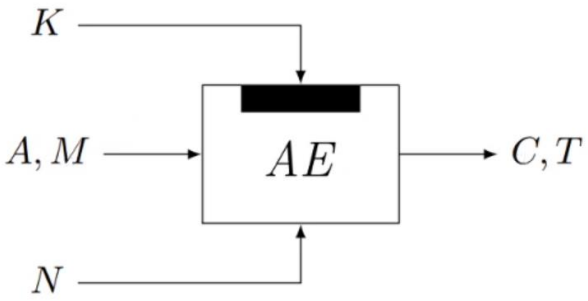
CSE 470 Cryptography
Project Report

Melihcan Çilek
1801042092

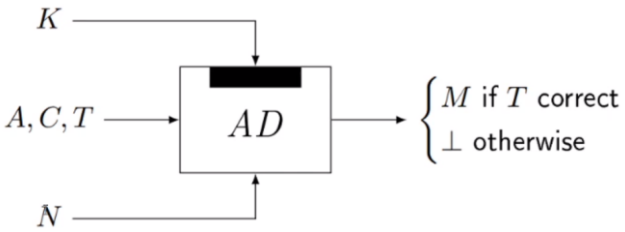
Algorithms
Elephant and Xoodyak

AE (Authenticated Encryption)

Authenticated Encryption with Associated are forms of encryption which simultaneously assure the confidentiality and authenticity of data.



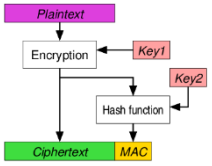
- Ciphertext C encryption of message M
- Tag T authenticates associated data A and message M
- Nonce N randomizes the scheme

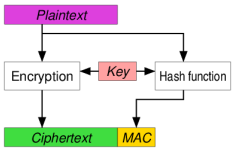


- Authenticated decryption needs to satisfy that
  - Message disclosed if tag is correct
  - Message is not leaked if tag is incorrect
- Correctness: $AD_k(N, A, AE_k(N, A, M)) = M$

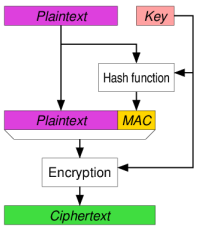
AEAD (Authenticated Encryption with Associated Data)

AEAD is a variant of Authenticated Encryption that allows a recipient to check the integrity of both the encrypted and unencrypted information in a message. AEAD binds associated data (AD) to the ciphertext and to the context where it is supposed to appear so that attempts to "cut-and-paste" a valid ciphertext into a different context are detected and rejected. There are three approaches to be used in authenticated encryption. These are:

- Encrypt-then-MAC  (EtM)

- Encrypt-and-MAC 

(E&M)

- MAC-then-Encrypt 

(MtE)

Overview

The National Institute of Standards and Technology (NIST) is an agency of the United States Department of Commerce whose mission is to promote American innovation and industrial competitiveness.

NIST has initiated a process to solicit, evaluate, and standardize lightweight cryptographic algorithms that are suitable for use in constrained environments where the performance of current NIST cryptographic standards is not acceptable.

In August 2018, NIST published a call for algorithms (test vector generation code) to be considered for lightweight cryptographic standards with authenticated encryption with associated data (AEAD) and optional hashing functionalities.

The deadline for submitting algorithms has passed. NIST received 57 submissions to be considered for standardization. After the initial review of the submissions, 56 were selected as Round 1 candidates. Of the 56 Round 1 candidates, 32 were selected to advance to Round 2. In March 2021, NIST announced ten finalists as ASCON, Elephant, GIFT-COFB, Grain128-AEAD, ISAP, Photon-Beetle, Romulus, Sparkle, TinyJambu, and Xoodyak. In October 2022, NIST posted the final status updates to the finalists page.

Elephant

Elephant is a family of lightweight authenticated encryption schemes and one of ten finalists in the NIST lightweight cryptography project.

The goal of the Elephant, minimize state size and complexity of design while still meeting expected security strength 2^{112} and limit on online complexity 2^{50} bytes as prescribed by the NIST call for proposals.

The first decision they made was which cryptographic primitive going to be used. There are many popular primitives such as Tweakable Block Ciphers, Block Ciphers etc. and all of them have their differences. The primitive they used is Permutation

which is similar to the traffic sign on the right. 

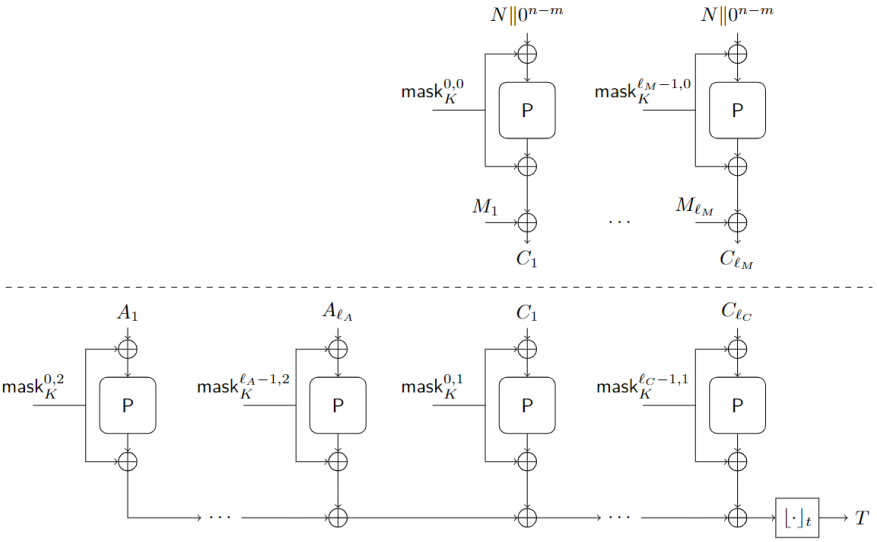
Second decision they made was which cryptographic mode to use. Established approach to use was Keyed duplex/sponge [BDPV11, MRV15, DMV17] with inherently sequential. Their approach is,

- Parallel evaluation of the permutation which requires proper masking
- Evaluating it in forward direction only which requires proper mode of use
- Goal was to minimize permutation size

Third decision they made was which Mask to use. In this, simplified version of Masked-Even-Mansou (MEM) is used [GJMN16].

- φ_1 is fixed LFSR, $\varphi_2 = \varphi_1 \oplus \mathsf{id}$ where LFSR is Linear Feedback Shift Register which means a shift register whose input bit is a linear function of its previous state.
- $\mathsf{mask}_K^{a,b} = \varphi_2^b \circ \varphi_1^a \circ \mathsf{P}(K \| 0^{n-k})$

  Advantages of this masking are executed in constant-time meaning more efficient compared to the alternatives and simple to implement.



On the previous page at the top, we can see the Encryption part. Encryption gets input as Nonce (N) and appended with 0's and fed to all permutation calls without a counter, yet. The counter is in the mask just like the key. a and b go as (0,0), (1,0) ... ($\ell_M - 1$, 0) where ℓ_M is the number of message blocks.

Encryption

- Nonce N input to all P calls
- K and counter in mask
- Padding $M_1 \dots M_{\ell_M} \xleftarrow{n} M$
- Ciphertext $C \leftarrow \lfloor C_1 \dots C_{\ell_M} \rfloor_{|M|}$

Algorithm

Algorithm 1 Elephant encryption algorithm enc

Input: $(K, N, A, M) \in \{0,1\}^k \times \{0,1\}^m \times \{0,1\}^* \times \{0,1\}^*$
Output: $(C, T) \in \{0,1\}^{|M|} \times \{0,1\}^t$

1: $M_1 \dots M_{\ell_M} \xleftarrow{n} M$
2: **for** $i = 1, \dots, \ell_M$ **do**
3: $\quad C_i \leftarrow M_i \oplus \mathsf{P}(N \| 0^{n-m} \oplus \mathsf{mask}_K^{i-1,0}) \oplus \mathsf{mask}_K^{i-1,0}$
4: $C \leftarrow \lfloor C_1 \dots C_{\ell_M} \rfloor_{|M|}$
5: $T = 0$
6: $A_1 \dots A_{\ell_A} \xleftarrow{n} N \| A \| 1$
7: $C_1 \dots C_{\ell_C} \xleftarrow{n} C \| 1$
8: **for** $i = 1, \dots, \ell_A$ **do**
9: $\quad T \leftarrow T \oplus \mathsf{P}(A_i \oplus \mathsf{mask}_K^{i-1,2}) \oplus \mathsf{mask}_K^{i-1,2}$
10: **for** $i = 1, \dots, \ell_C$ **do**
11: $\quad T \leftarrow T \oplus \mathsf{P}(C_i \oplus \mathsf{mask}_K^{i-1,1}) \oplus \mathsf{mask}_K^{i-1,1}$
12: **return** $(C, \lfloor T \rfloor_t)$

On the previous page at the bottom, we can see the Authentication part. We get the Nonce and the Associated Data which is a part that not encrypted but will only be authenticated, then concatenate with 1 and padded into integral number of n bit blocks $A_1 \dots A_{\ell_A}$ and in cyphertext part, cyphertext taken that is coming from encryption, appended with 1 and padded into integral number of n bit cyphertext $C_1 \dots C_{\ell_C}$ blocks. Now, the inputs are fed through a permutation with a mask.

Authentication

- Padding $A_1 \dots A_{\ell_A} \xleftarrow{n} N \| A \| 1$
- Padding $C_1 \dots C_{\ell_C} \xleftarrow{n} C \| 1$
- K and counter in mask
- Tag T truncated to t bits

Algorithm 2 Elephant decryption algorithm dec

Input: $(K, N, A, C, T) \in \{0,1\}^k \times \{0,1\}^m \times \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^t$
Output: $M \in \{0,1\}^{|C|}$ or $\perp$

1: $C_1 \dots C_{\ell_M} \xleftarrow{n} C$
2: **for** $i = 1, \dots, \ell_M$ **do**
3: $\quad M_i \leftarrow C_i \oplus \mathsf{P}(N \| 0^{n-m} \oplus \mathsf{mask}_K^{i-1,0}) \oplus \mathsf{mask}_K^{i-1,0}$
4: $M \leftarrow \lfloor M_1 \dots M_{\ell_M} \rfloor_{|C|}$
5: $\bar{T} = 0$
6: $A_1 \dots A_{\ell_A} \xleftarrow{n} N \| A \| 1$
7: $C_1 \dots C_{\ell_C} \xleftarrow{n} C \| 1$
8: **for** $i = 1, \dots, \ell_A$ **do**
9: $\quad \bar{T} \leftarrow \bar{T} \oplus \mathsf{P}(A_i \oplus \mathsf{mask}_K^{i-1,2}) \oplus \mathsf{mask}_K^{i-1,2}$
10: **for** $i = 1, \dots, \ell_C$ **do**
11: $\quad \bar{T} \leftarrow \bar{T} \oplus \mathsf{P}(C_i \oplus \mathsf{mask}_K^{i-1,1}) \oplus \mathsf{mask}_K^{i-1,1}$
12: **return** $\lfloor \bar{T} \rfloor_t = T \ ? \ M \ : \ \perp$

As we can see, the b values changing in case of the main mode of the encryption, decryption and authentication concepts. For encryption part, b values are 0; for associated data, b values are 2 and for cyphertext, b values are 1.

Mode Properties

- Encrypt-then-MAC
  - CTR encryption
  - Wegman-Carter-Shoup
- Fully parallelizable
- Uses single primitive P
- P in forward direction only

Masks can easily be updated as seen on the right side, evaluation φ_1 to previous mask.

Mask Properties

- Mask can be easily updated
- $\text{mask}_K^{i,0} = \varphi_1 \circ \text{mask}_K^{i-1,0}$
- $\text{mask}_K^{i-1,0} \oplus \text{mask}_K^{i-1,1} = \text{mask}_K^{i,0}$

Security of Elephant

$$\mathbf{Adv}^{\text{ae}}_{\text{Elephant}}(\mathcal{A}) \lesssim \frac{4\sigma p}{2^n}$$

where σ shows online complexity which counts number of associated data and number of text blocks, p shows offline complexity which calls number of primitive calls.

This proof is given under the assumption that P is a random permutation φ_1 has maximal length and $\varphi_2{}^b \circ \varphi_1{}^a \neq \varphi_2{}^{b'} \circ \varphi_1{}^{a'}$ for $(a, b) \neq (a', b')$ and A is nonce-based adversary.

This proof shows that the parameters of NIST lightweight call can be met with a 160-bit permutation.

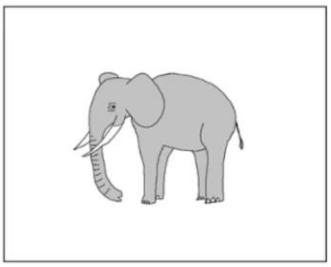
Instances of Elephant:

instance	k	m	n	t	P	φ_1	expected security strength	limit on online complexity
Dumbo	128	96	160	64	80-round Spongent-π[160]	φ_{Dumbo}	2^{112}	$2^{50}/(n/8)$
Jumbo	128	96	176	64	90-round Spongent-π[176]	φ_{Jumbo}	2^{127}	$2^{50}/(n/8)$
Delirium	128	96	200	128	18-round Keccak-f[200]	$\varphi_{\text{Delirium}}$	2^{127}	$2^{74}/(n/8)$

- All LFSRs operate on 8-bit words:

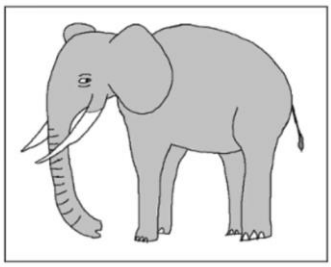
$$\varphi_{\text{Dumbo}} : (x_0, \ldots, x_{19}) \mapsto (x_1, \ldots, x_{19}, x_0 \lll 3 \oplus x_3 \ll 7 \oplus x_{13} \gg 7)$$
$$\varphi_{\text{Jumbo}} : (x_0, \ldots, x_{21}) \mapsto (x_1, \ldots, x_{21}, x_0 \lll 1 \oplus x_3 \ll 7 \oplus x_{19} \gg 7)$$
$$\varphi_{\text{Delirium}} : (x_0, \ldots, x_{24}) \mapsto (x_1, \ldots, x_{24}, x_0 \lll 1 \oplus x_2 \lll 1 \oplus x_{13} \ll 1)$$

where *k* shows the number of bits of the key, *m* is the number of bits for Nonce's, *n* is the permutation size, *t* is the size of the tag, P is the type of primitive that is used. The most important part is that for each instance of Elephant, different LFSR's used for three different functions. Reason for that is, to fit to the permutation that is used.



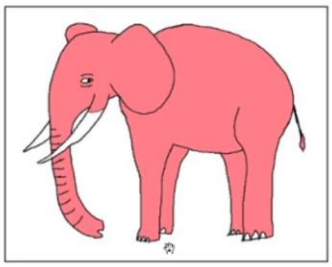
Dumbo

- Spongent-$\pi[160]$
- Minimalist design
  - Time complexity 2^{112}
  - Data complexity 2^{46}



Jumbo

- Spongent-$\pi[176]$
- Conservative design
  - Time complexity 2^{127}
  - Data complexity 2^{46}
- ISO/IEC standardized



Delirium

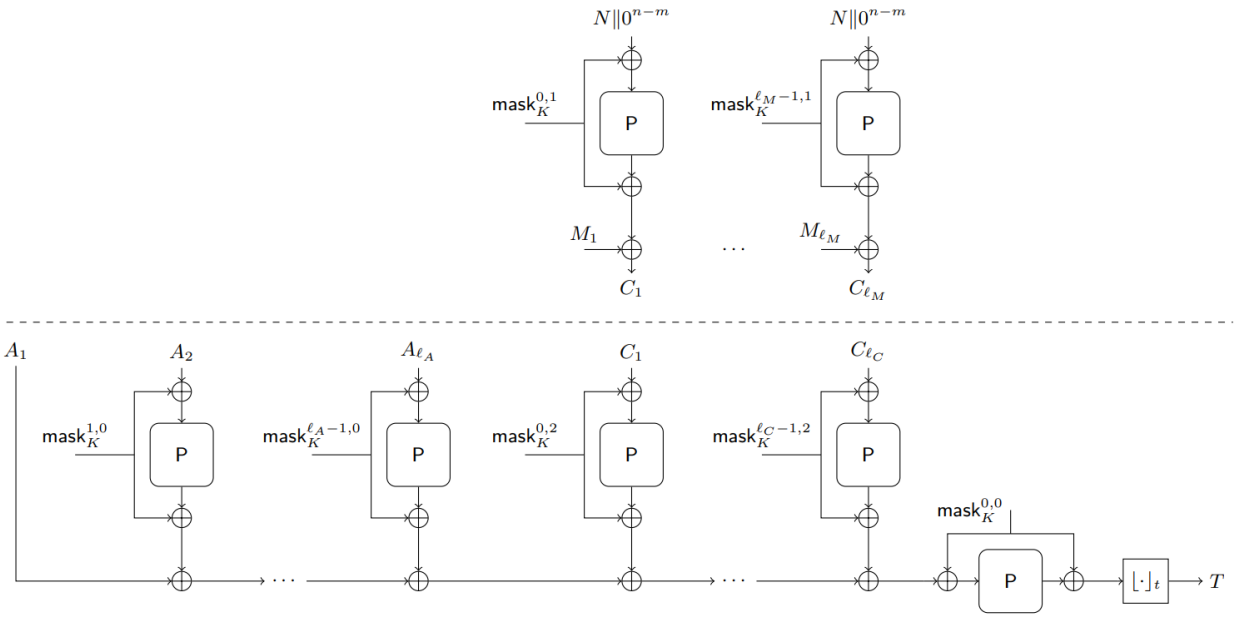
- Keccak-$f[200]$
- High security
  - Time complexity 2^{127}
  - Data complexity 2^{70}
- Specified in NIST standard

Elephant v2

The Elephant v2 mode is permutation-based, only evaluates this permutation in the forward direction, and it is highly parallelizable. Whereas permutations are a popular approach in lightweight cryptography (6 finalists are permutation based), Elephant v2 is unique in its parallelizability (it is the only finalist satisfying this property). Due to this parallelizability, there is no need for large permutations, we can go as small as 160-bit permutations while still matching the security goals recommended by the NIST lightweight call. In detail, the Elephant family consists of three members:

1. Dumbo: Elephant-Spongent-π [160]
    a. This instance achieves 112-bit security provided that the online complexity is at most around 2^{46} blocks. This instance is particularly well-suited for hardware, as Spongent itself is.

2. Jumbo: Elephant-Spongent- π [176]
    a. This is a slightly more conservative instance of Elephant and achieves 127-bit security under the same conditions on the online complexity. In particular, Spongent- π [176] is ISO/IEC standardized (ISO/IEC is the world's best known standard for information security management systems (ISMS) and their requirements).
3. Delirium: Elephant-Keccak-f [200]
    a. This variant performs well in both software and hardware. It also achieves 127-bit security, with a higher bound of around 2^{70} blocks on the online complexity. The permutation is the smallest instance that is specified in the NIST SHA-3 standard that fits our needs.



The updated specification included a security proof of Elephant v2. In detail, in this specification, the following is proven about the mode of Elephant v2 (in the ideal permutation model):

- confidentiality in the nonce-respecting model
- authenticity in the nonce-respecting model
- authenticity in the nonce-misuse model

In a recent work, multi-user security of Elephant v2 has been analyzed, and it was demonstrated that above three security properties also hold in the multi-user setting (up to a logical and negligible security loss).

In addition, there is a tweak proposal done which is corresponds to these two changes,

1. Permutation for the first associated data property A_1 is removed and moved to the end. The importance of this that Wegman-Carter-Shoup authenticators translated to their own setting called “Protected Counter Sum Authenticator” which gives better security.
2. Slight change in roles of mask parameters which is known ‘b’ values in the mask. Now for encryption, b is 1, for cyphertext, b is 2 and for associative data, b is 0. It is just done for aesthetic reasons.

	Elephant v1.1		Elephant v2	
security	confidentiality	authenticity	confidentiality	authenticity
nonce-respecting	✓	✓	✓	✓
nonce-misuse	✗	✗	✗	✓

Updated Algorithm

to their own setting called “Protected Counter Sum Authenticator” which gives better security.

Algorithm 1 Elephant encryption algorithm enc

Input: $(K, N, A, M) \in \{0,1\}^k \times \{0,1\}^m \times \{0,1\}^* \times \{0,1\}^*$
Output: $(C, T) \in \{0,1\}^{|M|} \times \{0,1\}^t$

$M_1 \dots M_{\ell_M} \xleftarrow{n} M$
for $i = 1, \dots, \ell_M$ **do**
  $C_i \leftarrow M_i \oplus \mathsf{P}(N \| 0^{n-m} \oplus \mathsf{mask}_K^{i-1,1}) \oplus \mathsf{mask}_K^{i-1,1}$
$C \leftarrow \lfloor C_1 \dots C_{\ell_M} \rfloor_{|M|}$
$A_1 \dots A_{\ell_A} \xleftarrow{n} N \| A \| 1$
$C_1 \dots C_{\ell_C} \xleftarrow{n} C \| 1$
$T \leftarrow A_1$
for $i = 2, \dots, \ell_A$ **do**
  $T \leftarrow T \oplus \mathsf{P}(A_i \oplus \mathsf{mask}_K^{i-1,0}) \oplus \mathsf{mask}_K^{i-1,0}$
for $i = 1, \dots, \ell_C$ **do**
  $T \leftarrow T \oplus \mathsf{P}(C_i \oplus \mathsf{mask}_K^{i-1,2}) \oplus \mathsf{mask}_K^{i-1,2}$
$T \leftarrow \mathsf{P}(T \oplus \mathsf{mask}_K^{0,0}) \oplus \mathsf{mask}_K^{0,0}$
return $(C, \lfloor T \rfloor_t)$

Algorithm 2 Elephant decryption algorithm dec

Input: $(K, N, A, C, T) \in \{0,1\}^k \times \{0,1\}^m \times \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^t$
Output: $M \in \{0,1\}^{|C|}$ or $\perp$

$C_1 \dots C_{\ell_M} \xleftarrow{n} C$
for $i = 1, \dots, \ell_M$ **do**
  $M_i \leftarrow C_i \oplus \mathsf{P}(N \| 0^{n-m} \oplus \mathsf{mask}_K^{i-1,1}) \oplus \mathsf{mask}_K^{i-1,1}$
$M \leftarrow \lfloor M_1 \dots M_{\ell_M} \rfloor_{|C|}$
$A_1 \dots A_{\ell_A} \xleftarrow{n} N \| A \| 1$
$C_1 \dots C_{\ell_C} \xleftarrow{n} C \| 1$
$\bar{T} \leftarrow A_1$
for $i = 2, \dots, \ell_A$ **do**
  $\bar{T} \leftarrow \bar{T} \oplus \mathsf{P}(A_i \oplus \mathsf{mask}_K^{i-1,0}) \oplus \mathsf{mask}_K^{i-1,0}$
for $i = 1, \dots, \ell_C$ **do**
  $\bar{T} \leftarrow \bar{T} \oplus \mathsf{P}(C_i \oplus \mathsf{mask}_K^{i-1,2}) \oplus \mathsf{mask}_K^{i-1,2}$
$\bar{T} \leftarrow \mathsf{P}(\bar{T} \oplus \mathsf{mask}_K^{0,0}) \oplus \mathsf{mask}_K^{0,0}$
return $\lfloor \bar{T} \rfloor_t = T \ ? \ M \ : \ \perp$

Target Applications

In general, the target application of Elephant is lightweight cryptography with a small footprint, while still providing the option to speed up using parallel implementations. Note that, indeed, Elephant is the candidate in the competition with the smallest cryptographic primitive, while still achieving comparable security, and the only remaining candidate offering parallelizability. This, together with the different levels of security that the mode achieves, additionally puts Elephant v2 at advantage over the current NIST standards

Xoodyak

Xoodyak is a versatile cryptographic object that is suitable for most symmetric-key functions, including hashing, pseudo-random bit generation, authentication, encryption and authenticated encryption. It is based on the duplex construction, and on its full-state variant when it is fed with a secret key. In practice, Xoodyak is straightforward to use, and its implementation can be shared for many different use cases.

Internally, Xoodyak makes use of the Xoodoo permutation. The design approach of this 384-bit permutation is inspired by *Keccak-p*. While it is dimensioned like Gimli authenticated ciphers for efficiency on low-end processors. The structure consists of three planes of 128 bits each, which interact per 3-bit columns through mixing and nonlinear operations, and which otherwise move as three independent rigid objects. Its round function lends itself nicely to low-end 32-bit processors as well as to compact dedicated hardware. The mode of operation on top of Xoodoo is called Cyclist, as a lightweight counterpart to *KEYAK*'s Motorist mode. It is simpler than Motorist, mainly thanks to the absence of parallel variants. Another important difference is that Cyclist is not limited to authenticated encryption, but rather offers fine-grained services and supports hashing. Of interest in the realm of embedded devices, Xoodyak contains several built-in mechanisms that help protect against side-channel attacks.

•Cyclist can absorb the session counter that serves as nonce in chunks of a few bits. This counters differential power analysis (DPA) by limiting the degrees of freedom of an attacker when exploiting a selection function

•Another mechanism consists in replacing the incrementation of a counter with a key derivation mechanism: After using a secret key, a derived key is produced and saved for the next invocation of Xoodyak. The key then becomes a moving target for the attacker

•Then, to mitigate the impact of recovering the internal state, e.g., after a side channel attack, the Cyclist mode offers a ratchet mechanism. This mechanism offers forward secrecy and prevents the attacker from recovering the secret key prior to the application of the ratchet

•Finally, the Xoodoo round function lends itself to efficient masking counter measures against differential power analysis and similar attacks.

Notations

1. The set of all bit strings is denoted $\mathbb{Z}_2^*$ and ε is the empty string.
2. Xoodyak works with bytes and in the sequel, they assume that all strings have a length that is multiple of 8 bits. The length in bytes of a string X is denoted |X|, which is equal to its bit length divided by 8.
3. They denote a sequence of m strings $X^{(0)}$ to $X^{(m-1)}$ as $X^{(m-1)} \circ \ldots \circ X^{(1)} \circ X^{(0)}$.
4. The set of all sequences of strings is denoted $(\mathbb{Z}_2^*)^*$ and ∅ is the sequence containing no strings at all.
5. They denote with $\text{enc}_8(x)$ a byte whose value is the integer $x \in Z_{256}$.

Xoodyak is a stateful object. It offers two modes: the hash and the keyed modes, one of which is selected upon initialization.

Hash (Unkeyed) Mode

In hash mode, it can absorb input strings and squeeze digests at will. Absorb(X) absorbs an input string X, while Squeeze(ℓ) produces an ℓ-byte output depending on the data absorbed so far. The simplest case goes as follows:

$\textsc{Cyclist}(\epsilon, \epsilon, \epsilon)$ {initialization in hash mode}
$\textsc{Absorb}(x)$ {absorb string x}
$h \leftarrow \textsc{Squeeze}(n)$ {get n bytes of output}

Xoodyak offers 128-bit security against any attack, unless easier on a random oracle. To get 128-bit collision resistance, we need to set n≥32 bytes (256bits), while for a matching level of (second) preimage resistance, it is required to have n≥16 bytes (128bits). This is like the SHAKE128 extendable output function (XOF – Extendable Output Functions in SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions)

$\textsc{Cyclist}(\epsilon, \epsilon, \epsilon)$
$\textsc{Absorb}(x)$
$\textsc{Absorb}(y)$
$h_1 \leftarrow \textsc{Squeeze}(n_1)$
$\textsc{Absorb}(z)$
$h_2 \leftarrow \textsc{Squeeze}(n_2)$

Keyed Mode

In keyed mode, Xoodyak can do stream encryption, message authentication code (MAC) computation and authenticated encryption.

$$\textsc{Cyclist}(K, \epsilon, \epsilon) \ \{\text{initialization in keyed mode with key } K\}$$
$$\textsc{Absorb}(M) \ \{\text{absorb message } M\}$$
$$T \leftarrow \textsc{Squeeze}(t) \ \{\text{get tag } T\}$$

The last line produces a t-byte tag, where *t* can be specified by the application. A typical tag length would be *t*= 16 bytes (128bits).

Encryption is done in a stream cipher-like way; hence it requires a nonce. The obvious way to do encryption would be do call Squeeze () and use the output as a keystream. Encrypt(P) works similarly, but it also absorbs P block per block as it is being encrypted.

$$\textsc{Cyclist}(K, \epsilon, \epsilon)$$
$$\textsc{Absorb}(\text{nonce})$$
$$C \leftarrow \textsc{Encrypt}(P) \ \{\text{get ciphertext } C\}$$

And to decrypt ciphertext C,

$$P \leftarrow \textsc{Decrypt}(C) \ \{\text{get plaintext } P\}$$

Symbols and strings appended to the process history.

Specifications

Xoodyak is an instance of the Cyclist mode of operation on top of the Xoodoo[12] permutation.

Hash mode:	
$\textsc{Absorb}(X)$	$X \circ \textsc{Absorb}$
$\textsc{Squeeze}(\ell)$ after another $\textsc{Squeeze}()$	$\textsc{Block}^{n_{\text{hash}}(\ell)} \circ \textsc{Squeeze}$
$\textsc{Squeeze}(\ell)$ (otherwise)	$\textsc{Block}^{n_{\text{hash}}(\ell)}$
Keyed mode:	
$\textsc{Cyclist}(K, \text{id}, \text{counter})$	$\text{counter} \circ \text{id} \circ \textsc{AbsorbKey}$
$\textsc{Absorb}(X)$	$X \circ \textsc{Absorb}$
$C \leftarrow \textsc{Encrypt}(P)$	$P \circ \textsc{Crypt}$
$P \leftarrow \textsc{Decrypt}(C)$	$P \circ \textsc{Crypt}$
$\textsc{Squeeze}(\ell)$	$\textsc{Block}^{n_{\text{kout}}(\ell)} \circ \textsc{Squeeze}$
$\textsc{SqueezeKey}(\ell)$	$\textsc{Block}^{n_{\text{kout}}(\ell)} \circ \textsc{SqueezeKey}$
$\textsc{Ratchet}()$	$\textsc{Ratchet}$

Algorithms

Algorithm 1 Definition of $\textsc{Cyclist}[f, R_{\text{hash}}, R_{\text{kin}}, R_{\text{kout}}, \ell_{\text{ratchet}}]$

Instantiation: cyclist $\leftarrow \textsc{Cyclist}[f, R_{\text{hash}}, R_{\text{kin}}, R_{\text{kout}}, \ell_{\text{ratchet}}](K, \text{id}, \text{counter})$
  Phase and state: $(\textsc{phase}, s) \leftarrow (\text{up}, \text{`00`}^{b'})$
  Mode and absorb rate: $(\textsc{mode}, R_{\text{absorb}}, R_{\text{squeeze}}) \leftarrow (\text{hash}, R_{\text{hash}}, R_{\text{hash}})$
  if K not empty **then** $\textsc{AbsorbKey}(K, \text{id}, \text{counter})$

Interface: $\textsc{Absorb}(X)$
  $\textsc{AbsorbAny}(X, R_{\text{absorb}}, \text{`03`}\ (\text{absorb}))$

Interface: $C \leftarrow \textsc{Encrypt}(P)$, with $\textsc{mode} = \text{keyed}$
  return $\textsc{Crypt}(P, \text{false})$

Interface: $P \leftarrow \textsc{Decrypt}(C)$, with $\textsc{mode} = \text{keyed}$
  return $\textsc{Crypt}(C, \text{true})$

Interface: $Y \leftarrow \textsc{Squeeze}(\ell)$
  return $\textsc{SqueezeAny}(\ell, \text{`40`}\ (\text{squeeze}))$

Interface: $Y \leftarrow \textsc{SqueezeKey}(\ell)$, with $\textsc{mode} = \text{keyed}$
  return $\textsc{SqueezeAny}(\ell, \text{`20`}\ (\text{key}))$

Interface: $\textsc{Ratchet}()$, with $\textsc{mode} = \text{keyed}$
  $\textsc{AbsorbAny}(\textsc{SqueezeAny}(\ell_{\text{ratchet}}, \text{`10`}\ (\text{ratchet})), R_{\text{absorb}}, \text{`00`})$

Algorithm 2 Internal interfaces of $\textsc{Cyclist}[f, R_{\text{hash}}, R_{\text{kin}}, R_{\text{kout}}, \ell_{\text{ratchet}}]$

Internal interface: $\textsc{AbsorbAny}(X, r, c_D)$
  for all blocks X_i in $\textsc{Split}(X, r)$ **do**
    if $\textsc{phase} \neq \text{up}$ **then** $\textsc{Up}(0, \text{`00`})$
    $\textsc{Down}(X_i, c_D$ **if** first block **else** `00`$)$

Internal interface: $\textsc{AbsorbKey}(K, \text{id}, \text{counter})$, with $|K \,||\, \text{id}| \leq R_{\text{kin}} - 1$
  $(\textsc{mode}, R_{\text{absorb}}, R_{\text{squeeze}}) \leftarrow (\text{keyed}, R_{\text{kin}}, R_{\text{kout}})$
  $\textsc{AbsorbAny}(K \,||\, \text{id} \,||\, \text{enc}_8(|\text{id}|), R_{\text{absorb}}, \text{`02`}\ (\text{key}))$
  if counter not empty **then** $\textsc{AbsorbAny}(\text{counter}, 1, \text{`00`})$

Internal interface: $O \leftarrow \textsc{Crypt}(I, \textsc{decrypt})$
  for all blocks I_i in $\textsc{Split}(I, R_{\text{kout}})$ **do**
    $O_i \leftarrow I_i \oplus \textsc{Up}(|I_i|, \text{`80`}\ (\text{crypt})$ **if** first block **else** `00`$)$
    $P_i \leftarrow O_i$ **if** $\textsc{decrypt}$ **else** I_i
    $\textsc{Down}(P_i, \text{`00`})$
  return $||_i\, O_i$

Internal interface: $Y \leftarrow \textsc{SqueezeAny}(\ell, c_U)$
  $Y \leftarrow \textsc{Up}(\min(\ell, R_{\text{squeeze}}), c_U)$
  while $|Y| < \ell$ **do**
    $\textsc{Down}(\epsilon, \text{`00`})$
    $Y \leftarrow Y \,||\, \textsc{Up}(\min(\ell - |Y|, R_{\text{squeeze}}), \text{`00`})$
  return Y

Internal interface: $\textsc{Down}(X_i, c_D)$
  $(\textsc{phase}, s) \leftarrow (\text{down}, s \oplo (X_i \,||\, \text{`01`} \,||\, \text{`00`}^* \,||\, (c_D \,\&\, \text{`01`}$ **if** $\textsc{mode} = \text{hash}$ **else** $c_D)))$

Internal interface: $Y_i \leftarrow \textsc{Up}(|Y_i|, c_U)$
  $(\textsc{phase}, s) \leftarrow (\text{up}, f(s$ **if** $\textsc{mode} = \text{hash}$ **else** $s \oplo (\text{`00`}^* \,||\, c_U)))$
  return $s[0] \,||\, s[1] \,||\, \ldots \,||\, s[|Y_i| - 1]$

Internal interface: $[X_i] \leftarrow \textsc{Split}(X, n)$
  if X is empty **then return** array with a single empty string $[\epsilon]$
  return array $[X_i]$, with $X = ||_i\, X_i$ and $|X_i| = n$ except possibly the last block.

Xoodyak and its claimed security

Xoodyak is Cyclist[f,R_{hash},R_{kin},R_{kout},$l_{ratchet}$] with

•f=Xoodoo[12] of width 48 bytes (or b= 384 bits)

•R_{hash}= 16 bytes

•R_{kin}= 44 bytes

•R_{kout}= 24 bytes

• $l_{ratchet}$ = 16 bytes

1. The success probability of any attack on Xoodyak in hash mode shall not be higher than the sum of that for a random oracle and $N^2/2^{255}$, with N the attack complexity in calls to Xoodoo[12] or its inverse. This means that Xoodyak hashing has essentially the same claimed security as, e.g., SHAKE128.
2. Let K=(K_0, … ,K_{u-1}) be an array of u secret keys, each uniformly and independently chosen from $\mathbb{Z}_2{}^{\kappa}$ with $\kappa \le 256$ and κ a multiple of 8. Then, the advantage of distinguishing the array of Xoodyak objects after initialization with Cyclist(Ki,·,·)with i∈ $\mathbb{Z}_u$ from an array of random oracles $RO(i,h)$, where h∈$(Z_2{}^* \cup S)^*$ is a process history,

$$\frac{q_{iv}N + \binom{u}{2}}{2^{\kappa}} + \frac{N}{2^{184}} + \frac{(L+\Omega)N + \binom{L+\Omega+1}{2}}{2^{192}} + \frac{M^2}{2^{382}} + \frac{Mq}{2^{\min(192+\kappa,384)}}$$

   the notations of the generic security bound of the full-state keyed duplex, namely.
   - N is the computational complexity expressed in the (computationally equivalent) number of executions of Xoodoo[12].
   - M is the online or data complexity expressed in the total number of input and output blocks processed by Xoodyak.
   - $q \le M$ is the total number of initializations in keyed mode.
   - $\Omega \le M$ is the number of blocks, in keyed mode, that overwrite the outer state (i.e., the first R_{kout} bytes of the state) and for which the adversary gets a subsequent output block.
   - $L \le M$ is the number of blocks, in keyed mode, for which the adversary knows the value of the outer state from a previous query and can choose the input block value (e.g., in the case of authentication without a nonce, or of authenticated encryption with nonce repetition).

- $q_{iv} \leq u$ is the maximum number of keys that are used with the same id.

1 and 2 ensure Xoodyak has 128bits of security both in hash and keyed modes (assuming $\kappa \geq 128$). Regarding the data complexity, it depends on the values of q, Ω and L. Given that they are bounded by M, Xoodyak resists to a data complexity of up to 264 blocks The equation is negligible as long as $N << 2^{128}$ and $M << 2^{64}$. In the case of $L + \Omega = 0$, it resists even higher data complexities, as the probability remains negligible also when $M << 2^{160}$.

The parameter q_{iv} relates to the possible security degradations in the case of multi-target attacks, as an exhaustive key search would erode security by $\log_2 q_{iv} \leq \log_2 u$ bits in this case. However, when the protocol ensures $q_{iv} = 1$, there is no degradation and the security remains at min (128, κ) bits even in the case of multi-target attacks.

Implementation

The purpose of the NIST Lightweight Cryptography Standardization Process is to get algorithms that are suitable for use in constrained environments. In this section, they test the implementation of Xoodyak and report implementations on ARM Cortex-M0 and Cortex-M3.

	ARM Cortex-M0	ARM Cortex-M3
Hash mode		
Absorb()	134.5	39.3
Squeeze()	136.2	40.6
Keyed mode		
Absorb()	48.7	14.2
Encrypt()	91.2	27.1
Decrypt()	91.3	27.4
Squeeze()	86.2	24.3

	Xoodyak	of which Xoodoo[12]	Notes
ARM Cortex-M0	3494	468	1 round in a loop
ARM Cortex-M3	4058	2388	12 rounds unrolled

Code sizes (in bytes) of our implementations of Xoodyak and of the permutation only.

As a comparison, the Sparkle384 permutation takes 484 bytes on Cortex-M3, which shows that Xoodoo's code size is competitive when it is not unrolled. Note that an unrolled implementation of the Gimli permutation takes 3950 bytes on Cortex-M3. At the time of this writing, Xoodyak experimenting with code size optimizations, and we expect to reach around 1200 bytes of code for the full Xoodyak with similar speeds.

How Project Works

For Elephant Algorithm, there is an "elephant.c" file that contains all the code of Elephant Algorithm. And also there is a makefile that allow user build, debug or execute C code.

"make all" command builds C file and creates executable inside 'bin' file. "./main" command runs the Algorithm.

```
char pl[ELEPHANTBYTES]="gtuceng";
char hex[ELEPHANTBYTES]="";
char keyhex[2*BYTESOFKEY+1]="ez5BmBPpOK8dkwJxzmbK5n30N5z4gdKy";
char nonce[2*ELEPHANT_NPUBBYTES+1]="100110111100100111101110 ";
char additional[KEEPBYTES]="gtuceng";
```

There is a keyHex, a Nonce and additional data inside that C file, it can be changed.

For Xoodyak Algorithm, there are 4 different files that contains all Xoodyak implementations. "xoodyak_cycle.h" header and "xoodyak_cycle.c" C files contain all function definitions and functions written.

There are two different test files which are "xoodyak_hash.c" and "xoodyak_hash.c" files. "xoodyak_hash.c" file contains test for hash(unkeyed) part of the Xoodyak Algorithm. On the other hand,

“xoodyak_keyed.c” file contains test for keyed part. Same example used in these C files that is used in this report which starts at second page of the Xoodyak Algorithm explanation.

There is also Makefile included in this part,

“make xoodyak_hash” command builds hash project, “make xoodyak_keyed” command builds keyed test of the project and “make all” command builds both of these projects. Executables are created in the source folder so it is easy to execute. For Clean part, “make clean” command should clear all unnecessary files and directories.

Keyed part:

```
xoocycle_cyclist(&cyc, (CU8P)"key", 3, xoocycle_empty, 0,
                 xoocycle_empty, 0);
xoocycle_absorb(&cyc, (CU8P)"nonce", 5);
xoocycle_absorb(&cyc, (CU8P)"associated", 10);
xoocycle_encrypt(&cyc, plain, PLAIN);

printf("\nCiphertext after encryption - ");
print8(plain, PLAIN);
xoocycle_squeeze(&cyc, tagEnc, TAG);
printf("Tag - ");
print8(tagEnc, TAG);
xoocycle_erase(&cyc);
printf("Xoodyak Settings Removed!!\n");
printf("---------------------------------------\n");

printf("Xoodyak Decryption\n");
printf("Ciphertext - ");
print8(plain, PLAIN);
xoocycle_cyclist(&cyc, (CU8P)"key", 3, xoocycle_empty, 0,
                 xoocycle_empty, 0);
printf("Xoodyak Settings Done!!\n");

xoocycle_absorb(&cyc, (CU8P)"nonce", 5);
xoocycle_absorb(&cyc, (CU8P)"associated", 10);
xoocycle_decrypt(&cyc, plain, PLAIN);
printf("Decrypted Text after decryption - ");
print8(plain, PLAIN);
xoocycle_squeeze(&cyc, tagDec, TAG);
printf("Tag - ");
print8(tagDec, TAG);
```

Hash part:

```
xoocycle_cyclist(&cyc, xoocycle_empty, 0, xoocycle_empty, 0,
                 xoocycle_empty, 0);
printf("Xoodyak Settings Done!!\n");
while (1) {
  len = read(STDIN_FILENO, io, IO);
  if (len < 0) {
    fprintf(stderr, "error\n");
    exit(EXIT_FAILURE);
  }
  if (len == 1) {
    break;
  }
  xoocycle_absorb(&cyc, io, len);
  printf("Hash Creating - ");
  print8(io, HASH);
}
xoocycle_squeeze(&cyc, io, HASH);
printf("Final hash Created - ");
print8(io, HASH);
xoocycle_erase(&cyc);
```

References

Elephant

1. https://www.youtube.com/watch?v=s5sgjkzRUVQ
2. https://iacr.org/cryptodb/data/paper.php?pubkey=30509
3. https://tosc.iacr.org/index.php/ToSC/article/view/8616/8182
4. https://www.esat.kuleuven.be/cosic/elephant/
5. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/elephant-spec-final.pdf
6. https://asecuritysite.com/light/lw_elephant
7. https://csrc.nist.gov/csrc/media/Projects/lightweight-cryptography/documents/finalist-round/status-updates/elephant-update.pdf
8. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/changelog-files/elephant-changes-final-round.pdf

Xoodyak

1. https://www.youtube.com/watch?v=h7chn74DCNQ
2. https://tosc.iacr.org/index.php/ToSC/article/view/8618/8184
3. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/xoodyak-spec-final.pdf
4. https://keccak.team/xoodyak.html
5. https://csrc.nist.gov/csrc/media/Projects/lightweight-cryptography/documents/finalist-round/status-updates/xoodyak-update.pdf
6. https://rweather.github.io/lightweight-crypto/xoodyak_8h.html
7. https://asecuritysite.com/light/go_xoo