == True:  # Doesn't matter
        for i in range(len(chunks)):
            encoded = algorithm(initialization_vector, key)
            handled_chunks.append(encoded)
            initialization_vector = handled_chunks[len(handled_chunks)-1]
            handled_chunks[len(handled_chunks)-1] = handled_chunks[len(handled_chunks)-1] ^ chunks[i]
    return handled_chunks
```

For OFB, encode or decode it didn’t matter so I applied algorithm like that and xor’ed last element to get the new last element. Here algorithm refers for sparkle’s function.

```python
def write_as_bytes(chunks, needs_unpadding, file_path):
    pad = 0
    if needs_unpadding == True:
        # Unpadding because of the padding
        # chunks[len(chunks)-1] & 0b11111111 is checksum
        if (chunks[len(chunks)-1] & 0b11111111) < 2:
            pad = 0
        elif (chunks[len(chunks)-1] & 0b11111111) > 49:
            pad = 0
        elif (chunks[len(chunks)-1] & 0b11111111) == 49:
            if (chunks[len(chunks)-2] & 0b11111111) == 0\
            and (chunks[len(chunks)-1] & 0b11111111) == 49:
                chunks = chunks[:-1]
                pad = 1
            else:
                pad = 0
        # check if the last byte is a padding byte using the mask 2 ** (8 * (last_byte & 0b11111111)) - 1
        elif (chunks[len(chunks)-1] & (2**(8 * (chunks[len(chunks)-1] & 0b11111111))-1)) == (chunks[len(chunks)-1] & 0b11111111):
            pad = chunks[len(chunks)-1] & 0b11111111
        else:
            pad = 0
    with open(file_path, 'wb') as file:
        block_length = 48
        for chunk in chunks:
            if needs_unpadding == True:
                if chunk is chunks[len(chunks)-1]: # last chunk
                    block_length = block_length - pad
                    chunk >>=  pad * 8
            to_write = chunk.to_bytes(block_length, byteorder='big')
            file.write(to_write)
```

For writing to file as bytes, I created this function. This ANDs last block element with 0xFF to get the value so that it can find the padding to be added. After opening output file, it checks whether it needs unpadding or not. Because if it needs unpadd,ng, it subtracts pad from block length to find new block length. Then it converts blocks to bytes and writes to file.

```python
def read_as_bytes(block_length, file_path, ignore_last=False):
    readed = []

    with open(file_path, 'rb') as file:
        block = file.read(block_length)
        while block is not None and block != b'':
            integer_represented = int.from_bytes(block, byteorder='big')
            readed.append(integer_represented)
            block = file.read(block_length)

    if ignore_last==True:
        readed = readed[:len(readed)-1]
        if getsize(file_path) % block_length == 0:
            pass
        else:
            last_block = readed[-1]
            readed[-1] = last_block >> ((block_length - (getsize(file_path) % block_length)) * 8)

    if getsize(file_path) % block_length == 0:
        pass # no problem
    else:
        # Padding
        # print("The file is not a multiple of 48 bytes so the last block will be padded with zeros")
        last_block = readed[-1]
        last_block = last_block << (block_length - (getsize(file_path) % block_length)) * 8
        pad = []
        if (getsize(file_path) % block_length) + 1 < block_length:
            last_block = last_block ^ (block_length - (getsize(file_path) % block_length))
        pad.append(last_block)
        if (getsize(file_path) % block_length) + 1 >= block_length:
            pad.append(-(getsize(file_path) % block_length) + block_length * 2)
        readed = readed[:len(readed)-1] + pad
    return readed
```

To read bytes, I created read_as_bytes() function. This one reads blocks every time as block_length and to get the integer representation uses int.from bytes.

But if file is not a multiple of block_length (for example 48), it appends padding.

To do this, it shifts last block to left by the size of 8*checksum subtracted from block length. So this way, it calculates the pad to be appended.

At the end function returns readed data as bytes.

Checking changes using mode as hash

```
mode = take_input("Select mode: 1 for ECB, 2 for OFB, 3 for CFB, 4 for CBC: ", int_type=True)
input1 = take_input("Enter the input file path: ")
input2 = take_input("Enter the comparison file path: ")

input_blocks = read_as_bytes(block_length=48, file_path=input1)
compared = read_as_bytes(block_length=48, file_path=input2)

output = get_output_by_mode(mode, input_blocks, sparkle, is_encoder=True)

output = xor(output)
if output[0] == compared[0]:
    print("Files are using same hash.")
else:
    print("Files are using different hash.")
```

It takes input file 1 and input file 2 so that xoring output comes from a mode will give the results if the hashes are same or not.

Sign input with key into output

```
input_blocks = read_as_bytes(block_length=48, file_path=input_file)
key = read_as_bytes(block_length=32, file_path=key)[0]
if mode == "2":
    output = ofb(input_blocks, sparkle.encoder, key)
elif mode == "3":
    output = cfb(input_blocks, sparkle.encoder, key)
elif mode == "4":
    output = cbc(input_blocks, sparkle.encoder, key)
else:
    output = ecb_mode_encoding(input_blocks, sparkle.encoder, key)
output = xor(output)
with open(input_file, "wb") as f:
    f.write(output[0].to_bytes(48, byteorder='big'))
print("Done writing/signing hash to file.")
```

To sign input to output using I used this. This xor's output and writes the output's 0^{th} index to bytes.

Integrity Validation

```
input_blocks = read_as_bytes(block_length=48, file_path=input_file, ignore_last=True)
key = read_as_bytes(block_length=32, file_path=key)[0]
if mode == "2":
    output = ofb(input_blocks, sparkle.encoder, key)
elif mode == "3":
    output = cfb(input_blocks, sparkle.encoder, key)
elif mode == "4":
    output = cbc(input_blocks, sparkle.encoder, key)
else:
    output = ecb_mode_encoding(input_blocks, sparkle.encoder, key)
output = xor(output)
try:
    with open(input_file, "wb") as f:
        f.seek(-48, 2)
        comp = int.from_bytes(f.read(48), byteorder='big')
except:
    print("Integrity is valid.") # if file isn't signed, it's valid
    return
if output[0] == comp:
    print("Integrity is valid.")
else:
    print("Integrity is not valid.")
```

For integrity validation. It seeks file offset by -48 bits s that it can read last 48 bytes to see if integrity is valid by comparing output and comp.

Hash Generation

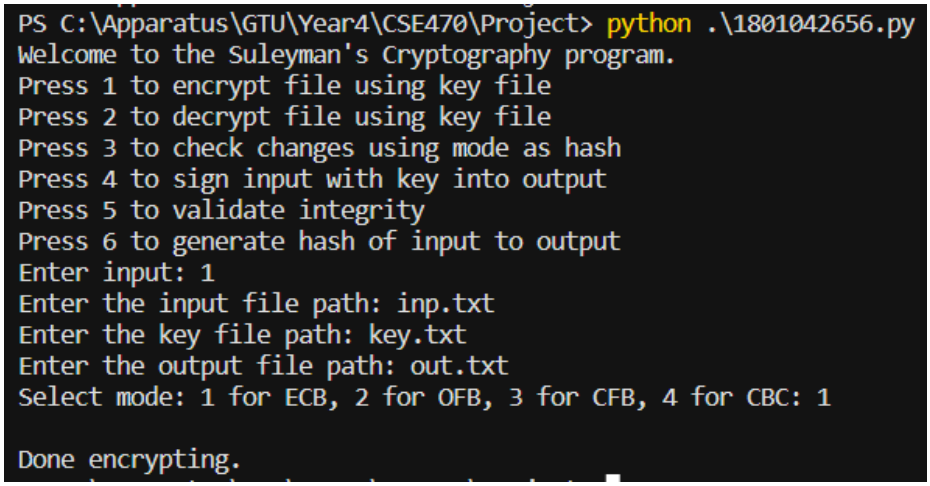
```
input_blocks = read_as_bytes(block_length=48, file_path=input_file)
if mode == "2":
    output = ofb(input_blocks, sparkle.encoder, key=None)
elif mode == "3":
    output = cfb(input_blocks, sparkle.encoder, key=None)
elif mode == "4":
    output = cbc(input_blocks, sparkle.encoder, key=None)
else:
    output = ecb_mode_encoding(input_blocks, sparkle.encoder, key=None)
output = xor(output)
write_as_bytes(output, needs_unpadding=False, file_path=output_file)
print("Done generating hash to file. Result is: ", hex(output[0]))
```

As it seen above, to generate hash, it takes mode and write as bytes and prints hash in hex format.

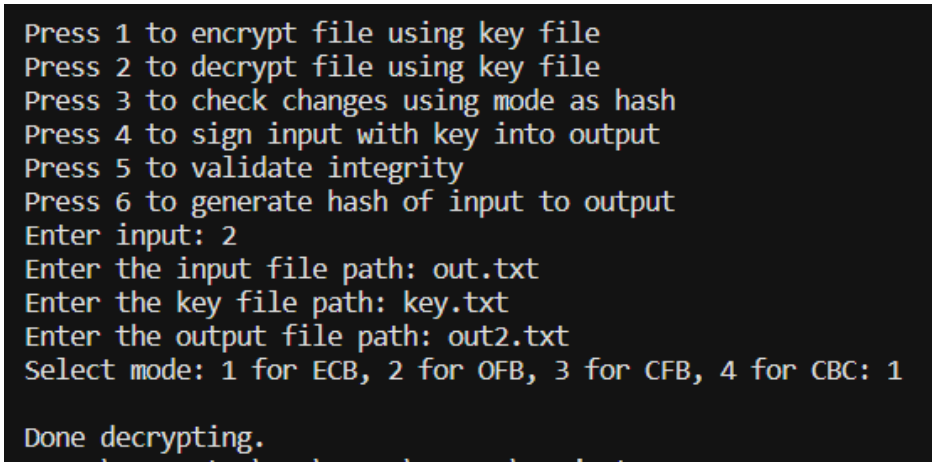
Usage Example:

For Sparkle

Example Encryption



Example Decryption



Example change checker using a mode(ECB,OFB,CFB or CBC) as hash

```
PS C:\Apparatus\GTU\Year4\CSE470\Project> python .\1801042656.py
Welcome to the Suleyman's Cryptography program.
Press 1 to encrypt file using key file
Press 2 to decrypt file using key file
Press 3 to check changes using mode as hash
Press 4 to sign input with key into output
Press 5 to validate integrity
Press 6 to generate hash of input to output
Enter input: 3
Select mode: 1 for ECB, 2 for OFB, 3 for CFB, 4 for CBC: 3
Enter the input file path: inp.txt
Enter the comparison file path: out.txt
Files are using different hash.
```

Example file signer with key to output

```
PS C:\Apparatus\GTU\Year4\CSE470\Project> python .\1801042656.py
Welcome to the Suleyman's Cryptography program.
Press 1 to encrypt file using key file
Press 2 to decrypt file using key file
Press 3 to check changes using mode as hash
Press 4 to sign input with key into output
Press 5 to validate integrity
Press 6 to generate hash of input to output
Enter input: 4
Enter the input file path: inp.txt
Enter the key file path: key.txt
Enter the output file path: out.txt
Select mode: 1 for ECB, 2 for OFB, 3 for CFB, 4 for CBC: 4
Done writing/signing hash to file.
```

Example integrity validator

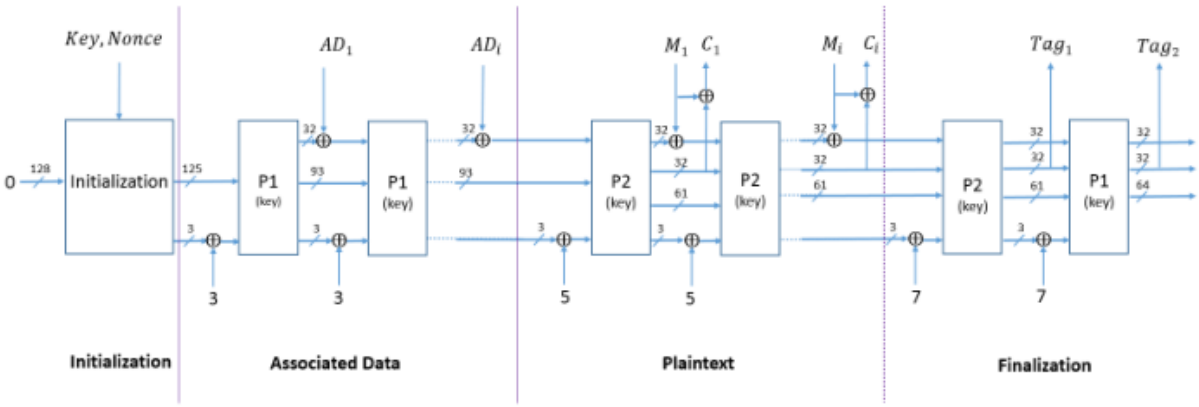
```
PS C:\Apparatus\GTU\Year4\CSE470\Project> python .\1801042656.py
Welcome to the Suleyman's Cryptography program.
Press 1 to encrypt file using key file
Press 2 to decrypt file using key file
Press 3 to check changes using mode as hash
Press 4 to sign input with key into output
Press 5 to validate integrity
Press 6 to generate hash of input to output
Enter input: 5
Enter the input file path: inp.txt
Enter the key file path: key.txt
Select mode: 1 for ECB, 2 for OFB, 3 for CFB, 4 for CBC: 4
Integrity is valid.
```

Example hash generator

```
PS C:\Apparatus\GTU\Year4\CSE470\Project> python 1801042656.py
Welcome to the Suleyman's Cryptography program.
Press 1 to encrypt file using key file
Press 2 to decrypt file using key file
Press 3 to check changes using mode as hash
Press 4 to sign input with key into output
Press 5 to validate integrity
Press 6 to generate hash of input to output
Enter input: 6
Enter the input file path: new1801042656.c
Enter the key file path: out2.txt
Enter the output file path: out2.txt
Select mode: 1 for ECB, 2 for OFB, 3 for CFB, 4 for CBC: 3
Done generating hash to file. Result is:  0xd8451a533f24c42a24af297aac9fffc07
4cb7f42d350052d94819f823323ea83ff7f3d3afb82a3f05044b5b32a7c665f
```

1.2) TINYJAMBU

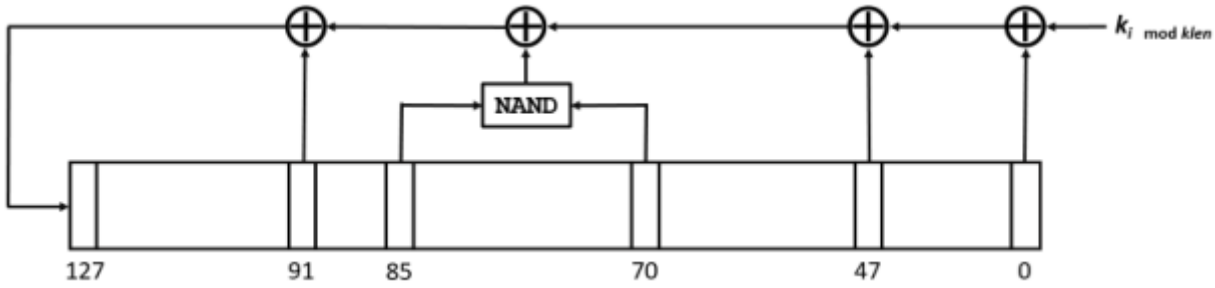
TinyJambu is actually another mode of Jambu which is a lightweight authenticated encryption mode. It's authenticated and base on keyed permutation. I tried to implement TinyJambu but had some errors while implementing so I didn't include in source code.



For 128 bits, it uses keyed-permutations 128 bits and and 32 bits block sizes.

It totally supports 2 keys sizes.

- Primary member: TinyJAMBU-128
  128-bit key, 96-bit nonce, 64-bit tag, 128-bit state
- TinyJAMBU-192
  192-bit key, 96-bit nonce, 64-bit tag, 128-bit state
- TinyJAMBU-256
  256-bit key, 96-bit nonce, 64-bit tag, 128-bit state



For keyed permutation, this is the nonlinear feedback register.

```
StateUpdate(S, K, i):
    feedback = s0 ⊕ s47 ⊕ (~ (s70&s85)) ⊕ s91 ⊕ ki mod klen
    for j from 0 to 126: sj = sj+1
    s127 = feedback
end
```

To update states, it uses this logic (for 128 bits).

For setting up; it consists of 2 setups. Key setup and nonce setup.

For key setup; it sets the 128-bit state S as 0. Then updates the state using P1024.

For nonce setup it has three steps. In each step frame bits are xor'ed with state so that keyed permutation P_{640} will be updated.

After the setup part, data needs to be processed so for that in each step, the Frame bits of associated data are Xor'ed with the state, then it updates the state using the keyed

permutation P640, then 32 bits of the associated data are Xor'ed with the state.

```
if (adlen mod 32) > 0:
  s{36···38} = s{36···38} ⊕ FrameBits{0···2}
  Update the state using P640
  lenp = adlen mod 32 /* number of bits in the partial block /
  startp = adlen − lenp / starting position of the partial block /
  s{96···96+lenp−1} = s{96···96+lenp−1} ⊕ ad{startp···adlen−1}
  s{32···33} = s{32···33} ⊕ (lenp/8)
end if
```

After the processing part, plaintexts needs to be encrypted. For that

```
for i from 0 to [mlen/32]:
  s{36···38} = s{36···38} ⊕ FrameBits{0···2}
  Update the state using P_1024
  s{96···127} = s{96···127} ⊕ m{32i···32i+31}
  c{32i···32i+31} = s{64···95} ⊕ m{32i···32i+31}
end for
```

This can be used.

But sometimes the last block may not be a full block. (Because of no padding. Or not a multiple of block size)

In that case last block chunk needs to be Xor'ed to the state and byte number also needs to be Xor'ed with the state.

For decryption part, it's similar with encryption. (For the initialization and processing parts.) At the end, we need to decrypt the ciphertext. Every time plaintext frame bits are xor'ed with state so that we can update the state with keyed permutation.

```
for i from 0 to bmlen/32c:
  s{36···38} = s{36···38} ⊕ F rameBits{0···2}
  Update the state using P1024
  m{32i···32i+31} = s{64···95} ⊕ c{32i···32i+31}
  s{96···127} = s{96···127} ⊕ m{32i···32i+31}
end for
```

For security, each pair of key/nonce is just to protect only 1 message. So, when verification fails, decrypted plaintext won't be given as output. Also, for unprotected decryption when attacker has physical access to device that makes the decryption; this is the security goals of TinyJambu.

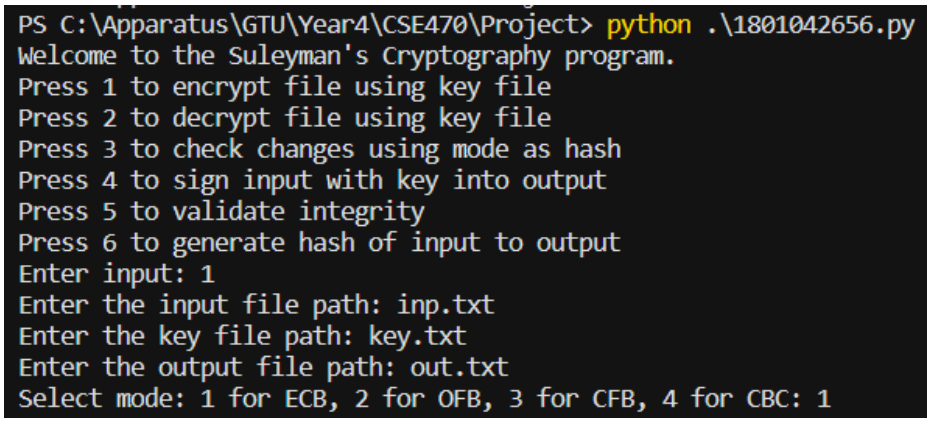
Table 5.3: Security Goals of TinyJAMBU for Unprotected Decryption

	Secret Key	Authentication	Max. Forgery Adv.
TinyJAMBU-128	112-bit	64-bit	2^{-15}
TinyJAMBU-192	168-bit	64-bit	2^{-15}
TinyJAMBU-256	224-bit	64-bit	2^{-15}

TinyJAMBU Mode in an authenticated encryption mode, the probability of state collision plays the role for protecting the confidentiality of plaintext and for resisting the forgery attack on the message.

Example image from app:

To run the app; use ‘python 1801042656.py’



Bibliography

1.

TinyJAMBU: A Family of Lightweight Authenticated Encryption Algorithms, Hongjun Wu and Tao Huang, Nanyang Technological University, May 2021

2.

Schwaemm and Esch: Lightweight Authenticated Encryption and Hashing using the Sparkle Permutation Family, Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großsch¨adl1, Amir Moradi, L´eo Perrin, Aein Rezae Shahmirzadi, Aleksei Udovenko, Vesselin Velichkov and Qingju Wang