PAEQ v1

Designers: Alex Biryukov and Dmitry Khovratovich
University of Luxembourg, Luxembourg

Submitters: Alex Biryukov and Dmitry Khovratovich
alex.biryukov@uni.lu, dmitry.khovratovich@uni.lu, khovratovich@gmail.com
https://www.cryptolux.org/index.php/PAEQ

15th March, 2014

Contents

Chapter 1

Specification

PAEQ [peik] (Parallelizable Authenticated Encryption based on Quadrupled AES) is an authenticated encryption scheme. It is introduced together with a generic mode of operation PPAE, which takes a permutation $\mathcal{F}$ of width n. PAEQ is PPAE where $\mathcal{F}$ is the AESQ [esk] permutation. PAEQ and AESQ within it are defined in this submission, whereas the security bounds are proved for PPAE and hence applicable to PAEQ or other instantiations of $\mathcal{F}$.

1.1 Parameters

PAEQ has three parameters: key length, nonce length, and tag length. Each parameter is an integer number of bytes.

- The key length is between 8 bytes (64 bits) and 32 bytes (256 bits).
- The nonce length is between 4 bytes (32 bits) and 32 bytes (256 bits).
- The key and nonce length in total does not exceed 48 bytes (368 bits).
- The tag length is between 4 bytes (32 bits) and 64 bytes (512 bits).

1.2 Recommended parameter sets

We recommend the following parameter sets (in bits) depending on the user's requirements to the overall security level and the need of extra features of nonce-misuse resistance and quick tag update.

Name	Key length	Nonce length	Tag length	Comment
Primary sets				
`paeq64`	64	64	64	64-bit security
`paeq80`	80	80	80	80-bit security
`paeq128`	128	96	128	128-bit security
Secondary sets				
`paeq64-t`	64	64	512	64-bit security with quick tag update
`paeq64-tnm`	64	128	512	64-bit security with nonce-misuse and tag update options
`paeq128-t`	128	128	512	128-bit security with quick tag update
`paeq128-tnm`	128	256	512	128-bit security with nonce-misuse and tag update options
`paeq192`	192	128	128	128-bit security with 192-bit keys
`paeq160`	160	128	160	160-bit security
`paeq256`	256	128	128	128-bit security with 256-bit keys

The reference implementation allows for all these parameter sets, with the tuple (128,96,128) fixed by default.

1.3 Authenticated encryption

1.3.1 Notation

The *authenticated encryption scheme* is denoted by Π and is defined as a pair of functions $\mathcal{E}$ and $\mathcal{D}$, which provide encryption and decryption, respectively. The inputs to $\mathcal{E}$ are a plaintext P, associated data A, a public message number N, a key K, and a tag length τ.

- The number of bytes in N is the nonce length. The bit length of N is denoted by r.
- The number of bytes in K is the key length. The bit length of K is denoted by k.
- The number of bytes in P is between 1 and 2^{96}, and in A is between 0 and 2^{96}.
- There is no secret message number; i.e., the secret message number is empty.

The encryption function outputs ciphertext C and tag T:

$$\mathcal{E}:\ \mathcal{K}\times\mathcal{N}\times\mathcal{A}\times\mathcal{P}\rightarrow\mathcal{C}\times\mathcal{T}.$$

For fixed (N,A,K) the encryption function is injective and hence defines the decryption function. The decryption function takes a key, a nonce, associated data, the ciphertext, and the tag as input and returns either a plaintext or the "invalid" message $\perp$:

$$\mathcal{D}:\ \mathcal{K}\times\mathcal{N}\times\mathcal{A}\times\mathcal{C}\times\mathcal{T}\rightarrow\mathcal{P}\cup\{\perp\}.$$

For brevity we will write $\mathcal{E}_K^{N,A}(\cdot)$ and $\mathcal{D}_K^{N,A}(\cdot)$.

To accomodate nonce-misuse cases, the user may choose to generate N as a keyed function of plaintext, key, and associated data. It is then called *extra nonce* and denoted by N_e. The user is then supposed to communicate N_e to the receiver, with ordinary nonce transmission rules applied.

Let X be an internal s-bit variable. Then we refer to its bits as $X[1],X[2],\ldots,X[s]$ and denote the subblock with bits from s_1 to s_2 as $X[s_1\ldots s_2]$. In pictures and formulas the least' significant bits and bytes are at the left, and the concatenation of multi-bit variables is defined as follows:

$$X||Y = X[1]X[2]\ldots X[t]Y[1]Y[2]\ldots Y[s].$$

The counter values have their least significant bits as the least significant bits of corresponding variables.

PAEQ is a concrete instantiation of generic mode PPAE, which takes permutation $\mathcal{F}$ of width n. In this submission n is always equal to 512, and $\mathcal{F}$ is the AESQ permutation defined below.

1.3.2 PPAE mode of operation

For the domain separation the following two-byte constants are used:

$$D_i = (k,(r+i) \pmod{256}),\ i=1,2,3,4,5,6,$$

where the second value is taken modulo 256.

We refer to Figure 1.1 for a graphical illustration of the PAEQ functionality.

Encryption.

1. During the first stage the plaintext is splitted into blocks P_i of $(n-k-16)$ bits ($n/8-k/8-2$ bytes) $P_1,P_2,\ldots,P_l$. We encrypt block P_i as follows:

$$\begin{aligned}V_i &\leftarrow D_0||R_i||N||K;\\ W_i &\leftarrow \mathcal{F}(V_i);\\ C_i &\leftarrow W_i[17..(n-k)]\oplus P_i.\end{aligned}$$

   Here V_i,W_i are n-bit intermediate variables, and C_i is a $(n-k-16)$-bit block. The counter $R_i=i$ occupies the $(n-k-r-16)$-bit block. If last plaintext block is incomplete and has length t', then D_0 is replaced with D_1.

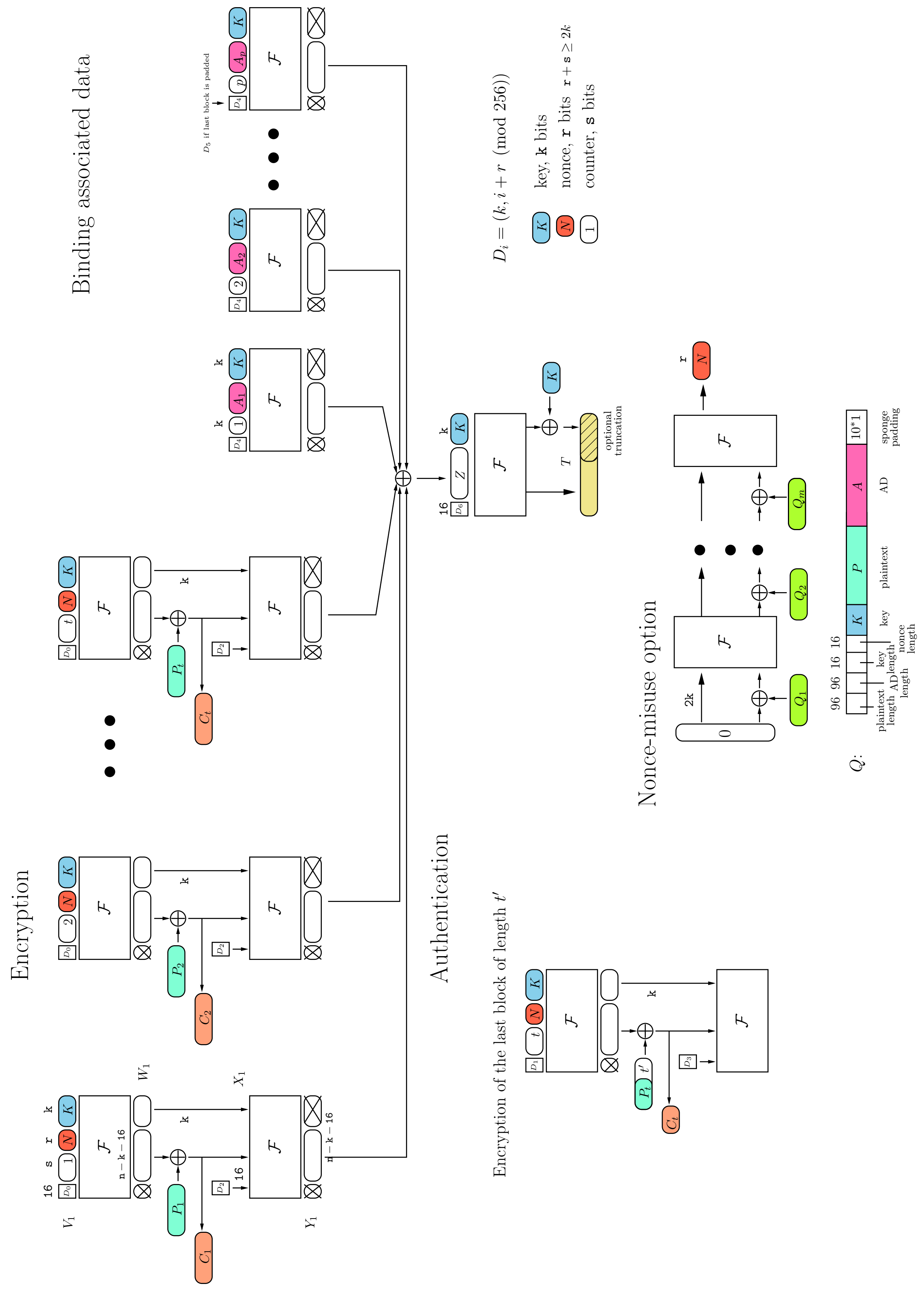

Figure 1.1: Encryption and authentication with PAEQ.

2. During the second stage we compute intermediate variables for authentication:

$$X_i \leftarrow D_2||C_i||W_i[(n-k+1)..n];$$
$$Y_i \leftarrow (\mathcal{F}(X_i))[17..(n-k)].$$

If the last plaintext block P_t is incomplete, then we define $P'_t = P_t||a||a||\dots||a$, where a is one-byte variable with value equal to t', the byte length of P_t, and define

$$C'_t \leftarrow W_t[17..(n-k)] \oplus P'_t;$$
$$X_t \leftarrow D_3||C'_t||W_i[(n-k+1)..n].$$

Finally,

$$Y \leftarrow \bigoplus_i Y_i;$$

3. During the third stage we compute an intermediate variable for authenticating associated data. The AD is splitted into blocks $A_1, A_2, \dots, A_s$ of length $n-2k-16$. Then we compute

$$X'_i \leftarrow D_4||R_i||A_i||K;$$
$$Y'_i \leftarrow (\mathcal{F}(X'_i))[17..(n-k)],$$

where R_i is a k-bit counter starting with 1. If the last AD block is incomplete (has t'' bytes), it is padded with bytes whose value is t'', and the constant D_4 is changed to D_5. Finally,

$$Y' \leftarrow \bigoplus_i Y'_i;$$

4. In the final stage we compute the tag. First, we define

$$Z \leftarrow Y \bigoplus Y'.$$

and then,

$$T \leftarrow \mathcal{F}(D_6||Z||K) \oplus (0^{n-k}||K).$$

The tag T is truncated to $T[1\dots\tau]$.

The encryption and authentication process is illustrated in Figure 1.1.

Decryption. The decryption process repeats the encryption process with minor corrections. We decrypt as

$$P_i \leftarrow W_i[17..(n-k)] \oplus C_i$$

with appropriate corrections for the incomplete block if needed. C_i is used as given when composing X_i. Finally, the tag T' is computed and matched with submitted T. If $T \neq T'$ (including length mismatch), the decryption process returns invalid.

The plaintext is returned only if the tags match.

1.3.3 The AESQ permutation

The $\mathcal{F}$ permutation in this submission is the AESQ permutation, which is defined below. AESQ operates on 512-bit inputs, which are viewed as four 128-bit registers. The state undergoes 20 identical rounds. The rounds use standard AES operations: SubBytes, ShiftRows, and MixColumns, which are applied to the 128-bit registers exactly as in AES (Figure 1.2).

The round constants are chosen as follows in the matrix register view:

$$Q_{i,j,k} = \begin{pmatrix} 8i+4j+k & 8i+4j+k & 8i+4j+k & 8i+4j+k \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Here is the Shuffle mapping that permutes columns of the internal states:

	A				B				C				D			
From	$A[0]$	$A[1]$	$A[2]$	$A[3]$	$B[0]$	$B[1]$	$B[2]$	$B[3]$	$C[0]$	$C[1]$	$C[2]$	$C[3]$	$D[0]$	$D[1]$	$D[2]$	$D[3]$
To	$A[3]$	$D[3]$	$C[2]$	$B[2]$	$A[1]$	$D[1]$	$C[0]$	$B[0]$	$A[2]$	$D[2]$	$C[3]$	$B[3]$	$A[0]$	$D[0]$	$C[1]$	$B[1]$

```
Input: 128-bit states $A, B, C, D$, round constants $Q_{i,j,k}$
for $0 \leq i < R = 10$ do
    for $0 \leq j < 2$ do
        $A \leftarrow \text{MixColumns} \circ \text{ShiftRows} \circ \text{SubBytes}(A)$;
        $A_0 \leftarrow A_0 \oplus Q_{i,j,1}$;
        $B \leftarrow \text{MixColumns} \circ \text{ShiftRows} \circ \text{SubBytes}(B)$;
        $B_0 \leftarrow B_0 \oplus Q_{i,j,2}$;
        $C \leftarrow \text{MixColumns} \circ \text{ShiftRows} \circ \text{SubBytes}(C)$;
        $C_0 \leftarrow C_0 \oplus Q_{i,j,3}$;
        $D \leftarrow \text{MixColumns} \circ \text{ShiftRows} \circ \text{SubBytes}(D)$;
        $D_0 \leftarrow D_0 \oplus Q_{i,j,4}$;
    end
    $(A, B, C, D) \leftarrow \text{Shuffle}(A, B, C, D)$
end
```

Algorithm 1: Pseudocode for the AESQ permutation with $2R$ rounds

1.3.4 Extra nonce

The extra nonce N_e is a function of the key, the plaintext, and the associated data. It is not online, i.e. it needs to know the length of all inputs and the output. It is the sponge hash function:

$$T_1 = \mathcal{F}(Q_1||0^{2k}),\ T_2 = \mathcal{F}(T_1 \oplus (Q_2||0^{2k})),\ \ldots,\ T_m = \mathcal{F}(T_{m-1} \oplus (Q_m||0^{2k})),$$

and N_e is the truncation of T_m to r bits. The injection blocks Q_i come from the string Q, which is composed as follows:

$$Q = |P|\,||\,|A|\,||\,k_b||\,r_b\,||K||P||A||10^*1,$$

where $|P|$ is the plaintext length in bytes, $|A|$ is the associated data length in bytes, k_b is the key length in bytes, r_b is the nonce length in bytes, and 10^*1 is the sponge padding: one byte with value 1, then as many zero bytes as needed to fill all but one bytes in the injection block, and then the byte with value 1.

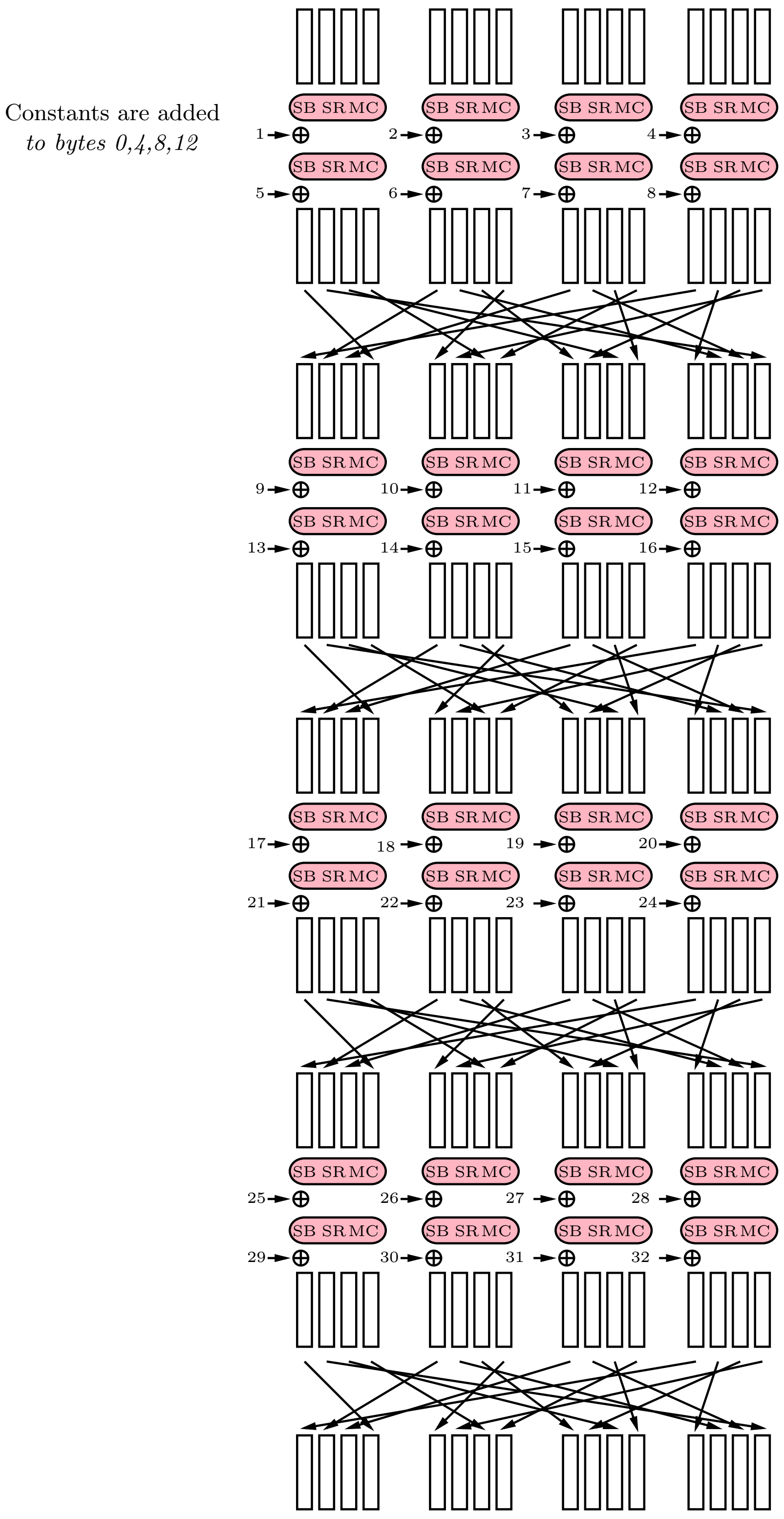

Figure 1.2: 8 rounds of the AESQ permutation.

Chapter 2

Security goals

Now we formulate the security goals for our primary sets of parameters, where the key, the nonce, and the tag have the same length.

	Key/tag length		
	64 bits	80 bits	128 bits
Confidentiality for the plaintext	64	80	128
Integrity for the plaintext	64	80	128
Integrity for the associated data	64	80	128
Integrity for the public message number	64	80	128

There is no secret message number. The public message number is a nonce.

The security bounds are proved in the random-permutation model in Theorems 1 (confidentiality) and 2 (integrity).

If the nonce is reused to encrypt another pair (plaintext, associated data) under the same key, then the confidentiality for the plaintext is no longer promised. However, the integrity for both plaintext and associated data is promised up to the half of the security level, i.e. half of the key length (Theorem 3).

Additionally, the scheme offers the nonce-misuse feature, which generates the nonce (called *extra nonce*) as a MAC out of the key, the plaintext, and the associated data. If the extra nonce is used, the scheme offers confidentiality and integrity with the normal security level or half of the nonce length, whatever is smaller (Proposition 1).

It is safe to use a single key with nonces of different lengths.

Chapter 3

Security analysis

3.1 Security analysis of the mode of operation

In this section we provide the security analysis of the PPAE mode of operation, which takes a permutation $\mathcal{F}$ of width n. Though this submission fixes the permutation to AESQ and its width to 512 bits, the following proof is useful when defining other scheme within the PPAE mode.

The security of a AE scheme is defined as the inability to distinguish between the two worlds, where an adversary has access to some oracles and a permutation. One world consists of the encryption oracle $\mathcal{E}_K(\cdot,\cdot,\cdot)$ and decryption oracle $\mathcal{D}_K(\cdot,\cdot,\cdot)$, where the secret key is randomly chosen and shared. The second world consists of the "random-bits" oracle $\$(\cdot,\cdot,\cdot,\cdot)$ and the "always-invalid" oracle $\perp(\cdot,\cdot,\cdot,\cdot)$. In addition, the adversary and all the oracles have an oracle access to the permutation $\mathcal{F}$. The encryption requests must be nonce-respecting, i.e. all the nonces in those requests must be distinct. A decryption request (N, A, C, T) shall not contain the ciphertext previously obtained with (N, A).

We give a security proof for a fixed key length k, nonce length r, and random permutation $\mathcal{F}$ of width n. The variable-length security proof is not given in this submission, but can be mounted thanks to the D_i constants. The proof also does not take into account the calls to $\mathcal{F}$ made in the extra nonce computation.

3.1.1 Confidentiality

We prove confidentiality as indistinguishability of the pair (ciphertext, tag) from a random string for the tag length n. The result for truncated tags is a trivial corollary. Let $\Pi[\mathcal{F}]$ be the PPAE mode instantiated with permutation $\mathcal{F}$. We do not consider the decryption oracle here, as we later show that with overwhelming probability it always returns $\perp$.

Theorem 1. *Suppose adversary $\mathcal{A}$ has access to $\Pi[\mathcal{F}]$, $\mathcal{F}$, and $\mathcal{F}^{-1}$, and let $K \xleftarrow{\$} \mathcal{K}$. Let σ_Π be the total number of queries to $\mathcal{F}$ made during the calls to the Π oracle, and $\sigma_{\mathcal{F}}$ be the total number of queries to $\mathcal{F}$ and $\mathcal{F}^{-1}$ oracles together. Then his advantage of distinguishing the oracle $\Pi[\mathcal{F}]$ from the random-bits oracle $\$$ is upper bounded as follows:*

$$\mathbf{Adv}_{\Pi}^{\mathrm{conf}}(\mathcal{A}) \leq \frac{(\sigma_{\mathcal{F}} + \sigma_\Pi)^2}{2^n} + \frac{2\sigma_\Pi^2}{2^{n-k-16}} + \frac{2\sigma_{\mathcal{F}}^2}{2^k} + \frac{2\sigma_{\mathcal{F}}\sigma_\Pi}{2^{n-16}}. \tag{3.1}$$

Proof. We denote i-th ciphertext by $C^{(i)}$ and its j-th block by $C_j^{(i)}$. The same notation is used for intermediate variables X, Y, Z.

In the games G_i below, p_i is the probability that $\mathcal{A}$ outputs 1. The following transitions between games are made to prove the high probability of the following properties: first, that Π and the adversary ask non-intersecting sets of queries to $\mathcal{F}$ or $\mathcal{F}^{-1}$, and secondly, that the ciphertexts and the tags are composed of random and independent blocks and hence are indistinguishable from random strings.

Game G_0. The permutation $\mathcal{F}$ is randomly chosen. The adversary interacts with Π, $\mathcal{F}$, and $\mathcal{F}^{-1}$.

Game G_1. The permutation $\mathcal{F}$ is defined in a lazy manner: we maintain a table $T_{\mathcal{F}}$ which is initially empty, and fill the table values whenever $\mathcal{F}$ or $\mathcal{F}^{-1}$ are queried alone or within Π.

Game G_2. We modify oracles $\mathcal{F}$ or $\mathcal{F}^{-1}$ so that they choose the undefined values completely at random from $\{0,1\}^n$, but explicitly abort in case the permutation constraint is not satisfied. The probability of such an abort is upper bounded by $(\sigma_{\mathcal{F}} + \sigma_\Pi)^2/2^n$.

Game G_3. We now require Π to explicitly abort if there is a collision in the set $\{X_j^{(i)}[17 \ldots (n-k)]\}_{i,j}$ or in the set $\{T[1 \ldots (n-k)]^{(i)}\}$. The probability of this event is upper bounded by $\sigma_\Pi^2/2^{n-k-16}$.

Game G_4. We now require that the adversary does not ask the oracle $\mathcal{F}$ with a query ending with K. Since the outputs of $\mathcal{F}^{-1}$ are completely random, the adversary has no choice but to guess K. Hence the probability of this event is upper bounded by $\sigma_{\mathcal{F}}/2^k$.

Game G_5. We now require the oracle $\mathcal{F}^{-1}$ to explicitly abort if its reply ends with K. The adversary can guess the unknown parts of X or $T \oplus K$, respectively (he can not hope for collisions in X or T due to the transition in game G_3). The probability of this event for all cases is upper bounded by $\sigma_{\mathcal{F}}/2^k$.

Game G_6. We require Π to abort if it produces such $X_j^{(i)}$ that it is already in $T_{\mathcal{F}}$. The probability of this event is upper bounded by $\sigma_{\mathcal{F}}\sigma_\Pi/2^{n-16}$. We note that the adversary can not manipulate X_j with M_j since the choice is made before the X_j is generated.

Game G_7. In the same manner, we require $\mathcal{F}$ to abort if it is queried with an already generated $X_j^{(i)}$ and $\mathcal{F}^{-1}$ to abort if it is queried with an already generated $X_j^{(i)}$. The probability of this event is upper bounded by $\sigma_{\mathcal{F}}\sigma_\Pi/2^{n-16}$.

Game G_8. Finally, we require Π to abort if there is a collision in the set $\{Z^{(i)}\}_i$. Since every Z is the XOR of uniformly generated Y_i and Y_i', the probability of this event is upper bounded by $\sigma_\Pi^2/2^{n-k-16}$.

Let us now look at game G_8. The oracle Π, in fact, never asks the oracle $\mathcal{F}$ with a query that corresponds to an existing entry in $T_{\mathcal{F}}$. We ruled out such an event in games G_4, G_5 for the first layer of encryption and the layer of the AD processing. We ruled out the event in games G_6, G_7 for the second layer of encryption. Finally, we ensure this for the authentication level in game G_8. The game G_3 is auxiliary.

The indistinguishability of the pair (ciphertext,tag) from a random string follows from the following facts:

- all the $V_j^{(i)}$ inputs are distinct (nonce-misuse requirement);
- hence, all the values $W_j^{(i)}$ are chosen uniformly in Games G_2–G_8;
- all the $Z^{(i)}$ values are distinct (Game G_8);
- all the tags T are chosen uniformly (and independently of ciphertexts) in Game G_8.

Therefore, Game G_8 is indistinguishable from the interaction with the random-bit oracle. The adversary's advantage in Game G_0 is upper bounded by the sum of transition probabilities in Games G_1–G_8, which yields Equation (3.1).

This concludes the proof. □

3.1.2 Ciphertext integrity

Theorem 2. *Let an adversary be in the setting of Theorem 1. Suppose he makes q decryption requests to Π. Then his advantage of distinguishing Π^{-1} from the "always-invalid" oracle $\perp$ is upper bounded as follows:*

$$\mathbf{Adv}_\Pi^{\mathrm{int}}(\mathcal{A}) \leq \mathbf{Adv}_\Pi^{\mathrm{conf}}(\mathcal{A}) + \frac{\sigma_\Pi^2}{2^{n-k-16}} + \frac{q}{2^\tau} + \frac{\sigma_\Pi q}{2^{n-k-16}} + \frac{q}{2^k}. \quad (3.2)$$

Proof. We first need to strengthen our setting. With probability $\mathbf{Adv}_\Pi^{\mathrm{conf}}(\mathcal{A})$ we are in Game G_8. Let us make additional computations in Game G_9.

Let M be a plaintext to encrypt, and A be the associated data, $Y(N,M)$ and $Y'(A)$ be intermediate variables. Then for each encryption request (N,M,A) we also compute $Y(N,M')$ and $Y'(A')$, for every full-block prefix M' of M and every full-block prefix A' of A. We store all the resulting Y and Y' in special tables Y_{ext} and Y'_{ext}. Let us note that the size of both tables is upper bounded by σ_Π.

Game G_9. We additionally require Π to explicitly abort if there are collisions in the set $\{Z_{ext} = Y \oplus Y' \,|\, Y \in Y_{ext},\, Y' \in Y'_{ext}\}$. The probability of this event is upper bounded by $\sigma_\Pi^2/2^{n-k-16}$.

Lemma 1. *Suppose we are in Game G_9. A valid decryption (N, A, C, T) request does not generate $\perp$ with probability upper bounded by*

$$2^{-\tau} + \sigma_\Pi/2^{n-k-16} + 1/2^k.$$

Proof. There are four possible options:

1. Fresh nonce;
2. Old nonce, fresh ciphertext (not obtained with that nonce);
3. Old nonce, old ciphertext, fresh associated data;
4. Old nonce, old ciphertext, old associated data, fresh tag.

The last option clearly never succeeds.

Fresh nonce. Since N is fresh, the decryption request turns out to be a valid encryption request. Let M' be the plaintext of length $|C|$ that consists of zero bits, and let C' be its ciphertext under nonce N. Then the decryption request (N, A, C, T) results in the encryption request $(N, A, C \oplus C')$, which generates the tag uniformly at random according to Game G_8. Therefore, the probability to forge the tag is upper bounded by $2^{-\tau}$.

Old nonce N, fresh ciphertext C. Let C' be the ciphertext previously obtained with N. Consider two cases.

1. C' is a prefix of C. Then Z is fresh due to requirements of Game G_9 and is chosen uniformly at random.
2. C' is not a prefix of C. Let j be any block index where C' differs from C. By definition, the variables X'_j and X_j are distinct. There are two cases:
   - The block $X'_j[17 \dots (n-k)]$ has been previously obtained with nonce N'' as block $X''_s[17 \dots (n-k)]$. Only one such nonce could exist because of the requirements of Game G_3. Since all X_i generated during encryption requests are sampled uniformly at random, the probability of $X''_s = X'_j$ is upper bounded by $1/2^k$.
   - The block $X'_j[17 \dots (n-k)]$ has never been obtained before. Then $Y(X'_j)$ is sampled uniformly at random, and with probability $\sigma_\Pi/2^{n-k-16}$ generates a not-fresh Z.

Therefore, Z (and hence the tag) is chosen uniformly at random in all cases except one that has probability $1/2^k$. Thus probability to forge the tag in this case is upper bounded by

$$2^{-\tau} + \sigma_\Pi/2^{n-k-16} + 1/2^k.$$

Old nonce N, old ciphertext C, fresh associated data A. Let $Y'(A)$ be the value obtained during the processing of the associated data A. There are two options:

- $Y'(A) = Y'(A^0)$ for some previously queried A^0. Due to requirements of Game G_9, the value $Y(N, C) \oplus Y'(A_0) = Z(N, C, A)$ is fresh.
- $Y'(A)$ is fresh. Then it generates a not-fresh Z with probability upper bounded by $\sigma_\Pi/2^{n-k-16}$.

Therefore, the probability to forge the tag in this case is upper bounded by

$$2^{-\tau} + \sigma_\Pi/2^{n-k-16}.$$

This concludes the proof of lemma. □

Now we are ready to prove Theorem 2. The requirements of Game G_9 are fulfilled with probability $\mathbf{Adv}_\Pi^{\mathrm{conf}}(\mathcal{A}) + \frac{\sigma_\Pi^2}{2^{n-k-16}}$. The first decryption request returns $\perp$ with probability lower bounded in Lemma 1. Then the adversary learns nothing with his request, and we can ignore filling the table $T_\mathcal{F}$. All the other requests have the same lower bound to be invalid, so we merely multiply this probability by q.

This concludes the proof of Theorem 2. □

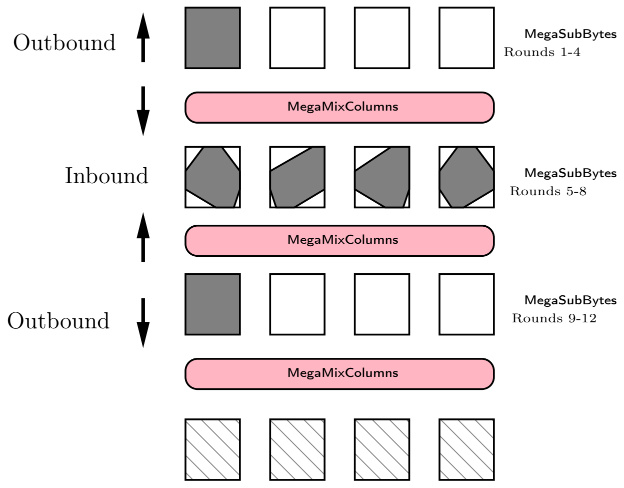

Figure 3.1: Rebound attack on AESQ.

3.1.3 Basic robustness to nonce misuse

In this section we investigate the level of robustness of PAEQ to nonce misuse, i.e. to the multiple use of the same nonce(s) for encryption.

First, we note that confidentiality is almost destroyed in the case of nonce reuse. In fact, it is easy for any person to compute a new ciphertext C' for message M under the same nonce N, as

$$C' = M \oplus M' \oplus C.$$

However, with high probability the tag T is still indistinguishable from the random string, which allows us to prove the integrity property in the case of nonce reuse, though with a worse bound.

Lemma 2. *Suppose adversary $\mathcal{A}$ has access to $\Pi[\mathcal{F}]$, $\mathcal{F}$, and $\mathcal{F}^{-1}$, and let $K \overset{\$}{\leftarrow} \mathcal{K}$. Let σ_Π be the total number of queries to $\mathcal{F}$ made during the calls to the Π oracle, and $\sigma_\mathcal{F}$ be the total number of queries to $\mathcal{F}$ and $\mathcal{F}^{-1}$ oracles together. The adversary is allowed to repeat nonces, but not to repeat plaintexts.*

Then his advantage of distinguishing the tag T produced by the oracle $\Pi[\mathcal{F}]$ from the random-bits oracle \$ is upper bounded as follows:

$$\mathbf{Adv}_{\Pi}^{\mathrm{conf}}(\mathcal{A}) \leq 2\frac{(\sigma_\mathcal{F}+\sigma_\Pi)^2}{2^k}. \tag{3.3}$$

Proof. To prove this lemma, we follow the same games as in Theorems 1 and 2. The transitions from Game G_0 to G_1 and G_2 hold with the same probability.

The Game G_3 is replaced with Game G_3', where we require Π to abort if there are collisions in $\{X_j^{(i)}[(n-k+1)\ldots k]\}$ for all used nonces and counter values. The probability of this event is upper bounded by $(\sigma_\mathcal{F}+\sigma_\Pi)^2/2^k$.

Games G_4–G_7 remain unmodified.

Finally, we unite Games G_8 and G_9 by requiring no collisions in Z_{ext}. Since all the pairs (plaintext, associated data) are distinct and there is no collision in $X[(n-k+1)\ldots k]$ according to G_3', every Z has a fresh and uniformly chosen Y or Y' in its XOR. Therefore, the upper bound on the collision event is $(\sigma_\mathcal{F}+\sigma_\Pi)^2/2^k$.

As long as all Z are unique, the tags are uniformly generated. Therefore, the adversary's advantage is upper bounded by $2(\sigma_\mathcal{F}+\sigma_\Pi)^2/2^k$. □

Now we can prove the same bounds for integrity.

Theorem 3. *Let an adversary be in the setting of Lemma 2. Suppose he makes q decryption requests to Π. Then his advantage of distinguishing Π^{-1} from the "always-invalid" oracle $\perp$ is upper bounded as follows:*

$$\mathbf{Adv}_{\Pi}^{\mathrm{int}}(\mathcal{A}) \leq 2\frac{(\sigma_\mathcal{F}+\sigma_\Pi)^2}{2^k}. \tag{3.4}$$

Proof. The proof directly follows from Lemma 2. Since the adversary can not distinguish a tag from a random string under repeating nonces, she can not create a forgery. Indeed, suppose that an adversary B has advantage α of distinguishing Π^{-1} from the "always-invalid" oracle for a single query, i.e. she can guess the tag with probability α. Then B can distinguish Π from the random-bit oracle with the same probability, which is in turn bounded in Lemma 2. $\square$

3.1.4 Extra nonce

The extra nonce N_e is a function of P, K, and A. It invokes a sponge construction, which is proved indifferentiable from random oracle up to $2^{c/2}$ queries [5], where the parameter c is equal to $2k$ in PAEQ (and PPAE). The extra nonce function is thus pseudo-random and generates unique nonces unless it reaches the birthday bound of $2^{r/2}$ queries (this is smaller than the 2^k indifferentiability bound in this submission). Therefore, the following statement holds.

Proposition 1. *Let A be the adversary that makes q encryption queries (P, A) to the PPAE encryption oracle Π_K with extra nonce feature. Let $K \overset{\$}{\leftarrow} \mathcal{K}$ and $\mathcal{F} \overset{\$}{\leftarrow} \mathrm{Perm}(n)$. If, additionally, all the pairs (P, A) are distinct, then the advantage of the adversary to violate confidentiality or data integrity are bounded by values given in Theorems 1 and 2 or $\max(\frac{q^2}{2^{2k}}, \frac{q^2}{2^r})$, whatever is larger.*

3.2 Security of AESQ

In this section we discuss the security of the AESQ permutation. We expect that the reader is familiar with properties of the AES internal operations, and refer to [9] in case of questions.

3.2.1 Structure and decomposition

First we recall that one round of AES does not provide full diffusion as it mixes together only 32 bits. For instance, consider a column in the AES round that undergoes the MixColumns transformation. Before that its bytes have been permuted by ShiftRows and substituted by SubBytes; afterwards they are xored with a subkey and again go through SubBytes and ShiftRows. These operations can be grouped into a single 32-bit so-called "SuperSBox' parametrized with a 32-bit subkey. As a result, two AES rounds can be viewed as a layer SuperSubBytes of four 32-bit SuperSBoxes followed by ShiftRows and MixColumns, so that we can view AES-128 as a 5-round cipher with larger S-boxes.

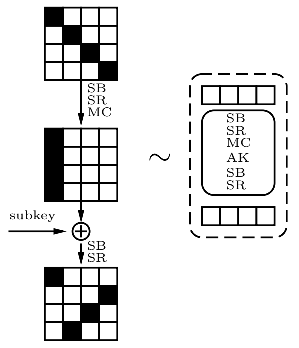

Figure 3.2: SuperSBox in AES.

The same strategy applies to the AESQ permutation. The two-round groups that process the registers A, B, C, D can be viewed as a single round with $16 = 4 \cdot 4$ parallel SuperSBoxes and a large linear transformation. This yields an R-round SPN permutation with 32-bit S-boxes out of the original $2R$-round one.

We can go further and view as many as four rounds of AESQ as a single round with MegaSBoxes (cf. MegaBoxes in [8]). Indeed, refer to Figure 3.3 and Algorithm 1. Consider a 128-bit register in round 2, for instance register A. Let us first look at its input. Going into backward direction, the columns spread to all the registers and then each column undergoes MixColumns without any influence from the other parts of the register. Then the values are shuffled by ShiftRows, and finally updated byte-wise by SubBytes. Hence we can recompute the 16 bytes at the beginning of round 1, even though they are located in different registers. Let us now look at the register A at the end of round 3. Its columns again spread to all the registers, and again undergo SubBytes and ShiftRows independently of the other register bytes. As a result, we can view as many as four rounds as a layer MegaSubBytes of four 128-bit MegaSBoxes followed by a MixColumns-based linear transformation, which we call MegaMixColumns. It has branch number 5, as it is exactly a set of MixColumns transformations with reordered inputs and outputs. Note that this decomposition must start with an odd round, and does not work for even rounds.

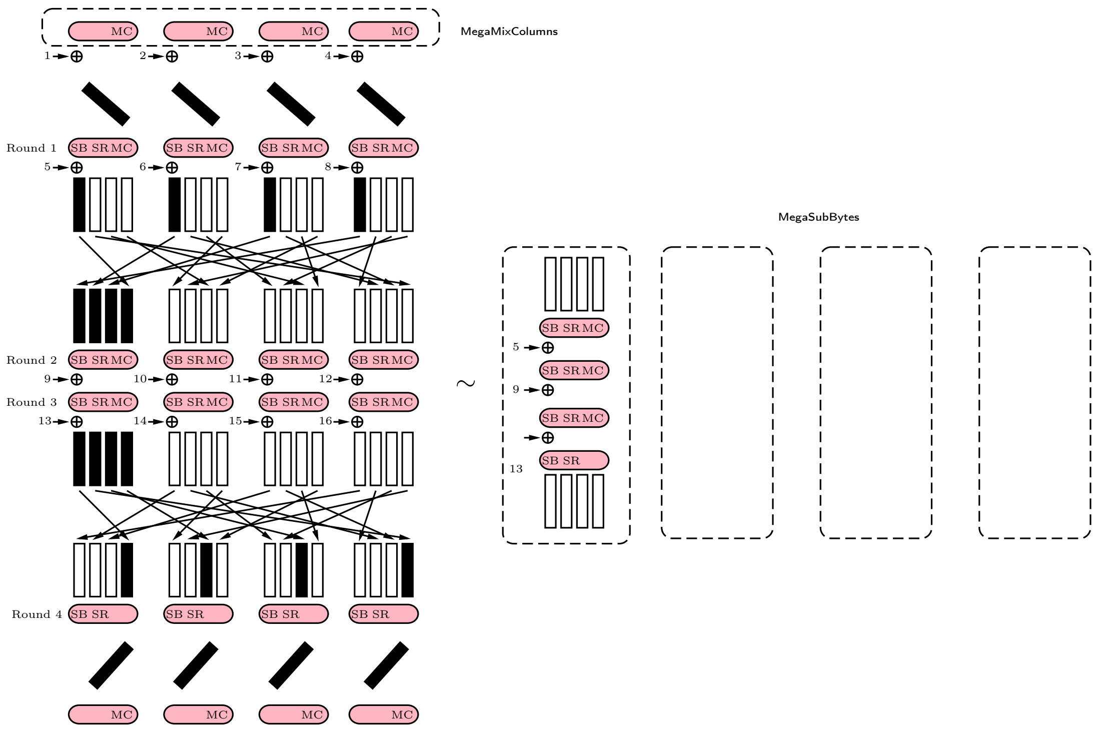

Figure 3.3: MegaSBox in AESQ.

3.2.2 Analysis of permutations in the attack context

Only a few permutations as a single and secure object have been designed for the use in practical constructions. The most well-known is the Keccak 1600-bit permutation, which is used in the Keccak/SHA-3 hashing algorithm; the others are used in the SHA-3 competitors: CubeHash [4], Grostl [11], JH [17]. It is worth noticing that a permutation per se can not be formally defined "secure". The best we can make is an informal statement like the 2^l *"flat sponge" claim* [6], which basically states that no attack with complexity below 2^l and specific for the particular permutation exists. The parameter l is used in defining the capacity parameters in sponge functions and in fact measures the designers' confidence.

In our case we claim $l = 256$ or the 256-bit security of AESQ against all attacks. In order to support our claim, we look at the existing attacks on permutation-based designs and check if they apply to AESQ.

Collision attacks. We first consider collision attacks on sponge-based hash functions. The collision attacks on the reduced Keccak [10] strongly rely on high-probability differential trails [16], and only add a couple of rounds over their length with the help of message-modification techniques. The so-called internal-differential attack, while exploiting similarities within the internal state, is also limited by the propagation of difference generated by the round constants. Hence to prevent these attacks we have to demonstrate the absence of high-probability differential trails for a high number of rounds.

Let us now consider compression functions based on permutations. For example, Grostl uses functions

$$P(x \oplus y) \oplus Q(y) \oplus x \quad \text{and} \quad x \oplus P(x),$$

where P and Q are AES-based permutations. The main strategy in collision attacks [14, 12] is the construction of a truncated differential trail with low input and output Hamming weight. Then the conforming inputs are found with the rebound attack and are tested for a collision.

Preimage attacks. The preimage attacks on sponge-based hash functions have been also based on the differential properties of the permutation. As long as a differential generated by message difference ΔM has high probability in some output bits, it can be used to speed up the preimage search [15]. There are also generic methods that can save a factor of several bits by exploiting incomplete diffusion in the final rounds, but we note that their complexity can not be reduced much. The invariant attacks [2] do not apply because of round constants.

Preimage attacks on Grostl are based on the meet-in-the-middle framework. Whereas it is difficult to formalize the necessary conditions for these attacks to work, we notice that the number of rounds attacked with meet-in-the-middle/bicliques is smaller compared to the rebound attack (even though the attack goals are distinct). Hence it should be sufficient to protect against rebound attacks.

Other attacks. A more generic set of attacks are given by the CICO and multi-CICO problems [6], which require the attacker to find one or more (input, output) pair conforming to certain bit conditions. There is no comprehensive treatment of these attacks, but they seem to be limited by twice the number of rounds needed for full diffusion.

Consider an 8-round version F of AESQ, for instance, that $F(X) = Y$, and we know the last 384 bits of both X and Y. How difficult is it to restore X and Y? The last 384 bits mean three registers B, C, D. Hence, we can compute registers B, C, D through rounds 0,1 and 6,7. Then we know 96 bits of each register in the beginning of round 2 and in the end of round 5. By a simple meet-in-the-middle attack we match between rounds 3 and 4 and recover X and Y with complexity 2^{32}. However, we did not manage to extend this technique to more rounds.

3.2.3 Possible attacks

Differential analysis of AESQ. Let us evaluate the differential properties of AESQ. We are backed up with the analysis of AES by Daemen and Rijmen. They prove the following lemma[1].

Lemma 3. *Any differential trail over 4 AES rounds has at least 25 active S-boxes.*

We have demonstrated in Section 3.2.2 that four rounds of AESQ can be viewed as a layer of 128-bit S-boxes and a large linear transformation, which has branch number 5. The 128-bit MegaSBoxes are exactly four-rounds of AES without the key additions and the last MixColumns. Hence each active MegaSBox has at least 25 active S-boxes.

The branch number of MegaMixColumns implies that eight AESQ rounds starting from an odd round have at least 5 active MegaSBoxes, and hence 125 active S-boxes. Therefore, nine rounds starting from an even round (as required by the permutation) have at least 126 active S-boxes, as the first round must have at least one. More generally, $8R + 1$ rounds must have at least $125R + 1$ active S-boxes. In turn, the last round has at maximum 64 active S-boxes, so the lower bound for $8R$ rounds is $125R - 63$ active S-boxes. The other bounds are given in Table 3.1.

Rounds	Active S-boxes	Rounds	Active S-boxes
2	5	8R+2	125R +2
4	10	8R+4	125R +7
6	27	8R+6	125R + 27
8R	125R-63	8R+8	125R+68

Table 3.1: Minimal number of active S-boxes in the AESQ permutation.

The table implies that a 20-round trail has at least 257 active S-boxes. Given that the maximal differential probability of each S-box is 2^{-6}, the vast majority of these trails have differential probability equal to zero. The effect of clustering these trails into a single differential and the resulting probability is not completely studied, but the bound of maximum expected differential probability for 4 rounds of AES of 2^{-113} [13] indicates that these values should be very low for 16 rounds of AESQ and more unless the round constants accidentally admit a large number of conforming pair for a differential (the so called *height* parameter).

If we consider slide-like attacks, where differences between registers or states shifted by several rounds are considered, we obtain that the round constants generate one-byte difference in every round, so we can not hope for high probability differentials in this case.

Rebound attacks. Rebound attacks aim to construct a conforming input for a truncated differential trail with a little amortized cost. The attack consists of two phases: inbound phase, where a conforming input is constructed for the low-probability part in the meet-in-the-middle manner, and the outbound

[1]Actually, they prove a more general theorem for SPN ciphers but the AESQ permutation does not satisfy its assumptions.

phase, where the differences are traced through the rest of the primitive without the control of the cryptanalyst.

Again, given the decomposition of AESQ into MegaSBoxes, we can generate a full difference distribution table for MegaSBoxes with the complexity 2^{256}. Consider MegaSBoxes in rounds 5–8 of the permutation. Let us select an active S-box in rounds 4 and 9, and select a one-byte difference in them. Then we apply MegaMixColumns and obtain input/output differences for MegaSBoxes. We obtain actual state values and then recompute the difference in the outer directions. Active S-boxes in round 4 activates one MegaSBox in rounds 1–4, and then activates all the S-boxes of round 0. At the other end, the active S-box in round 9 activates a MegaSBox in rounds 9–12, which activates all S-boxes in round 13. This yields a clear distinguisher for 12 rounds of AESQ with complexity 2^{256}, and possibly some sort of distinguisher for 14 rounds, which should be compared to the distinguisher of the 8-round AES permutation.

Final choice. We take the 20-round AESQ permutation within PAEQ, which should provide a 256-bit security against all attacks. After the third-party cryptanalysis, if little improvement over our analysis will be shown, we might consider reducing the number of rounds to 16 for 80- and 128-bit security.

Chapter 4

Features

We offer PAEQ as an authenticated encryption scheme for heavyweight environments where AES operations are fast (e.g., on recent Intel/AMD CPUs). It allows very high security level up to 160 bits, which is conveniently equal to the key length, in contrast to AES-based AE modes, which maximally allow 64 bits of security due to the birthday phenomenon. We offer three primary sets of parameters: `paeq64`, `paeq80`, and `paeq128`, which provide 64, 80, and 128 bits of security, respectively, by having the key and the tag of same length. We consider `paeq80` and `paeq128` secure for all practical applications, and recommend `paeq64` for protecting less sensitive information.

PAEQ allows to encrypt and decrypt the data on the arbitrary number of subprocessors with tiny amount of shared memory. Those processors, threads, or other computation units may perform decryption and encryption of incoming blocks in any order.

PAEQ is based on the AES block cipher, and does not use any operations except those of AES. The mode of operation around the AES-based permutation AESQ uses only XORs and counter increments, and aims to be amongst easiest authenticated encryption modes to implement.

4.1 List of features

In this section we provide an extensive list of features of PAEQ. We have tried to make the mode as universal as possible, and to provide the users with almost every capability an authenticated encryption mode might have.

Key/nonce/tag length	We allow keys, nonces, and tags of arbitrary length, as long as they fit into the permutation, fulfill some minimal requirements, and constitute the integer number of bytes.
Performance	Depending on the key size, the encryption speed is about 7 cycles per byte on modern CPUs with permutation AESQ.
Security level	Depending on the key length and the permutation width, we support a range of security levels from 64 to 256 bits. For the permutation of width n bits we can use a key of about $(n/3 - 6)$ bits and get the same security level for both confidentiality and ciphertext integrity. Hence a permutation of 400 bits width already delivers a security level of 128 bits.
Security proof	Our mode is provably secure in the random permutation model, whereas the security proof is short and verifiable by the third parties.
Parallelism	Our scheme is fully parallelizable: all blocks of plaintext and associated data can be processed in parallel; only the last call of the permutation needs all the operations to finish.
Online processing	Our scheme is fully online, being able to process plaintext blocks or blocks of associated data as soon as they are ready without knowing the final length.

Patents	It is not patented, and we are not aware of any patent covering any part of the submission.
Tag update	If the tag is not truncated, then the last permutation call can be inverted given the key, and only two extra permutation calls are needed to encrypt and authenticate a new plaintext with one new block.
Inverse	The permutation is used in the forward direction only, with a sole exception of tag update, if this feature is needed.
Nonce misuse	If the nonce is reused, then the integrity is still provided. Additionally, a user may generate a nonce out of the key, the plaintext, and the associated data with a dedicated routine.

4.2 Advantages over other ciphers

The scheme has the following advantages over AES-GCM:

1. It provides higher security levels up to 160 bits, with the key length being equal to the security level.
2. Its mode of operation contain only simple operations as XOR, which allows for clear structure and easier and verifiable security proofs.
3. It preserves the integrity feature in the case of nonce misuse, and provides an extra feature of generating an input-based nonce in the environment where the nonce uniqueness can not be guaranteed.
4. It allows plaintexts and associated data as long as 2^{96} bytes, which in practice means no restrictions at all for the foreseeable future.

Compared to other AE designs, we highlight the following features of our scheme:

- Designs that use the AES cipher (COPA [1], COBRA, OTR) can not deliver a security level higher than 64 bits due to the birthday phenomena at the 128-bit AES state. In contrast, PAEQ easily brings the security level of 128 bits and higher.
- The extra nonce feature that mitigates a nonce-misuse environment does not come at the cost of hardening the mode, but rather is a dedicated routine that affects performance only when called.
- PAEQ achieves the maximum level of parallelism, as it can process incoming plaintext and associated data blocks in any order and needs only 512 bits of memory to store intermediate results (apart from the ciphertext). In contrast, COPA [1] needs to process all the preceding blocks to produce a ciphertext block.
- It is one of the very few schemes that allows the incremental tag update.
- It does not need the permutation inverse as long as the tag update feature is not used.
- Thanks to the permutation-based mode of operation, PAEQ uses only XOR and concatenation operators, which makes the design and security proof much easier to understand and less prone to bugs.

Chapter 5

Design rationale

5.1 Design of PPAE

When creating the new scheme, we pursued the following goals:

- Offer high security level, up to 128 bits, ideally equal to the key length,
- Make the mode of operation simple enough to yield compact and reliable security proofs.
- Deliver as many features as possible.

To achieve these goals, we decided to trade performance for clarity and verifiability.

Existing block ciphers were poor choice for these goals. They commonly have a 128-bit block, which almost inevitably results in the loss of security at the level of 2^{64} cipher calls. The 256-bit cipher Threefish could have been used, but the lack of cryptanalysis in the single-key model makes it a risky candidate. Our mode of operation would be also restricted to a single cipher.

Instead, we constructed a permutation-based mode, which takes a permutation of any width if it is at least twice as large as the key. This choice makes the scheme much more flexible, allows for variable key and nonce length, and simplifies the proof. The key update also becomes very easy. The downside of the permutation-based approach is that the security proof has to be devised in the random oracle/permutation model, and does not rely on the PRP assumption. This is inevitable, but the success of the sponge-based constructions tells that it is not necessarily a drawback.

For the encryption stage we have chosen an analogy with the CTR mode, so that we do not have to use the permutation inverse. It also allows us to truncate some parts of the intermediate variables. For the authentication stage we use a parallel permutation-based construction. It takes the yet unused secret input from the encryption stage, which provides pseudo-randomness.

It remains to choose a permutation. Initially we thought of using a family of permutations with different widths. Examples could be Keccak [6], Spongent [7], or Quark [3] permutations. However, the performance loss would be too high given two invocations of a permutation per plaintext block. Instead, we designed our own permutation which shows the best performance on modern CPUs. It can be used in other permutation-based constructions, e.g. the extended Even-Mansour cipher or the sponge construction.

5.2 Design of AESQ

When designing AESQ, we needed a permutation wide enough to accomodate 128-bit keys and nonces. The AES permutation would be too short, while AES-based permutations used in the SHA-3 context would be too large or not well optimized for AES instructions on modern CPUs.

We decided to run 4 AES states in parallel and regularly shuffle the state bytes. Since two AES rounds provide full diffusion, the shuffle should occur every two rounds. The shuffle operation should make each state to affect all four states, resembling the ShiftRows transformation in AES. The recent Intel processors, along with dedicated AES instructions, provide a set of instructions that interleave the 32-bit subwords of 256-bit registers. Those subwords are columns of the AES state, so we shuffled the columns. The shuffle function in this submission is one of the permutations that provide full diffusion and needs the minimal 8 number of processor instructions.

5.3 No weakness

The designers have not hidden any weaknesses in this cipher.

Chapter 6

Implementation

We provide a portable reference implementation of PAEQ in C++. Now it consists of a single file `encrypt.cpp`. It contains the following functions:

- `crypto_aead_encrypt` is the standard encryption routine.
- `crypto_aead_decrypt` is the standard decryption routine. It returns -1 as $\perp$.
- `crypto_aead_encrypt_no_nonce` performs encryption with the extra nonce option. It calls `GenerateNonce` and writes the output to the `npub` input parameter.
- `FPerm` applies the AESQ permutation to the 512-bit input.
- `GenerateNonce` generates the extra nonce out of the plaintext, the key, and the associated data.
- `AES_Round` applies the AES operations SubBytes, ShiftRows, and MixColumnsto the 128-bit state. This function is used within the AESQ permutation.
- `genKAT` generates key, nonce, plaintext, and associated data of given length in deterministic way, performs encryption and decryption, and writes all the inputs to the log file `out.log`. This function should help implementing PAEQ on other platforms.
- `gmul_o` is an auxiliary function that performs multiplication in $GF(256)$.
- `Init` initializes the multiplication table in $GF(256)$, which is used for AES operations. This function must be called before the first call of `FPerm` (it is done in encryption and decryption routines).
- `main` calls `genKAT`().
- `FPermAsm` is the assembler-optimized version of `FPerm`. It works on Windows platform only and is disabled for portability.

6.1 Performance of AESQ

Our permutation is best suited for the last Intel and AMD processors equipped with special AES instructions. Of the AES-NI instruction set, we use only AESENC instruction that performs a single round of encryption. More precisely, AESENC(S, K) applies ShiftRows, SubBytes, and MixColumnsto S and then XORs the subkey K. In our scheme the subkey is a round constant. For two rounds of AESQ, we use the following instructions:

- 8 `aesenc` instructions for AES round calls;
- 8 `vpunpckhdq` instructions to permute the state columns between registers;
- 8 `vpaddq` instructions to update round constants.

This gives a total of 24 instructions per 64 bytes of the state in two rounds. This means that the full $2R$-round AESQ permutation needs $24R$ instructions, and theoretically may run in $3R/8$ cycles per byte if properly pipelined. This speed may be achieved in practice, since the throughput of AES instructions is 1 cycle starting from the Sandy Bridge architecture (2010), and the `vpaddq` instruction may be even faster. The following strategy gives us the best performance on the newest Haswell architecture:

- Process two 512-bit states in parallel in order to mitigate the latency of 8 cycles of the AESENC instruction. Store each state into 4 `xmm` registers.
- Store round constants in 4 `xmm` registers and use them in the two parallel computations.
- Use one `xmm` register to store the constant 1 to update the round constants, and one temporary register for the Shuffle operation.

We note that on the earlier Westmere architecture the AESENC instruction has latency 6 and throughput 2, hence the AES round calls should be interleaved with mixing instructions conducted on several parallel states.

We have made our own experiments and concluded that the 16-round version of AESQ ($R = 8$) runs at 2.4 cycles per byte on a Haswell-family CPU, whereas the 20-round version runs at 3 cycles per byte. This can be compared to the speed of the Keccak-1600 permutation. As eBASH reports, on a Haswell CPU the Keccak hash function with rate 1088 runs at 10.6 cpb, which implies that the full 1600-bit permutation runs at approximately $\frac{10.6 \cdot 1088}{1600} = 7.2$ cpb. Therefore, the 20-round AESQ is 2.5 times as fast as Keccak-1600.

6.2 Performance of PAEQ

The current submission provides an optimized implementation of AESQ only, but not of PAEQ. To achieve the highest throughput of AESQ calls, an optimized implementation should call two AESQ instances in parallel. This would allow for the total throughput 1. Given that AESQ runs at 3 cpb, we conclude that `paeq64` should run at about 7 cpb, and `paeq128` — at 8 cpb.

Chapter 7

Intellectual property

The authors are aware of no known patents, patent applications, planned patent applications, and other intellectual-property constraints relevant to use of the cipher.

If this information changes, the submitter will promptly (and within at most one month) announce these changes on the `crypto-competitions` mailing list.

Chapter 8

Consent

The submitter hereby consents to all decisions of the CAESAR selection committee regarding the selection or non-selection of this submission as a second-round candidate, a third-round candidate, a finalist, a member of the final portfolio, or any other designation provided by the committee. The submitter understands that the committee will not comment on the algorithms, except that for each selected algorithm the committee will simply cite the previously published analyses that led to the selection of the algorithm. The submitter understands that the selection of some algorithms is not a negative comment regarding other algorithms, and that an excellent algorithm might fail to be selected simply because not enough analysis was available at the time of the committee decision. The submitter acknowledges that the committee decisions reflect the collective expert judgments of the committee members and are not subject to appeal. The submitter understands that if he disagrees with published analyses then he is expected to promptly and publicly respond to those analyses, not to wait for subsequent committee decisions. The submitter understands that this statement is required as a condition of consideration of this submission by the CAESAR selection committee.

Bibliography

[1] Elena Andreeva, Andrey Bogdanov, Atul Luykx, Bart Mennink, Elmar Tischhauser, and Kan Yasuda. Parallelizable and authenticated online ciphers. In *ASIACRYPT'13*, volume 8269 of *Lecture Notes in Computer Science*. Springer, 2013.

[2] Jean-Philippe Aumasson, Eric Brier, Willi Meier, María Naya-Plasencia, and Thomas Peyrin. Inside the hypercube. In *ACISP*, volume 5594 of *Lecture Notes in Computer Science*, pages 202–213. Springer, 2009.

[3] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and María Naya-Plasencia. Quark: A lightweight hash. In *CHES'10*, volume 6225 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2010. `https://131002.net/quark/quark_full.pdf`.

[4] Daniel J. Bernstein. CubeHash specification (2.b.1). Submission to NIST (Round 2), 2009.

[5] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. On the indifferentiability of the sponge construction. In *EUROCRYPT'08*, volume 4965 of *Lecture Notes in Computer Science*, pages 181–197. Springer, 2008.

[6] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. The Keccak reference, version 3.0, 2011. `http://keccak.noekeon.org/Keccak-reference-3.0.pdf`.

[7] Andrey Bogdanov, Miroslav Knezevic, Gregor Leander, Deniz Toz, Kerem Varici, and Ingrid Verbauwhede. Spongent: A lightweight hash function. In *CHES'11*, volume 6917 of *Lecture Notes in Computer Science*, pages 312–325. Springer, 2011.

[8] Joan Daemen, Mario Lamberger, Norbert Pramstaller, Vincent Rijmen, and Frederik Vercauteren. Computational aspects of the expected differential probability of 4-round aes and aes-like ciphers. *Computing*, 85(1-2):85–104, 2009.

[9] Joan Daemen and Vincent Rijmen. *The Design of Rijndael. AES — the Advanced Encryption Standard*. Springer, 2002.

[10] Itai Dinur, Orr Dunkelman, and Adi Shamir. New attacks on Keccak-224 and Keccak-256. In *FSE'12*, volume 7549 of *Lecture Notes in Computer Science*, pages 442–461. Springer, 2012.

[11] Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schlffer, and Sren S. Thomsen. Grøstl – a SHA-3 candidate. Submission to NIST (Round 3), 2011.

[12] Kota Ideguchi, Elmar Tischhauser, and Bart Preneel. Improved collision attacks on the reduced-round Grøstl hash function. In *ISC*, volume 6531 of *LNCS*, pages 1–16. Springer, 2010.

[13] Liam Keliher and Jiayuan Sui. Exact maximum expected differential and linear probability for 2-round Advanced Encryption Standard (AES). *IACR Cryptology ePrint Archive*, 2005:321, 2005.

[14] Florian Mendel, Christian Rechberger, Martin Schlffer, and Sren S. Thomsen. Rebound attacks on the reduced Grøstl hash function. In *CT-RSA*, volume 5985 of *LNCS*, pages 350–365. Springer, 2010.

[15] Pawel Morawiecki, Josef Pieprzyk, Marian Srebrny, and Michal Straus. Preimage attacks on the round-reduced Keccak with the aid of differential cryptanalysis. Cryptology ePrint Archive, Report 2013/561, 2013. `http://eprint.iacr.org/`.

[16] María Naya-Plasencia, Andrea Röck, and Willi Meier. Practical analysis of reduced-round keccak. In *INDOCRYPT'11*, volume 7107 of *Lecture Notes in Computer Science*, pages 236–254. Springer, 2011.

[17] Hongjun Wu. The hash function JH. Submission to NIST (round 3), 2011.