The CBEAMr1 Authenticated Encryption Algorithm

First Round CAESAR Competition Submission Document

Designer and Submitter

Markku-Juhani O. Saarinen
mjos@item.ntnu.no

March 15, 2014

For updates and further information:

http://www.cbeam.mx

NTNU
Norwegian University of Science and Technology

Contents

Preface

This is a specification for CBEAMr1, a first round submission to the CAESAR competition. The document has been written to strictly adhere to the structure suggested in the CAESAR call for submissions:

`http://competitions.cr.yp.to/caesar-call.html`

Therefore this particular document may not be easily accessible to someone who is not a professional cryptographer, even though I try to illuminate key parts with examples.

My two CAESAR submissions, CBEAM and STRIBOB [32], utilize the same BLNK Sponge padding mechanism, and relevant sections appear almost identical (apart from some very important parameter selections). CAESAR submission documents need be effectively self-contained when it comes to specifying the AEAD mode.

However, the sponge permutations themselves have almost nothing in common and are based on entirely different design paradigms.

- **CBEAM** is based on rotation-invariant ϕ functions, feeble Boolean one-wayness, and other novel ideas. CBEAM is completely original work. Its design is geared towards limited-resource ("lightweight") medium-security applications.
- **STRIBOB** uses traditional S-Boxes and MDS matrices, and is therefore a close relative of the AES Block Cipher. STRIBOB gets additional security assurance from its even closer relationship with the new Russian hash standard, Streebog. The design is geared towards general high-security applications.

Some of the material used in these submissions has recently appeared in technical conferences or has been submitted to such [29, 30, 31].

This is the version 1.20140315200000 of this document. We urge the reader to check for updates, revisions, and reference data at:

`http://www.cbeam.mx`

If you find bugs, typos, obvious security blunders, or clever cryptanalytic attacks, I would be very interested to hear about that. My e-mail address can be found at the front page.

Cheers and have fun,

- **Markku**, the fjords

Chapter 1

Specification

1.1 CBEAMr1 Family and Parameters

CBEAM is an algorithm for Authenticated Encryption with Associated Data (AEAD). CBEAM accepts almost arbitrary ranges for its input parameters; however for CAESAR we propose only one concrete parameter set "`cbeam128r1`" as follows:

Secret key size	128 bits	`CRYPTO_KEYBYTES 16`
Secret sequence number	Unused (but supported)	`CRYPTO_NSECBYTES 0`
Public sequence number (nonce)	64 bits	`CRYPTO_NPUBBYTES 8`
Authentication tag (message expansion)	64 bits	`CRYPTO_ABYTES 8`

We first give a mathematical description of the cryptographic permutation mx in Section 1.2, together with some example computations, and then describe its use to implement Authenticated Encryption in Section 1.3, followed by a trace of full AEAD computation in 1.4.

Note on Conventions

A little-endian convention is used throughout. We use C-style zero-based indexing for arrays and matrices. For transport and storage, and for use by the BLNK mode, the state must be accessible as a byte sequence as described here. We index the 256-bit state **s** interchangeably as:

- A 16×16 - bit matrix $s[\,0 \cdots 15\,][\,0 \cdots 15\,]$.
- A vector of 16-bit words $s_w[\,0 \cdots 15\,]$ with $s_w[\,i\,] = \sum_{j=0}^{15} 2^j s[\,i\,][\,j\,]$.
- Four quadwords $s_q[\,0 \cdots 3\,]$ with $s_q[\,i\,] = \sum_{j=0}^{3} 2^{16j} s_w[\,4i + j\,]$.
- Byte sequence $s_b[\,0 \cdots 31\,]$ is interpreted as $s_w[\,i\,] = s_b[\,2i\,] + 2^8 s_b[\,2i + 1\,], 0 \leq i \leq 15$.

Modulo 16 arithmetic in indexing is equivalent to logical masking with `0xF`; a mod 16 is always in the range $0, 1, \cdots, 15$.

We give a rather sparse logic-base description here. This description is poorly suited for direct software implementation. Please see Chapter 4 for implementation notes on a variety of platforms.

1.2 Mixing Function mx

The basic building block of CBEAM is the mx mixing function, which is a bijective transform on a 256-bit state variable **s**. Six rounds of mx make up $\mathsf{mx}^6 = \pi$, the core cryptographic permutation of CBEAM.

The mixing function mx is composed of (in order of execution): exclusive-or addition of a round constant rc^r, bit matrix transpose, linear mixing λ, and nonlinear mixing ϕ. For rounds $r = 0, 1, \ldots 5$ we iterate $\mathbf{s} = \mathsf{mx}^r(\mathbf{s})$ on the state:

$$\mathsf{mx}^r(\mathbf{s}) = \phi(\lambda((\mathbf{s} \oplus \mathsf{rc}^r)^T)) = (\phi \circ \lambda^T)(\mathbf{s} \oplus \mathsf{rc}^r). \tag{1.1}$$

To evaluate $\pi = \mathsf{mx}^6$ we compute six rounds $r = 0 \ldots 5$ by iterating these three steps:

1.2.1 Round Constant rc^r

Let the individual round bits be $r = 4r_2 + 2r_1 + r_0$. We have $s'[\,i\,][\,j\,] = s[\,i\,][\,j\,]$ for all $0 \le i, j \le 15$ except the following:

$$\begin{aligned}
s'[\,0\,][\,0\,] &= s[\,0\,][\,0\,] \oplus (r_0 \wedge \neg r_1) & s'[\,8\,][\,2\,] &= s[\,8\,][\,2\,] \oplus (r_0 \wedge r_1) \\
s'[\,1\,][\,0\,] &= s[\,1\,][\,0\,] \oplus (r_0 \wedge r_2) & s'[\,10\,][\,2\,] &= s[\,10\,][\,2\,] \oplus r_0 \\
s'[\,3\,][\,0\,] &= s[\,3\,][\,0\,] \oplus r_0 & s'[\,11\,][\,2\,] &= s[\,11\,][\,2\,] \oplus (r_0 \wedge r_2) \\
s'[\,4\,][\,1\,] &= s[\,4\,][\,1\,] \oplus r_0 & s'[\,13\,][\,3\,] &= s[\,13\,][\,3\,] \oplus r_0 \\
s'[\,5\,][\,1\,] &= s[\,5\,][\,1\,] \oplus (r_0 \wedge \neg r_1) & s'[\,14\,][\,3\,] &= s[\,14\,][\,3\,] \oplus (r_0 \wedge r_1) \\
s'[\,6\,][\,1\,] &= s[\,6\,][\,1\,] \oplus (r_0 \wedge r_2) & s'[\,15\,][\,3\,] &= s[\,15\,][\,3\,] \oplus (r_0 \wedge r_2).
\end{aligned} \tag{1.2}$$

Finally $\mathbf{s} \leftarrow \mathbf{s}'$. Observe that the round constants are active only on odd rounds (when $r_0 = 1$).

1.2.2 Linear transform λ^T

Let $\mathbf{s}' = \lambda(\mathbf{s}^T)$ for $0 \le i, j \le 15$:

$$s'[\,i\,][\,j\,] = s[\,(j+4) \bmod 16\,][\,i\,] \oplus s[\,(j+8) \bmod 16\,][\,i\,] \oplus s[\,(j+12) \bmod 16\,][\,i\,]. \tag{1.3}$$

Finally $\mathbf{s} \leftarrow \mathbf{s}'$. λ^T transposes the matrix (note swapped i and j) and performs a nibble parity operation.

1.2.3 Nonlinear transform ϕ

We define $\mathbf{s}' = \phi(\mathbf{s})$ for $0 \le i, j \le 15$ as:

$$\begin{aligned} s'[\,i\,][\,j\,] = \phi_5\big(\, & s[\,i\,][\,j\,],\, s[\,i\,][\,(j-1) \bmod 16\,],\; s[\,i\,][\,(j-2) \bmod 16\,], \\ & s[\,i\,][\,(j-3) \bmod 16\,],\; s[\,i\,][\,(j-4) \bmod 16\,]\,\big), \end{aligned} \tag{1.4}$$

where ϕ_5 is defined the following Algebraic Normal Form (ANF) polynomial in $\mathbb{Z}_2$:

$$\begin{aligned} \phi_5(x_0, x_1, x_2, x_3, x_4) = {} & x_0x_1x_3x_4 + x_0x_2x_3 + x_0x_1x_4 + x_1x_2x_3 + x_2x_3x_4 + {} \\ & x_0x_3 + x_1x_3 + x_2x_3 + x_2x_4 + x_3x_4 + x_1 + x_3 + x_4. \end{aligned} \tag{1.5}$$

Finally $\mathbf{s} \leftarrow \mathbf{s}'$. The truth table for ϕ_5 is given below:

x_0	x_1	x_2	x_3	x_4	ϕ_5
0	0	0	0	0	0
1	0	0	0	0	0
0	1	0	0	0	1
1	1	0	0	0	1
0	0	1	0	0	0
1	0	1	0	0	0
0	1	1	0	0	1
1	1	1	0	0	1
0	0	0	1	0	1
1	0	0	1	0	0
0	1	0	1	0	1
1	1	0	1	0	0
0	0	1	1	0	0
1	0	1	1	0	0
0	1	1	1	0	1
1	1	1	1	0	1

x_0	x_1	x_2	x_3	x_4	ϕ_5
0	0	0	0	1	1
1	0	0	0	1	1
0	1	0	0	1	0
1	1	0	0	1	1
0	0	1	0	1	0
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	0	1	0
0	0	0	1	1	1
1	0	0	1	1	0
0	1	0	1	1	1
1	1	0	1	1	0
0	0	1	1	1	0
1	0	1	1	1	0
0	1	1	1	1	1
1	1	1	1	1	1

1.2.4 Example of $\pi = \mathsf{mx}^6$ Computation

We compute $\pi(V) = \mathsf{mx}^6(V)$ where V is given by the following byte sequence:

```
23 01 34 12 45 23 56 34 67 45 89 57 89 67 9A 78 AB 89 BC 9A CD AB DE BC EF CD F0 DE 01 EF 12 F0
```

Since CBEAM follows the little-endian convention, when loaded up to the internal state V is accessible as 16- and 64-bit words in the following way:

$$V_w[\,0 \cdots 15\,] = (\ \texttt{0123 1234 2345 3456 4567 5789 6789 789A 89AB 9ABC ABCD BCDE CDEF DEF0 EF01 F012}\ )$$
$$V_q[\,0 \cdots 3\,] = (\ \texttt{3456234512340123 789A678957894567 BCDEABCD9ABC89AB F012EF01DEF0CDEF}\ )$$

The corresponding binary matrix and its transpose (recall $\mathsf{rc}^0 = 0$, hence no key addition) are:

$$v[\,0 \cdots 15\,][\,0 \cdots 15\,] = \begin{pmatrix}
1&1&0&0&0&1&0&0&1&0&0&0&0&0&0&0\\
0&0&1&0&1&1&0&0&0&1&0&0&1&0&0&0\\
1&0&1&0&0&0&1&0&1&1&0&0&0&1&0&0\\
0&1&1&0&1&0&1&0&0&0&1&0&1&1&0&0\\
1&1&1&0&0&1&1&0&1&0&1&0&0&0&1&0\\
1&0&0&1&0&0&0&1&1&1&1&0&1&0&1&0\\
1&0&0&1&0&0&0&1&1&1&1&0&0&1&1&0\\
0&1&0&1&1&0&0&1&0&0&0&1&1&1&1&0\\
1&1&0&1&0&1&0&1&1&0&0&1&0&0&0&1\\
0&0&1&1&1&1&0&1&0&1&0&1&1&0&0&1\\
1&0&1&1&0&0&1&1&1&1&0&1&0&1&0&1\\
0&1&1&1&1&0&1&1&0&0&1&1&1&1&0&1\\
1&1&1&1&0&1&1&1&1&0&1&1&0&0&1&1\\
0&0&0&0&1&1&1&1&0&1&1&1&1&0&1&1\\
1&0&0&0&0&0&0&0&1&1&1&1&0&1&1&1\\
0&1&0&0&1&0&0&0&0&0&0&0&1&1&1&1
\end{pmatrix}
\quad
v^T = \begin{pmatrix}
1&0&1&0&1&1&1&0&1&0&1&0&1&0&1&0\\
1&0&0&1&1&0&0&1&1&0&0&1&1&0&0&1\\
0&1&1&1&1&0&0&0&0&1&1&1&1&0&0&0\\
0&0&0&0&0&1&1&1&1&1&1&1&1&0&0&0\\
0&1&0&1&0&0&0&1&0&1&0&1&0&1&0&1\\
1&1&0&0&1&0&0&0&1&1&0&0&1&1&0&0\\
0&0&1&1&1&0&0&0&0&0&1&1&1&1&0&0\\
0&0&0&0&0&1&1&1&1&1&1&1&1&1&0&0\\
1&0&1&0&1&1&1&0&1&0&1&0&1&0&1&0\\
0&1&1&0&0&1&1&0&0&1&1&0&0&1&1&0\\
0&0&0&1&1&1&1&0&0&0&0&1&1&1&1&0\\
0&0&0&0&0&0&0&1&1&1&1&1&1&1&1&0\\
0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1\\
0&0&1&1&0&0&1&1&0&0&1&1&0&0&1&1\\
0&0&0&0&1&1&1&1&0&0&0&0&1&1&1&1\\
0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&1
\end{pmatrix}$$

$$v^T_w = (\ \texttt{5575 9999 1E1E 1FE0 AA8A 3313 3C1C 3FE0 5575 6666 7878 7F80 AAAA CCCC F0F0 FF00}\ )$$

The λ and ϕ transforms complete the first round (note that λ has an exceptionally small effect on this particular vector with repeated nibbles).

$$\begin{aligned}
\lambda(v^T) &= (\ \texttt{7757 9999 1E1E 1FE0 88A8 1131 1E3E 1DC2 7757 6666 7878 7F80 AAAA CCCC F0F0 FF00}\ )\\
\phi(\lambda(v^T)) &= (\ \texttt{88A8 3333 BDBD BFC1 DD5D B87B BF7D A3B5 88A8 CCCC F6F6 FF06 5555 9999 EDED FE0D}\ )\\
&= \mathsf{mx}(V).
\end{aligned}$$

For second round the first step is constant addition (since this is an odd round), followed by the same steps as before. We set $\mathbf{s} \leftarrow \mathsf{mx}(V)$ and update it:

$$\begin{aligned}
\mathsf{rc}^1 &= (\ \texttt{0001 0000 0000 0001 0002 0002 0000 0000 0000 0000 0004 0000 0000 0008 0000 0000}\ )\\
\mathbf{s} \leftarrow \mathsf{s} \oplus \mathsf{rc}^1 &= (\ \texttt{88A9 3333 BDBD BFC0 DD5F B879 BF7D A3B5 88A8 CCCC F6F2 FF06 5555 9991 EDED FE0D}\ )\\
\mathbf{s} \leftarrow \mathbf{s}^T &= (\ \texttt{F0F7 0C12 DAD4 C375 34F6 45E7 5678 678D 78DE 8CCA DE5C EB7D BC7E CCEE DE10 EFFD}\ )\\
\mathbf{s} \leftarrow \lambda(\mathbf{s}) &= (\ \texttt{8780 F3ED 343A 1EA8 DA18 CD6F 9AB4 23C9 B412 AEE8 74F6 1482 5290 CCEE FC32 DCCE}\ )\\
\mathbf{s} \leftarrow \phi(\mathbf{s}) &= (\ \texttt{6F0D E713 4B47 B151 25BD 929F 2540 7780 4985 511D 818C A135 8426 9911 FB65 3991}\ )\\
\mathbf{s} &= \mathsf{mx}^2(V).
\end{aligned}$$

Intermediate values for the rest of the rounds and round constants (where nonzero) are:

$$\begin{aligned}
\mathsf{mx}^3(V) &= (\ \texttt{E50C EAE4 07F3 B08A 6476 2138 D90D F629 3919 3071 1E59 1458 DEEC 15F3 96DF 1FB2}\ )\\
\mathsf{rc}^3 &= (\ \texttt{0000 0000 0000 0001 0002 0000 0000 0000 0004 0000 0004 0000 0000 0008 0008 0000}\ )\\
\mathsf{mx}^4(V) &= (\ \texttt{8922 B751 6648 0EED C285 89E5 2DFC DBBF 4310 77FA 3494 7F13 47D9 6DD3 1E59 E502}\ )\\
\mathsf{mx}^5(V) &= (\ \texttt{2CA0 67B3 4F96 0A46 B209 AC7E 5C64 A125 CF7C B46F EB8A FAED 1130 934D CC02 0D67}\ )\\
\mathsf{rc}^5 &= (\ \texttt{0001 0001 0000 0001 0002 0002 0002 0000 0000 0000 0004 0004 0000 0008 0000 0008}\ )\\
\mathsf{mx}^6(V) &= (\ \texttt{5432 281E B184 9481 AAF0 C9BE A028 4C79 4B69 53BF 53C0 CFE8 8839 9D2A 89E3 1300}\ )
\end{aligned}$$

1.3 BLNK Sponge Mode and Padding

BLNK ("Blink") is a general and highly flexible Sponge mode of operation modified from the padding used in the original BLINKER [29] lightweight protocol.

In this section we describe only how it is used specifically in the `CBEAM128r1` Authenticated Encryption with Associated Data (AEAD) algorithm, ignoring many of its more advanced features.

Sponge functions in BLNK mode are characterized by the parameters permutation size b, rate r, and capacity c. These quantities are related by $b = r + c + \delta$, where:

b	State size. π has $b = 256$ bits.
r	Data rate or block size. $r = 64$ bits.
c	Capacity, the amount of secret information in the state. $c = b - \delta$ bits.
δ	Capacity consumed by padding. For `CBEAM128r1` we can bound this to $\delta < 2$ bits.

Furthermore, we fix the key size to $k = 128$ bits and the authentication tag to $t = 64$ bits. Authentication tags are contained in a ciphertext block.

1.3.1 BLNK Block Operations

We define four basic sponge operations for data absorption, squeezing, encryption, and decryption. Each one performs an operation on n bytes in a data domain specified by a single-byte padding argument pad, invoking the Sponge permutation π a total of $\max(\lceil n/8 \rceil, 1)$ times.

The four basic operations are:

$\mathsf{put}(\ D[\,n\,],\ pad\ )$	Absorb n bytes of data D into the state.
$D[\,n\,] \leftarrow \mathsf{get}(\ n,\ pad\ )$	Squeeze out n bytes of data D from the state.
$C[\,n\,] \leftarrow \mathsf{enc}(\ P[\,n\,],\ pad\ )$	Encrypt n bytes of plaintext P to ciphertext C.
$P[\,n\,] \leftarrow \mathsf{dec}(\ C[\,n\,],\ pad\ )$	Decrypt n bytes of data ciphertext C to plaintext C.

In the following generic pseudocode $\mathsf{op} \in \{\mathsf{put}, \mathsf{get}, \mathsf{enc}, \mathsf{dec}\}$ and $V[\,8\,]$ is the state.

```
1:  $i \leftarrow 0$                                      state index, initialized to first byte
2:  for $j = 0$ to $n - 1$ do
3:      if $i = 8$ then
4:          $V[8] \leftarrow V[8] \oplus$ BLNK_END $\oplus pad$     full block padding with block end marker
5:          $V \leftarrow \pi(V)$                            cryptographic permutation
6:          $i \leftarrow 0$                                 zero index
7:      end if
8:      if op = put then
9:          $V[i] \leftarrow V[i] \oplus D[j]$                   XOR input data to the state
10:     else if op = get then
11:         $D[j] \leftarrow V[i]$                           simply save the data
12:     else if op = enc then
13:         $C[j] \leftarrow V[i] \oplus P[j]$                   encrypt as in a stream cipher
14:         $V[i] \leftarrow C[j]$                           store ciphertext in state
15:     else if op = dec then
16:         $P[j] \leftarrow V[i] \oplus C[j]$                   decrypt as in a stream cipher
17:         $V[i] \leftarrow C[j]$                           store ciphertext in state
18:     end if
19:     $i \leftarrow i + 1$                                 advance block index
20: end for
21: $V[i] \leftarrow V[i] \oplus$ BLNK_END                     end marker (note: $i = 8$ possible)
22: $V[8] \leftarrow V[8] \oplus$ BLNK_FIN $\oplus pad$          final padding
23: $V \leftarrow \pi(V)$                                    final cryptographic permutation
```

The byte constants and padding argument *pad* made up as a combination of some of these byte values:

Flag name	Value	Padding bit or Domain identifier
`BLNK_END`	`0x01`	Padding marker bit
`BLNK_FIN`	`0x02`	Data element final block marker bit
`BLNK_KEY`	`0x10`	Secret key (in)
`BLNK_NPUB`	`0x20`	Public sequence number (in)
`BLNK_NSEC`	`0x30`	Secret sequence number (in / out)
`BLNK_AAD`	`0x40`	Authenticated Associated Data (in)
`BLNK_MSG`	`0x50`	Confidential Message Payload (in/out)
`BLNK_MAC`	`0x60`	Message Authentication Code (out)

1.3.2 The CAESAR encrypt() and decrypt() AEAD API

Input and output parameters to the encryption and decryption primitives are given below. Each one of these is used as a C-style zero-indexed byte vector in the descriptions that follow. Furthermore, $V[\,0\cdots 31\,]$ is the 32-byte internal state of CBEAM .

$K[\,16\,]$	Secret key of $k = 128$ bits, or 16 bytes.
$N[\,8\,]$	A 64-bit public sequence number or nonce for the message. Only integrity is protected for this data and the contents are not part of ciphertext.
$A[\,a\,]$	Associated Authenticated data, $0 \le a$ bytes. Only integrity is protected for this data and the contents are not part of ciphertext. If unused, set $a = 0$.
$P[\,n\,]$	Plaintext payload, $0 \le n$ bytes. Integrity and confidentiality is protected for this data.
$C[\,n+8\,]$	Ciphertext, $8 \le n+8$ bytes. Integrity and confidentiality is protected for this data.

Pseudocode for implementing standard AEAD API encryption:

```
$C[\,n+8\,] \leftarrow$ encrypt( $K[\,16\,]$, $N[\,8\,]$, $A[\,a\,]$ , $P[\,n\,]$ )
1: $V[\,32\,] \leftarrow (\,0, 0, \cdots, 0\,)$              initialize the state with zeros
2: put( $K[\,16\,]$, BLNK_KEY )                  secret key, two $\pi$ ops
3: put( $N[\,8\,]$, BLNK_NPUB )                  public nonce, single $\pi$ op
4: put( $A[\,a\,]$, BLNK_AAD )                   associated authenticated data
5: $C[\,0\cdots n-1\,] \leftarrow$ enc( $P[\,n\,]$, BLNK_MSG )   encryption
6: $C[\,n\cdots n+7\,] \leftarrow$ get( 8 , BLNK_MAC )       message authentication code
7: return $C[\,n+8\,]$                          authenticated ciphertext
```

Inverse operation by the recipient:

```
{ $P[\,n\,]$ or FAIL } $\leftarrow$ decrypt( $K[\,16\,]$, $N[\,8\,]$, $A[\,a\,]$ , $C[\,n+8\,]$ )
1: $V[\,32\,] \leftarrow (\,0, 0, \cdots, 0\,)$              initialize the state with zeros
2: put( $K[\,16\,]$, BLNK_KEY )                  secret key, two $\pi$ ops
3: put( $N[\,8\,]$, BLNK_NPUB )                  public nonce, single $\pi$ op
4: put( $A[\,a\,]$, BLNK_AAD )                   associated authenticated data
5: $P[\,n\,] \leftarrow$ dec( $P[\,0\cdots n-1\,]$, BLNK_MSG )   decryption
6: if $C[\,n\cdots n+7\,] =$ get( 8 , BLNK_MAC ) then
7:     return $P[\,N\,]$                        auth match: $C[\,n\cdots n+7\,] = V[\,0\cdots 7\,]$
8: else
9:     return FAIL                          plaintext should be ignored (and cleared)
10: end if
```

The encryption function always returns the protected ciphertext message. Decryption either returns the plaintext or FAIL, indicating authentication failure. It is important that the decryption routine always performs full processing regardless of fail condition in order to minimize the risk of a timing attack. Also the confidential state can be cleared in order to minimize leakage.

1.4 Trace of `cbeam128r1` Computation

To illustrate the operation with CAESAR parameters, we use following plain ASCII values for input to encrypt() with $n = 18$:

$$\begin{array}{rcl} K[\,16\,] & = & \texttt{"128-Bit Test Key"} \\ N[\,8\,] & = & \texttt{"Nonce 64"} \\ A[\,3\,] & = & \texttt{"AAD"} \\ P[\,18\,] & = & \texttt{"cbeam128r1 payload"} \end{array}$$

Steps 1-2: Keying. After zeroing V, the first input to π is the first half of the secret key $K[\,0 \cdots 7\,]$:

31 32 38 2D 42 69 74 20 **10** 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

The state V after mixing the first half of the key is:

```
2F FF 3C 9F BD D1 74 14 D1 ED B1 A6 8D 38 03 FC D6 A5 35 09 AA 2A BB C9 53 56 49 F4 41 9A C9 22
```

We XOR in the padded second half of the secret key $K[\,8 \cdots 15\,]$:

54 65 73 74 20 4B 65 79 **13** 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

The state V after both keying mix π ops is:

```
31 D1 9A 01 BC 7A BF 32 7D 98 97 3D 96 0F 8F B2 96 9D 2D 16 BF 94 FE 84 DB AE E4 28 C2 D0 68 12
```

Step 3: Nonce mixing. The nonce $N[\,8\,]$ padded XOR value:

4E 6F 6E 63 65 20 36 34 **23** 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

The state V after nonce mixing π is:

```
C9 78 61 F1 99 FC CF 20 E9 C6 CD 3D 40 54 89 74 0D 59 92 A5 80 7C 49 9E B6 9F 90 AE 06 90 36 03
```

Step 4: Associated Authenticated Data. Padded AAD $A[\,3\,]$ XOR value.

41 41 44 **01** 00 00 00 00 **42** 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

The state V after AAD mixing π is:

```
78 DB C6 B1 91 05 35 B6 A9 E9 CD 90 0E C6 47 81 C3 B9 71 F1 26 03 A5 23 B8 8A DC 61 F6 49 BB 28
```

Step 5: Payload Encryption. First 8 bytes of plaintext $P[\,0 \cdots 7\,]$ with padding:

63 62 65 61 6D 31 32 38 **50** 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Resulting state V after first plaintext block:

```
02 49 10 6E 88 F4 49 89 90 09 3D 9E E4 DE 22 F5 AF 38 8B C0 78 BF 67 D6 BE 0A DD F3 7C 76 29 B7
```

Second block of plaintext $P[\,8 \cdots 15\,]$ with padding:

72 31 20 70 61 79 6C 6F **50** 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Resulting state V after second plaintext block:

```
00 4D 52 D1 2E 1F 61 A0 AF E5 B2 80 47 DE F1 C2 05 39 DB 11 71 A1 2E 1D 22 F7 3E 02 23 8E 8C EC
```

Final block of plaintext $P[\,16 \cdots 17\,]$ with padding:

61 64 **01** 00 00 00 00 00 00 **52** 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

State V after encryption process:

```
07 8D BB 1E 48 0A 20 0E E3 F5 C7 F4 8C E3 89 45 AB 33 DA B8 70 22 DC 68 23 03 C3 1C 36 6E A5 5A
```

Step 6: Authentication code. Here are the ciphertext bytes $C[\,26\,]$ from Step 5, together with the authentication code, underlined. These 8 bytes correspond to first 8 bytes of the final state above.

```
1B B9 A3 D0 FC 34 07 8E 70 78 30 1E E9 8D 25 E6 61 29 07 8D BB 1E 48 0A 20 0E
```

Chapter 2

Security Goals

2.1 Specific Goals

With the "cbeam128r1" set of parameters (as specified in Sections 1.1 and 1.3) we have the following security claims and goals:

Category	Effort	Attack Goal
Confidentiality for the plaintext.	2^{127}	To recover to the plaintext from ciphertext or vice versa.
Integrity for the plaintext.	2^{63}	To forge plaintext payload.
Integrity for the associated data.	2^{63}	To forge Associated Data .
Integrity for the public message number.	2^{63}	To forge public message number.

Here we assume that the secret key is entirely unknown to the attacker. The complexities are given for $P = 0.5$ success probability. Furthermore we assume that no more than 2^{64} bits of data is processed under any specific key, sequence number pair. The "unit" for the effort is equivalent to the effort required to compute the π permutation.

2.2 Nonce Re-Use

CBEAM does **not** allow re-use of public message numbers under the same key. In other words, users are required to use the public message number as a **nonce**. CBEAM may lose all of its security if a legitimate key holder uses the same sequence number and key to encrypt (and authenticate) two different messages.

2.3 General Goals

Our main security goals are largely compatible with those laid out for Authenticated Encryption [26] and Duplex Sponges in particular -- proofs in [7, 9] are applicable. For the primitives of Section 1.3.2:

priv The expected effort to distinguish ciphertext $C = \mathsf{encrypt}(K,\ N,\ A,\ P)$ from random is 2^{k-1} for random unknown key K and nonrepeating nonce N. Multiple $(N,\ A,\ P)$ may be chosen by the attacker, up to the data limit.

auth The expected effort to forge a message $(N,\ A,\ C)$ that does not result in $\mathsf{decrypt}(K,\ N,\ A,\ C) = \mathsf{FAIL}$ authentication failure is 2^{t-1} for random unknown key K and nonrepeating nonce N. Multiple $(N,\ A,\ C)$ may be chosen by the attacker, up to the data limit.

In general, confidentiality of plaintext will be consistent with key size k and the integrity (authentication) will be consistent with authentication tag size t if conditions for data limits and nonce re-use are held. Secret message numbers will have the same confidentiality as other payload, if used. There should not be any easily exploitable related-key properties.

Chapter 3

Security Analysis

We show that efficient and secure cryptographic mixing functions can be constructed from low-degree rotation-invariant ϕ functions rather than conventional S-Boxes. These novel functions have surprising properties; many exhibit inherent feeble (Boolean circuit) one-wayness and offer speed/area trade-offs unobtainable with traditional constructs. Recent theoretical results indicate that even if the inverse is not explicitly computed in an implementation, its degree plays a fundamental role to the security of the iterated composition [16].

3.1 Rotation-Invariant ϕ Functions

Introduced in Daemen's 1995 PhD Thesis [19], ϕ functions are rotation-invariant n-bit invertible (bijective) functions. We use a slightly different notation from Daemen who used ϕ to denote non-invertible as well as invertible rotation-invariant functions.

Definition 1. Let $f : \{0,1\}^n \mapsto \{0,1\}^n$ be a function from n-bit vectors to n-bit vectors. f is a ϕ function if it is bijective (uniquely invertible) and rotation-invariant: $f(x) = y \Rightarrow f(x \lll r) = y \lll r$ for all r.

Lemma 1. Any $n \times n$-bit ϕ function f is unambiguously characterized by an $n \times 1$ - bit function $f_{(1)}$ that satisfies $f_{(1)}(x) = f(x) \wedge 1$.

Proof. Directly from rotation invariance. □

Each output bit of the function may be dependent only on some subset of n input bits. This subset is not arbitrary; we found that neighboring input bits are more likely to yield invertible functions. In the present work ϕ_5 is a specific 5×1 - bit function and ϕ_{16} is a 16×16 - bit function defined by it as per Lemma 1. We note that each output bit of the inverse function f^-1 may be dependent on all input bits even though this is not the case for f (See Figure 3.1.)

3.1.1 Invertibility

It is easy to see that there are invertible $n \times n$ - bit ϕ functions for any $n > 1$ by considering $f(x) = cx \pmod{2^n - 1}$, where $\gcd(c, 2^n - 1) = 1$. Rotation invariance: $2^n \equiv 1 \pmod{2^n - 1}$ and $f(2^k x) = 2^k cx \pmod{2^n - 1}$. For invertibility $f^{-1}(x) = c^{-1}x \pmod{2^n - 1}$.

The inverse function f^{-1} can also be characterized by an $n \times 1$ - bit function $f^{-1}_{(1)}$ (Lemma 1) since the inverse of any ϕ function is clearly also a ϕ function. It may also be the case that $f = f^{-1}$. Hummingbird-2ν is an example of a cipher that utilizes two 16-bit ϕ functions which are in fact involutions [28]. The SIMON family of block ciphers from NSA is an example of a cipher that utilizes a *non-surjective* rotation-invariant function f as part of a Feistel construction [2].

It is nontrivial to characterize which one-bit $f_{(1)}$ functions generate invertible f functions apart from simple properties such as bit balance: $\sum_{x=0}^{2^n-1} f_{(1)}(x) = 2^{n-1}$. Good ϕ functions appear to be rather hard to find -- we resorted to optimized exhaustive tabulation methods to find our implementation-friendly and ``feebly asymmetric'' ϕ_5.

3.1.2 On Cryptanalysis of ϕ Functions

Algorithms for finding differential [12] and linear [25, 24] cryptanalytic properties of a ϕ function are relatively fast and straightforward to implement. Thanks to Lemma 1, when determining linear bounds we may assume that the input mask is a subset of the input bits to its $f_{(1)}$.

For differential cryptanalysis we must consider the convolution of the input differential w.r.t. a single output bit. Due to rotation we may always by convention set the bit at index 0 in the input differential.

Countermeasures must be taken against rotational cryptanalysis [23] due to inherent rotational invariance of ϕ functions. Algebraically these functions have surprising properties. See Section 3.2.4 and Section 3.3 for tables and conjectures related to ϕ_5.

3.2 CBEAM Analysis

Ignoring the round constant, the mx transform may be viewed as a transpose of a matrix followed by 16 parallel, independent invocations of a 16 - bit permutation, $(\phi \circ \lambda)_{16}$. We start with the most fundamental observation:

Theorem 1. *The mx transform is bijective (reversible).*

Proof. The mx transform is bijective as all of its component functions are individually reversible. It is trivial to see that the linear transform λ is bijective. Since convolution by a nonlinear Boolean function is generally not reversible, one may compute the 2^{16} - entry table of ϕ_{16} to verify that it is surjective. $\square$

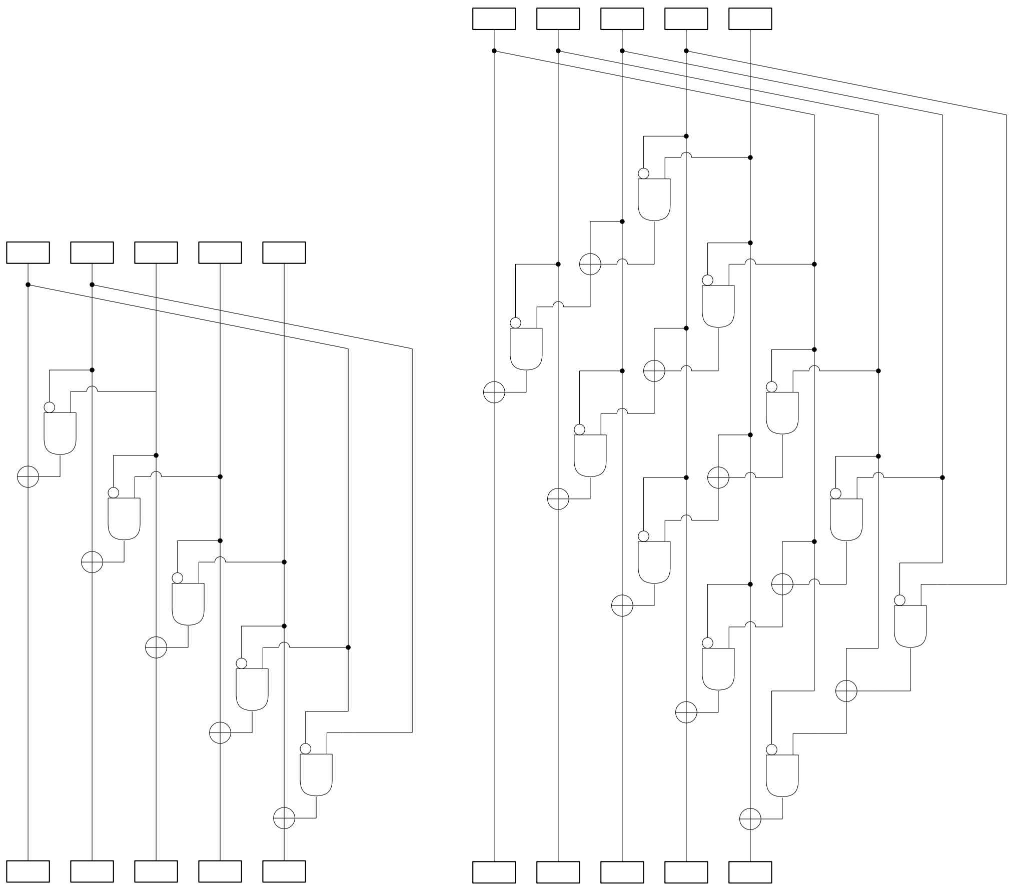

Figure 3.1: On left, a circuit implementing KECCAK's 5×5 - bit χ component, which happens to be a rotation-invariant ϕ function of degree 2. On right, a circuit implementing its inverse permutation, χ^{-1}, which has Degree 3 with each output bit dependent on all input bits. Such asymmetric Boolean and circuit complexity is characteristic of ϕ functions.

Table 3.1: Probabilities (%) of best differentials for $(\lambda \circ \phi)_{16}$ with specific input weight (rows) and output weight (columns). The best overall differential and the best differential with output weight 1 are emphasized in bold.

Wt	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	0	0	1.07	.513	.635	.562	.385	.330	.140	.137	.064	.021	.003	0	0	0
2	0	4.30	2.15	2.54	2.15	1.37	.592	.443	.284	.256	.116	.098	.037	.027	.006	.006
3	**17.2**	5.47	5.47	3.91	1.78	.922	.787	.476	.330	.195	.177	.119	.079	.052	.027	.003
4	.009	1.46	3.37	5.15	1.95	1.32	.903	.439	.305	.375	.232	.159	.101	.064	.049	.021
5	.684	2.93	6.74	2.49	2.20	1.76	.885	.635	.446	.363	.266	.192	.140	.085	.064	.021
6	7.03	**18.4**	5.47	3.91	2.34	1.37	.894	.702	.412	.354	.214	.168	.131	.128	.052	.018
7	.928	2.00	4.17	2.12	3.09	1.64	1.14	.671	.470	.299	.247	.223	.165	.101	.040	.024
8	2.93	3.22	3.22	4.15	3.12	1.95	1.20	.732	.522	.360	.220	.256	.140	.070	.049	.034
9	8.20	4.00	11.1	4.59	3.52	1.95	1.28	.885	.525	.366	.253	.171	.134	.101	.067	.024
10	.598	1.39	1.66	2.73	1.46	2.27	1.44	.781	.586	.323	.220	.208	.131	.153	.043	.021
11	.964	2.44	3.27	2.05	3.96	2.22	1.27	.879	.403	.232	.266	.192	.165	.092	.037	.027
12	.781	5.57	2.83	6.74	2.34	1.86	.696	.439	.290	.296	.198	.171	.128	.058	.040	.031
13	0	.122	.159	.323	.247	.269	.327	.317	.272	.214	.223	.119	.082	.058	.031	.009
14	0	.018	.073	.150	.177	.250	.269	.424	.235	.275	.140	.104	.061	.052	.024	.015
15	0	0	.003	.006	.079	.064	.122	.058	.076	.064	.055	.037	.043	.034	.021	0
16	0	0	0	0	0	.006	.006	.049	.024	.055	.031	.058	.015	.021	0	.012

3.2.1 Rotational and Slide Attacks

The choice of round constants was specially crafted to deter rotational [23] and slide [13, 14] attacks, while having minimal impact on implementation footprint.

Theorem 2. *Without the round constants the mx transform is shift-invariant both horizontally and vertically. Let* $\mathbf{s}' = \mathsf{mx}(\mathbf{s})$ *and* $\mathbf{t}' = \mathsf{mx}(\mathbf{t})$. *If each element* $s[\,i\,][\,j\,] = t[\,(i+4\Delta_i) \bmod 16\,][\,(j+\Delta_j) \bmod 16\,]$ *for some offsets* Δ_i *and* Δ_j, *then* $s'[\,i\,][\,j\,] = t'[\,(i+\Delta_j) \bmod 16\,][\,(j+4\Delta_i) \bmod 16\,]$.

Proof. The theorem follows form shift-invariant properties of all component functions. Note the exchange of indices $4\Delta_i$ and Δ_j due to transpose. $\square$

3.2.2 Selection of ϕ_5

We analyzed all $2^{2^5} = 2^{32}$ five-input Boolean functions, searching for ones that result in invertible 16-bit ϕ functions with particularly good properties. Five neighboring bits are used since rotation amounts that would yield better branching (such as the set $\{0, 1, 2, 4, 8\}$) didn't result in any appropriate functions. Single left rotations are used as it is universally available (16-bit left rotation can be implemented by adding a number to itself and then adding the carry bit back to the least significant bit of the sum).

There were 260 invertible functions, of which 56 were dependent on all five input bits in nonlinear fashion. Eight of these exhibited optimal differential and linear properties. However there are three independent mirror symmetries (inversion of all input and output bits and the order of input bits) and therefore $2^3 = 8$ equivalent functions. Discounting these symmetries, there is only one optimal function, ϕ_5 (Equation 1.5).

Invertibility ϕ_5 of for other word sizes besides $n = 16$ and the surprising properties of these inverse functions are analyzed in Section 3.3.

3.2.3 Differential and Linear Cryptanalysis

Sponge functions can be attacked with DC [12] and LC [25, 24] even though reasonable attack models are radically different from block ciphers.

Because of λ, changing one bit of the input will spread the difference to at least three bit positions outside the first quadword which can be modified by the attacker. After four of six mx iterations, there is no easily detectable bias regardless of input difference, which we feel is an appropriate security margin. See Table 3.3 for an illustration of progress of differentials in the state during forward and reverse iterations.

Table 3.2: Absolute biases (%) of best linear approximations for $(\lambda \circ \phi)_{16}$ with specific input mask weight (rows) and output weight (columns). The best approximation is emphasized in bold.

Wt	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	0	0	6.25	3.71	4.69	3.12	2.73	1.56	1.17	.586	.439	0	0	0	0	0
2	0	12.5	7.03	9.38	6.25	4.88	3.71	2.93	3.32	1.86	1.27	1.27	.684	.391	0	0
3	**25.0**	12.5	9.38	6.25	5.08	4.49	4.59	3.71	2.73	1.95	1.66	1.42	1.37	.830	.684	0
4	0	7.03	9.38	7.81	7.81	7.03	3.91	3.81	3.03	2.88	4.39	6.15	3.37	2.73	1.27	2.15
5	6.25	12.5	9.38	8.59	9.38	5.86	5.08	3.91	5.27	6.05	6.84	3.47	2.64	2.49	1.95	1.07
6	18.8	18.8	15.6	10.9	7.03	5.47	5.27	6.45	6.25	8.01	4.83	3.32	2.15	1.95	1.46	1.17
7	0	7.81	10.9	12.5	7.81	6.64	7.42	8.40	8.40	4.59	3.91	4.15	4.74	2.78	3.42	1.27
8	6.25	15.6	14.1	9.38	10.2	8.59	8.59	9.77	6.45	4.79	4.83	3.96	3.76	3.76	2.25	.977
9	18.8	18.8	15.6	10.9	8.59	10.5	10.9	6.25	4.69	3.96	4.39	3.56	2.88	2.15	1.95	.879
10	0	7.03	7.81	9.38	9.38	14.1	8.59	5.08	4.49	4.54	3.61	3.96	3.76	4.98	2.83	2.25
11	6.25	7.81	7.81	10.9	15.6	7.81	6.25	4.59	3.61	3.32	4.20	4.88	3.08	2.98	1.86	1.17
12	6.25	9.38	14.1	17.2	9.38	6.25	3.91	3.32	3.37	3.32	3.66	4.20	2.98	2.59	1.46	1.56
13	0	0	1.95	2.93	2.93	2.93	3.71	2.88	3.27	4.20	3.52	2.78	3.27	3.52	5.08	.586
14	0	0	.391	.684	1.90	2.05	2.44	2.29	2.93	2.98	2.59	2.98	2.69	2.78	1.46	.977
15	0	0	0	.684	.635	1.22	1.66	1.76	1.81	1.76	2.05	1.86	2.29	1.17	.684	0
16	0	0	0	0	0	0	0	.488	.537	.684	.977	1.37	1.32	2.78	1.17	9.38

For this analysis we view $(\phi \circ \lambda)_{16}$ row operation as a 16×16 - bit ``S-Box''. The highest-probability differential is `0CCC` $\rightarrow$ `8001` and its rotational equivalents. The probability of this differential is $\frac{12032}{2^{16}} \approx 0.1836$. From Table 3.1 we observe that a 1-bit input difference never yields a 1-bit output difference (branch number is greater than 2).

The best linear approximation for $(\phi \circ \lambda)_{16}$ is `0888` $\rightarrow$ `0001` and its rotational equivalents, which have a bias of $\frac{16384}{2^{16}} = \frac{1}{4}$. The other best approximations are given in Table 3.2. Significantly, all single bit approximations have 0 linear bias, as do 2-to-1 and 1-to-2 - bit approximations.

3.2.4 Algebraic Properties and One-Wayness

From its definition in Equation 1.5 one easily see that the degree of ϕ_5 is 4 (and ANF weight 13), and that is also the algebraic degree of ϕ state transform `mx` (see Equation 1.4).

The `mx` function has been designed to have a significant amount of algebraic ``one-wayness'' in the sense discussed by Hiltgen [21]. The following somewhat surprising observation can be verified by examining the inverse of ϕ_{16}:

Observation 1. *The algebraic degree of the ϕ_{16}^{-1} inverse function is 11. The weight (number of nonzero terms) of the ANF polynomial for each output bit of ϕ_{16}^{-1} is 13465.*

For a characterization of the Algebraic properties of the inverse of ϕ_n^{-1} for n other than 16, we refer to Section 3.3, where more general tables and conjectures are presented.

The algebraic degree of mx^n is bound by 4^n. We have verified that the output after six invocations actually has a degree up to 256. If state bits are observed as a function of $s_q[\ 0\ ]$, the number of terms of each degree are distributed in a way that indicates that CBEAM is not vulnerable to d-monomial distinguishers [27] or other traditional algebraic attacks.

Research by Boura and Canteaut on the algebraic degree of iterated permutations seen as multivariate polynomials shows that the degree depends on the algebraic degree of the *inverse* of the permutation which is iterated [16]. This indicates exceptional algebraic security for our proposal.

Table 3.3: Progression of differentials in consecutive invocations of mx. Here the zeroth bit has been flipped; $\Delta = 0^{255} \mid\mid 1$. Particular bit selection does not significantly alter outcome (as these are rotation-invariant functions). We observe that the full state is affected and there is no detectable bias after mx^4. The π transform has six rounds by default.

$\mathsf{mx}(x) \oplus \mathsf{mx}(x \oplus \Delta)$

#	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
00	38	50	63	50	37	50	62	50	37	50	63	38	00	00	00	**25**
01	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
02	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
03	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
04	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
05	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
06	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
07	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
08	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
09	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
11	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
12	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
13	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
14	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
15	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

$\mathsf{mx}^2(x) \oplus \mathsf{mx}^2(x \oplus \Delta)$

#	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
00	09	12	16	12	09	12	16	12	09	12	16	09	00	00	00	06
01	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
02	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
03	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
04	14	19	23	19	14	19	23	19	14	19	23	14	00	00	00	09
05	23	31	39	31	23	31	39	31	23	31	39	23	00	00	00	16
06	19	25	31	25	19	25	31	25	19	25	31	19	00	00	00	12
07	14	19	23	19	14	19	23	19	14	19	23	14	00	00	00	09
08	19	25	31	25	19	25	31	25	19	25	31	19	00	00	00	13
09	23	31	39	31	23	31	39	31	23	31	39	23	00	00	00	16
10	19	25	31	25	19	25	31	25	19	25	31	19	00	00	00	13
11	14	19	23	19	14	19	23	19	14	19	23	14	00	00	00	09
12	19	25	31	25	19	25	31	25	19	25	31	19	00	00	00	12
13	23	31	39	31	23	31	39	31	23	31	39	23	00	00	00	16
14	19	25	31	25	19	25	31	25	19	25	31	19	00	00	00	12
15	14	19	24	19	14	19	23	19	14	19	23	14	00	00	00	09

$\mathsf{mx}^3(x) \oplus \mathsf{mx}^3(x \oplus \Delta)$

#	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
00	33	34	33	32	33	34	33	32	33	36	36	37	38	39	36	32
01	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
02	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
03	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
04	40	42	41	39	40	42	41	39	40	43	43	42	44	45	43	39
05	47	48	47	45	47	49	47	45	46	48	48	47	49	49	48	45
06	44	46	45	43	44	46	45	43	44	46	46	46	47	48	46	43
07	40	42	41	39	40	42	41	39	40	43	43	42	44	45	43	39
08	44	46	45	43	44	46	45	43	44	46	46	46	47	48	46	43
09	46	48	47	45	47	49	47	45	46	48	48	47	48	49	48	45
10	44	46	45	43	44	46	45	42	44	46	46	46	47	48	46	43
11	40	42	41	39	40	42	41	39	40	42	43	42	44	45	43	39
12	44	46	45	43	44	46	45	43	44	46	46	46	47	48	46	43
13	46	48	47	45	47	49	47	45	46	48	48	47	48	49	48	45
14	44	46	45	43	44	46	45	43	44	46	46	46	47	48	46	43
15	40	42	41	39	40	42	41	39	40	43	43	42	44	45	43	39

$\mathsf{mx}^4(x) \oplus \mathsf{mx}^4(x \oplus \Delta)$

#	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
00	49	49	49	49	49	49	49	49	49	49	49	50	50	50	49	49
01	49	49	50	49	49	50	50	49	49	50	50	50	50	50	50	49
02	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
03	50	50	50	50	50	50	50	49	50	50	50	50	50	50	50	50
04	49	50	50	49	49	50	50	49	49	50	50	50	50	50	50	49
05	49	49	50	49	49	49	50	49	49	49	50	50	50	50	50	49
06	49	50	50	49	50	50	50	49	49	50	50	50	50	50	50	49
07	49	49	49	49	49	49	49	49	49	49	50	50	50	50	50	49
08	49	49	49	49	49	49	49	49	49	49	49	49	50	50	50	49
09	49	49	49	49	49	49	49	49	49	49	49	50	50	50	50	49
10	50	50	50	49	49	50	50	49	50	50	50	50	50	50	50	49
11	49	49	49	49	49	49	50	49	49	49	50	50	50	50	50	49
12	49	49	49	49	49	49	49	49	49	49	49	49	50	50	50	49
13	49	49	49	49	49	49	49	49	49	49	49	50	50	50	50	49
14	50	50	50	49	50	50	50	49	49	50	50	50	50	50	50	49
15	49	49	49	49	49	49	49	49	49	49	50	50	50	50	50	49

Table 3.4: Progression of differentials in consecutive invocations of inverse function mx^{-1}. Here again the zeroth bit is flipped. There is no detectable bias after third round.

$\mathsf{mx}^{-1}(x) \oplus \mathsf{mx}^{-1}(x \oplus \Delta)$

#	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
00	00	00	00	00	00	00	00	00	00	00	00	00	50	50	50	**50**
01	00	00	00	00	00	00	00	00	00	00	00	00	50	50	50	50
02	00	00	00	00	00	00	00	00	00	00	00	00	50	50	50	50
03	00	00	00	00	00	00	00	00	00	00	00	00	50	50	50	50
04	00	00	00	00	00	00	00	00	00	00	00	00	50	50	50	50
05	00	00	00	00	00	00	00	00	00	00	00	00	50	50	50	50
06	00	00	00	00	00	00	00	00	00	00	00	00	50	50	50	50
07	00	00	00	00	00	00	00	00	00	00	00	00	50	50	50	50
08	00	00	00	00	00	00	00	00	00	00	00	00	50	50	50	50
09	00	00	00	00	00	00	00	00	00	00	00	00	50	50	50	50
10	00	00	00	00	00	00	00	00	00	00	00	00	50	50	50	50
11	00	00	00	00	00	00	00	00	00	00	00	00	50	50	50	50
12	00	00	00	00	00	00	00	00	00	00	00	00	50	50	50	50
13	00	00	00	00	00	00	00	00	00	00	00	00	50	50	50	50
14	00	00	00	00	00	00	00	00	00	00	00	00	50	50	50	50
15	00	00	00	00	00	00	00	00	00	00	00	00	50	50	50	50

$\mathsf{mx}^{-2}(x) \oplus \mathsf{mx}^{-2}(x \oplus \Delta)$

#	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
00	20	25	19	25	34	34	24	25	24	25	25	25	25	25	25	25
01	24	21	22	20	20	20	20	21	20	21	21	20	21	21	21	21
02	23	21	21	21	21	21	20	21	20	21	21	21	21	21	21	22
03	39	39	39	39	40	40	39	39	39	39	39	39	39	39	39	39
04	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
05	49	50	50	50	49	49	50	50	50	50	50	50	50	50	50	50
06	46	47	47	48	49	49	48	47	48	47	47	48	47	47	47	47
07	49	48	47	47	47	47	46	47	46	48	48	47	48	48	48	49
08	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
09	49	50	50	50	47	47	49	50	49	50	50	50	50	50	50	50
10	44	46	44	47	51	51	47	46	47	46	46	47	46	46	46	46
11	50	47	48	47	47	47	46	47	46	47	47	47	47	47	48	48
12	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
13	49	49	49	49	47	47	49	49	49	49	49	49	49	49	49	50
14	41	45	42	45	50	50	46	45	46	45	45	45	45	45	45	45
15	47	38	40	36	38	38	35	37	34	38	37	36	39	38	39	40

3.3 Tables and Conjectures on Algebraic Properties of ϕ_n^{-1}

The ϕ_5 (Equation 1.5) Boolean mapping also defines reversible $n \times n$ - bit shift-invariant functions for other n apart from $n = 16$ via Definition 1. Each forward function has degree 4 as is evident from the function itself.

The characteristics of the Algebraic Normal Form of inverse functions up to $n = 29$ are given below (we have computed them up to $n = 32$). Each column contains the number of monomials of given degree in each output bit.

n	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
6	*Nonsurjective.*																			
7	4	11	17	15	6															
8	3	9	13	13	9	2														
9	*Nonsurjective.*																			
10	5	21	55	91	95	56	14													
11	4	18	45	75	88	69	28	4												
12	*Nonsurjective.*																			
13	6	34	125	303	502	565	408	168	30											
14	5	30	106	253	433	543	471	252	72	8										
15	*Nonsurjective.*																			
16	7	50	236	753	1705	2797	3293	2686	1430	446	62									
17	6	45	205	640	1456	2504	3236	3017	1912	766	172	16								
18	*Nonsurjective.*																			
19	8	69	397	1570	4506	9678	15684	19001	16832	10532	4402	1104	126							
20	7	63	351	1356	3866	8472	14450	18965	18645	13266	6554	2114	396	32						
21	*Nonsurjective.*																			
22	9	91	617	2910	10112	26816	55170	88281	109077	102570	71834	36250	12464	2618	254					
23	8	84	553	2548	8750	23352	49428	83181	110136	112723	87302	49868	20260	5510	892	64				
24	*Nonsurjective.*																			
25	10	116	905	4956	20216	63770	158824	315095	498190	624397	614364	467824	269904	114084	33356	6036	510			
26	9	108	820	4390	17654	55622	140638	288151	477827	636095	671875	555352	353222	168890	58546	13834	1980	128		
27	*Nonsurjective.*																			
28	11	144	1270	7918	37078	135562	396082	936523	1801051	2816653	3568633	3638674	2956588	1887016	925480	336844	85766	13646	1022	
29	10	135	1161	7083	32664	118764	349392	843177	1676448	2740338	3661044	3966297	3452310	2386518	1289610	532002	161404	33822	4348	256

We offer the following two conjectures:

Conjecture 1. *The inverse of ϕ_n is defined for each $n \geq 5$ with $n \neq 0 \pmod 3$ and* $\deg \phi_n^{-1} = \lceil \frac{2}{3} n \rceil$.

Conjecture 2. *Computation of ϕ_n^{-1} has at least polynomial complexity (with degree ≥ 2).*

The computation of ϕ_n has linear complexity $O(n)$ but the complexity of ϕ_n^{-1} is at least $O(n^2)$ since the number of input bits grows with n as per observation in Conjecture 1. Therefore these functions really appear to be "one-way".

Even super-polynomial complexity has not been ruled out as we do not know a polynomial time algorithm for ϕ_n^{-1}. The progression of the total number of nonzero monomials in the polynomials is captured in the two vectors

$$v = (53, 337, 2141, 13465, 83909, 519073, 3192557, 19545961, 119228885) \tag{3.1}$$

$$w = (49, 331, 2173, 13975, 88537, 554659, 3445141, 21256783, 130470385) \tag{3.2}$$

Where v_i corresponds to $n = 3i + 4$ and w_i to $n = 3i + 5$. These sequences are clearly exponential and correspond to $\approx 1.8^n$. Based on current evidence we are reluctant to believe in exponential evaluation complexity, however.

3.4 Sponge Functions

Sponge constructions generally consist of a state $S = (S^r \mid\mid S^c)$ which has $b = r + c$ bits and a b-bit keyless cryptographic permutation π. The S^r component of the state has r "rate" bits which interact with the input and the internal S^c component has c private "capacity" bits. Our selection for these parameters is given in Section 1.3.

These components, together with suitable padding and operating rules can be used to build provable Sponge-based hashes [4], Tree Hashes [10], Message Authentication Codes (MACs) [8], Authenticated Encryption (AE) algorithms [7], and pseudorandom extractors (PRFs and PRNGs) [5].

3.4.1 Absorbing and Squeezing

We recall the basic Sponge hash [4] concepts of "absorbing" and "squeezing" which intuitively correspond to insertion and extraction of data to or from the sponge. Let S_i and S_{i+1} be b-bit input and output states. For absorption of padded data blocks M_i (of r bits each) we iterate:

$$S_{i+1} = \pi(\ S_i^r \oplus M_i \mid\mid S_i^c\ ). \tag{3.3}$$

This stage is followed by squeezing out the hash $H = H(M)$ by consecutive iterations of:

$$\begin{aligned} H &= H \mid\mid S_i^r \\ S_{i+1} &= \pi(S_i). \end{aligned} \tag{3.4}$$

These constructions may be transformed into a keyed MAC by considering the state S_i as secret (keyed) [8]. Keying is then equivalent to initial absorption of keying material before the payload data. MAC is squeezed out exactly like a hash.

3.4.2 Duplexing

A further development was the Duplex construction [7] which allows us to encrypt and decrypt data while also producing a MAC in the end with a single pass.

The state is first initialized by inserting secret keying material and non-secret randomization data to the state via the absorption mechanism of Equation 3.3. To encrypt plaintext blocks P_i to ciphertext blocks C_i we iterate:

$$\begin{aligned} C_i &= S_i^r \oplus P_i \\ S_{i+1} &= \pi(C_i \mid\mid S_i^c). \end{aligned} \tag{3.5}$$

The effect on the state is the same as that of Equation 3.3. The inverse -- decryption operation -- is almost equivalent to encryption, which in itself has significant implementation advantages:

$$\begin{aligned} P_i &= S_i^r \oplus C_i \\ S_{i+1} &= \pi(C_i \mid\mid S_i^c). \end{aligned} \tag{3.6}$$

After encryption or decryption, a message authentication code for the message may be squeezed out as in Equation 3.4 and verified. To simplify exposition, we have left some key details regarding padding. We will come back to these in Section 3.4.4. Figure 3.2 shows operation of a generic Sponge-based AEAD.

3.4.3 MAC-and-Continue

There is really no need to constrain the iteration to a single message. With appropriate domain-separating padding the security proofs allow the sponge states to be used for any number of consecutive authenticated messages ("MAC-and-Continue") without the need for sequence numbers, and re-keying. This is one of the main observations which led to the present work and greatly reduces the latency of implementation as "initialization rounds" are not required for each message. This was also proposed as part of the original SpongeWrap construction. However, we are not using this capability as a part of the CAESAR proposal.

3.4.4 Duplex, Triplex, Multiplex

The SpongeWrap [7] and monkeyDuplex [9] padding rules offer concrete Sponge-based methods for performing authenticated encryption. Recent work on implementation of SpongeWrap and its variants on low-resource platforms is reported in [36].

The requirements laid out in [7] for the padding rule are that they are reversible, non-empty and that the last block is non-zero. The padding rule in Keccak is that a single 1 bit is added after the last bit of the message and also at the end of the input block.

In the Duplex construction of SpongeWrap additional padding is included for each input block; a secondary information bit called *frame bit* is used for domain separation. Sakura [10] uses additional frame bits to facilitate tree hashing. It is essential that the various bits of information such as the key, authenticated data, and authenticated ciphertext can be exactly "decoded" from the Sponge input to avoid trivial padding collisions. We use a more explicit padding mechanism but the following priv and auth bounds proven in [7] (Section 5.2 on Page 332) and [8] also hold for enc():

Theorem 3 (Theorem 1 from [7]). *The* SpongeWrap *and BLINKER authenticated encryption modes satisfy the following privacy and authentication security bounds:*

$$\mathrm{Adv}^{\mathsf{priv}}_{\mathsf{enc}}(\mathcal{A}) < q2^{-k} + \frac{N(N+1)}{2^{c+1}} \tag{3.7}$$

$$\mathrm{Adv}^{\mathsf{auth}}_{\mathsf{enc}}(\mathcal{A}) < q2^{-k} + 2^{-t} + \frac{N(N+1)}{2^{c+1}} \tag{3.8}$$

against any single adversary $\mathcal{A}$ if $K \xleftarrow{\$} \{0,1\}^k$, tags of $l \geq t$ bits are used, π is a randomly chosen permutation, q is the number of queries and N is the number of times π is called.

Note that even the Squeezing phase can utilize padding to mark the size of desired output (as we do in Section 1.3). In Keccak and SpongeWrap a convention has been adopted to have a null S_r input to π during squeezing in order to separate it from other phases (hence the requirement that padding rule does not produce null blocks). However this may lead to problems in some applications where the MAC length is not clear.

Current variants of Blinker utilize padding on MAC output, but this is not detectable on output unless MAC-and-Continue is used 3.4.3.

3.4.5 Multiplexing the Sponge

We term our multi-purpose padding as "Multiplex padding". There are more than two different data domains (as in Duplex padding). Input and output blocks, encrypted and authenticated data, keys, and nonces are all different data domains and are encoded unambiguously as Sponge inputs.

Rather than using frame bits per block for domain separation as in SpongeWrap, the data domains are explicitly encoded. This allows many more data types to be entered into the sponge as well and clearer domain separation between them. In a shared-state two-party half-duplex protocol that the originating

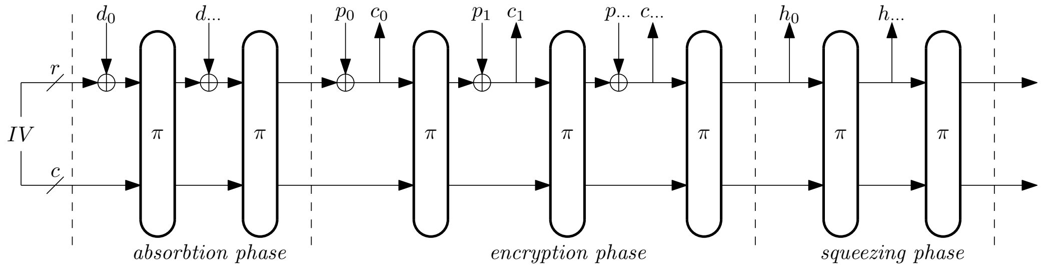

Figure 3.2: A simplified view of a Sponge-based AEAD. First the padded Secret Key, Nonce, and Associated Authenticated Data - all represented by d_u words - are "absorbed" or mixed into the Sponge state. The π permutation is then used to also encrypt data p_i into ciphertext c_i (or vice versa) and finally to "squeeze" out a Message Authentication Code h_i.

party of the block (Alice or Bob) is also used to mark domain separation between the two [29], but this feature is not used in this proposal.

We retain one d-bit word D in S^c for domain separation; $S^c = (S^d \mid\mid S^{c'})$ with $c' = c - d$. The iteration for arbitrary absorption, squeezing, and encryption is now:

$$S_{i+1} = \pi(\ S_i^r \oplus M_i \mid\mid S_i^d \oplus D_i \mid\mid S_i^{c'}\ ). \tag{3.9}$$

For decryption we have the following update function:

$$S_{i+1} = \pi(\ C_i \mid\mid S_i^d \oplus D_i \mid\mid S_i^{c'}\ ). \tag{3.10}$$

In our implementation $d = 8$ bits. Section 1.3.1 gives a description of padding mask byte bits (which may be OR'ed together). Message blocks are always padded with a single "1" bit and by zeros to fill r bits, followed by the multiplex padding byte. If full r bits are used in a block, the padding bit is the bit 0 in the multiplex word.

3.4.6 Domain Separation and Capacity Reduction

The domain indicator word is XORed with the capacity bits on all operations (Equations 3.9 and 3.10). We do this in order to remove the requirement for additional message padding buffers (caused by frame bits) and also to follow Horton's Principle [20, 34], *"Authenticate what is being meant, not what is being said."*

In CAESAR AEAD mode the different data domains follow each other in specific predetermined order (Section 1.3.2) and hence only two bits of entropy is sufficient to encode the final bit and separation between block and domain types. Therefore the effective c for values bounds of Theorem 3 need to be modified only by two bits when multiplex padding is used. We estimate the effective information theoretic capacity is reduced by the Multiplex construction to no less than $c - 2$ rather than $c' = c - d$.

The separation of the domain mask word from main "rate" input allows later expansions of functionality without breaking interface designs; for example we may adopt tree-based hashing - and by extension, tree MACs and encryption - by utilizing additional bits of D_i for this purpose rather than adding more frame bits as in Sakura [10]. If tree structure is used, the capacity should be reduced to $c - 3$ or $c - 4$ for security analysis. Adding further options or even increasing $d > 8$ for some applications will not break compatibility with existing implementations if these features are not used.

Since the protocol exchange can be unambiguously decoded from the sponge input and we do not reset the state between messages, the proofs of Theorem 3 [7, 8] apply to the protocol as a whole as well as individual messages. If one can forge an individual message authentication code or (by induction) a multi-message exchange, one can also break the Sponge in a SHA-3 - type hash construction.

Chapter 4

Features

4.1 Advantages over AES-GCM

CBEAM has been designed for resource-limited platforms. We expect serialized hardware implementations to have truly exceptionally small footprint in the range of only few hundred gate equivalents, excluding the state memory (which can be shared with other functionality such as protocol implementation). The reasons should be evident when examining the specification in Section 1.2.

We show in Section 4.5 that on a standard 16-bit MSP430 ultra-lightweight sensor platform CBEAM throughput is better to that of best hand-optimized AES-128 implementations, with implementation footprint of less than 10%. If we factor in the added complexity of running GCM on such a platform, CBEAM is signficantly faster.

However, unlike many other "lightweight" ciphers such as PRESENT [15] or KATAN [17], CBEAM scales very well with higher-end systems. We show in Section 4.6 how CBEAM reaches superior performance to AES on the Haswell architecture (when AES-specific assembler instructions are not used).

CBEAM also has a wider application base compared to many other lightweight cryptographic algorithms. Even full communications security suites can be implemented [29]. To summarize:

- **Originality.** CBEAM is a radically different type of novel design. The types of attacks that can be devised against AES-GCM are very unlikely to be applicable to CBEAM.
- **Lightweight applications.** CBEAM is very well suited for lightweight platforms, providing encryption, authentication, hashing, random number generation in only few hundred bytes or gates.
- **Scalability.** CBEAM scales very well with hardware, all the way to the latest 256-bit architectures.

Additionally we feel that CBEAM opens many interesting avenues for future exploration and theoretical research, especially considering the "one-wayness" conjectures given in Section 3.3.

4.2 Implementation Flexibility

One the most useful features of ϕ functions is its extreme flexibility with the amount of implementation trade-offs allowed. Computation of an $n \times n$ - bit ϕ function can take anywhere from 1 (fully unrolled) to $c \times n$ cycles (serial implementation -- here c is some constant), depending on target hardware platform. This is illustrated in Figure 4.1.

On software platform, ϕ functions allow efficient implementation of large "S-boxes" via a Boolean sequence programming technique resembling bit-slicing [11]. Finding a good bit-slicing Boolean description for an $n \times 1$ - bit function is much easier than for a generic $n \times n$ - bit S-Box. Such straight-line code is resistant to cache-based side-channel timing attacks such as those reported against AES [1, 3, 35].

CBEAM is highly flexible when it comes to implementation platforms. A standard C implementation may compute four rows in parallel using 64-bit data types whereas specific implementation strategies exist that fully utilize architectures from 16-bit to 256-bit word size. In hardware implementations, an invocation of the mx^n transform can take anywhere between 1 and several thousand clock cycles, depending on the number of gates, peak energy and amount of surface area available.

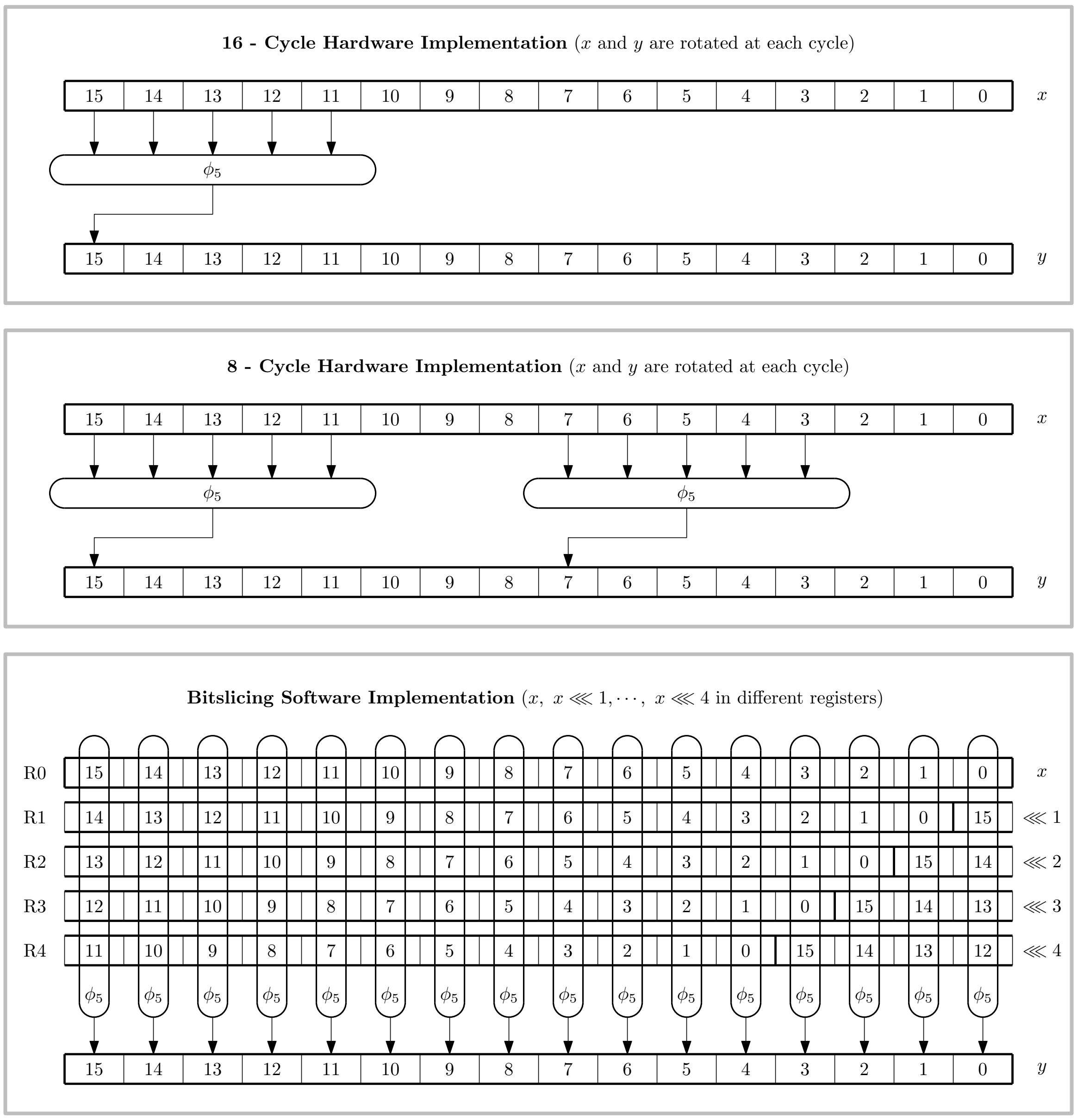
16 - Cycle Hardware Implementation (x and y are rotated at each cycle)
ϕ_5
x
y
8 - Cycle Hardware Implementation (x and y are rotated at each cycle)
ϕ_5
x
y
Bitslicing Software Implementation ($x,\ x \lll 1, \cdots,\ x \lll 4$ in different registers)
R0
R1
R2
R3
R4
x
$\lll 1$
$\lll 2$
$\lll 3$
$\lll 4$
ϕ_5
y

Figure 4.1: Example of a 16×16 - bit function ϕ_{16} based on a 5×1 - bit Boolean function ϕ_5. We observe 16, 8, and 1 cycle implementations of the same function. Note that the last example is similar to a 16-bit "bit slicing" software implementation using rotated words (of course ϕ_5 requires more than one cycle in software). With larger word sizes even higher levels of parallelism can be achieved.

4.3 Hardware Implementations

Figure 4.2 shows a simplified interface for a module that implements CBEAM in hardware. The mode of operation is determined by the domain separation padding word PADDING IN (as specified in Section 1.3.1) together with the SEND / RECEIVE signal that distinguishes between encryption and decryption, MAC generation and verification.

In a hardware implementation the secret "capacity" state bits never have to leave (or cannot leave) a specific hardware component, making the design attractive in HSM and smart card applications. Such separation is very difficult (and costly) to achieve with SSL and other legacy protocols which generally require CPU/MCU interaction to create encryption and authentication keys from session secrets. Note that this feature is not available in all sponge constructions.

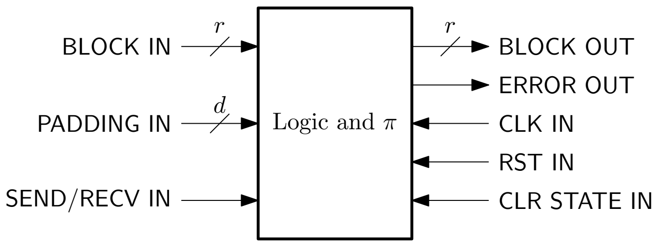

Figure 4.2: A simplified interface architecture for a semi-autonomous hardware component implementing CBEAM. The mode BLNK allows allows all c "capacity" bits to be protected inside HSM.

4.3.1 Block Implementation

This is a 1 - cycle implementation of the mx function (with e.g. 256 parallel ϕ_5 circuits). Depending on target platform and area, timing constraints, it is possible to implement more than one round of mx^2 in a single cycle. The interface is similar to that given in Figure 4.2. Pipelined operation using Sakura-like [10] hopping hash trees can also be considered with this mx core.

4.3.2 Serial Implementation

The Serial implementation assumes external 256-bit memory for the state and operates on that state one bit at a time. The implementation sacrifices a lot of clock cycles for reduction of gates and area. The implementation requires only 16 internal register bits in addition to address/clock counters. The implementation with a 1-bit data bus requires 256 read cycles and 256 write cycles for each MX iteration, 3072 clocks in total for full $\pi = \mathsf{mx}^6$. We estimate that the implementation footprint is only about 300 GE without the 256-bit external state memory. Note that this state memory is to be shared with other components such as IO hardware.

4.4 Implementing CBEAM in Software Without Matrix Transpose

In Software, an optimized bit-slicing method for ϕ_5 is used. The ANSI C reference implementation `ref/mx6-gcc.c` implements ϕ_5 as a macro as follows:

```
#define CBEAM_PHI5(x0, x1, x2, x3, x4)      \
    ((~(x0 & ((~x3 & x4) ^ (~x2 & x3))) &   \
    (x1 | (~x2 & x3))) ^ (~x2 & (~x3 & x4)))
```

Here we put our faith to the compiler optimizer for common subexpression elimination of `(~x2 & x3)` and `(~x3 & x4)`, which both appear twice. One can assign these to temporary variables if necessary. We have exhaustively verified that ϕ_5 cannot be implemented with less than eight logical instructions. Note that `(~x & y)` is a single op since it corresponds to the ANDN machine instruction (equivalent instruction is available at least in Intel, Power, ARM, Alpha SIMD architectures).

Since transposing a binary matrix is generally slow in software, one would typically want to combine two mx operations into a double-round with separate ``vertical'' and ``horizontal'' parts.

Figure 4.3 shows how the state fits into the register sets of various CPU architectures. Here's our data type definition which makes data accessible in various ways:

```
// 256-bit state
typedef union w256 {
    uint8_t b[32];          // bytes (octets)
    uint16_t w[16];         // words (16-bit)
    uint32_t d[8];          // doublewords (32-bit)
    uint64_t q[4];          // quadwords (64-bit)
#ifdef __AVX2__
    __m256i y;              // a single 256-bit integer type
#endif
} cbeam_w256;
```

We only give some generic guidance on how to implement mx^2 in software this way. One should examine the reference 16-bit, 64-bit, and 256-bit implementations for architecture-specific optimizations.

Step 1: Vertical linear transform λ

This step is easiest to implement by viewing the state as 64-bit words (``quadwords'') $s_q[0..3]$ with $\mathbf{s} = (\, s_q[\,0\,],\ s_q[\,1\,],\ s_q[\,2\,],\ s_q[\,3\,]\,)$.

$$\begin{aligned} t &= s_q[\,0\,]\ \oplus\ s_q[\,1\,]\ \oplus\ s_q[\,2\,]\ \oplus\ s_q[\,3\,] \\ \mathbf{s}' &= (s_q[\,0\,]\ \oplus\ t,\ s_q[\,1\,]\ \oplus\ t,\ s_q[\,2\,]\ \oplus\ t,\ s_q[\,3\,]\ \oplus\ t\,). \end{aligned} \tag{4.1}$$

Equivalent C code:

```
// 256-bit vertical Linear Mix
t1 = cb->q[0] ^ cb->q[1] ^ cb->q[2] ^ cb->q[3];
cb->q[0] ^= t1;
cb->q[1] ^= t1;
cb->q[2] ^= t1;
cb->q[3] ^= t1;
```

Step 2: Vertical nonlinear transform

We use the optimized bitslice method to compute one or more (depending on system word size) instances of ϕ_{16} simultaneously. For $0 \leq i \leq 15$ we can write with 16-bit words:

$$\begin{aligned} s'_w[\,i\,] = \phi_5\big(&s_w[\,i\,], s_w[\,(i-1) \bmod 16\,], s_w[\,(i-2) \bmod 16\,], \\ &s_w[\,(i-3) \bmod 16\,], s_w[\,(i-4) \bmod 16\,]\big). \end{aligned} \tag{4.2}$$

In 64-bit implementations two adjacent input 64-words are needed to construct the five words shifted versions. We can compute four parallel ϕ_{16} functions:

```
// 4x Vertical Nonlinear Mix on t1
t1 = cb->q[i];
t5 = cb->q[(i - 1) & 3];
t2 = (t1 << 16) ^ (t5 >> 48);
t3 = (t1 << 32) ^ (t5 >> 32);
t4 = (t1 << 48) ^ (t5 >> 16);
t1 = CBEAM_PHI5(t1, t2, t3, t4, t5);

// .. proceed to RC / horiz. Lambda with t1
```

Step 3: Round Constant

As the round constants are only active at odd rounds, they are in fact always applied between vertical and horizontal rounds in this type of implementation. Written as transposed quadwords, the three nonzero round constants are:

$$\begin{aligned} \mathsf{rc}_q^1 &= \mathtt{0x2000040000300009} \\ \mathsf{rc}_q^3 &= \mathtt{0x6000050000100008} \\ \mathsf{rc}_q^3 &= \mathtt{0xA0000C000070000B} \end{aligned} \tag{4.3}$$

Constants from Equation 4.3 are XORed over the first 64-bit word of state at round i:

$$s'_q[\,0\,] = s_q[\,0\,] \oplus \mathsf{rc}_q^i. \tag{4.4}$$

```
    // Round constants
    const uint64_t rc[3] =
        { 0x2000040000300009, 0x6000050000100008, 0xA0000C000070000B };
```

Step 4: Horizontal linear transform λ

There are many ways to implement this step -- it is a simple 4-bit nibble XOR parity operation. The step is highly parallelizable. For $0 \le i \le 15$ we can write it with 16-bit rotations:

$$\begin{aligned} t &= s_w[\,i\,] \;\oplus\; (s_w[\,i\,] \lll 4) \;\oplus\; (s_w[\,i\,] \lll 8) \;\oplus\; (s_w[\,i\,] \lll 12) \\ s'_w[\,i\,] &= s_w[\,i\,] \;\oplus\; t. \end{aligned} \tag{4.5}$$

The following piece of C computes four horizontal linear transforms in parallel on a 64-bit register:

```
        // 4x Horizontal Linear Mix on t1
        t2 = t1;
        t2 ^= t2 >> 8;
        t2 ^= t2 >> 4;
        t2 &= 0x000F000F000F000Fllu;
        t2 ^= t2 << 4;
        t2 ^= t2 << 8;
        t1 ^= t2;
```

Step 5: Horizontal nonlinear transform

Again a bit-slicing implementation of ϕ_5 is used, but on rotated values of each word. For $0 \le i \le 15$:

$$s'_w[\,i\,] = \phi_5\big(s_w[\,i\,], s_w[\,i\,] \lll 1, s_w[\,i\,] \lll 2, s_w[\,i\,] \lll 3, s_w[\,i\,] \lll 4\big). \tag{4.6}$$

This step is slightly more complicated to parallelize:

```
        // 4x Horizontal Nonlinear Mix on t1
        t2 = ((t1 << 1) & 0xFFFEFFFEFFFEFFFEllu) ^
             ((t1 >> 15) & 0x0001000100010001llu);
        t3 = ((t1 << 2) & 0xFFFCFFFCFFFCFFFCllu) ^
             ((t1 >> 14) & 0x0003000300030003llu);
        t4 = ((t1 << 3) & 0xFFF8FFF8FFF8FFF8llu) ^
             ((t1 >> 13) & 0x0007000700070007llu);
        t5 = ((t1 << 4) & 0xFFF0FFF0FFF0FFF0llu) ^
             ((t1 >> 12) & 0x000F000F000F000Fllu);

        t1 = CBEAM_PHI5(t1, t2, t3, t4, t5);
```

4.5 Sensors and Pervasive Devices: MSP430

Texas Instruments MSP430 is a well known family of low-cost and ultra-low power 16-bit SoC micro-controllers, widely used in sensor networks. CBEAM beats the more than dozen MSP430 encryption algorithm implementations reported in [18], often by an order of magnitude.

Our implementation of π is able to execute entirely on 12 general-purpose registers without having to resort to stack (except the top value) and therefore the running RAM requirement is equivalent to the state size, 32 bytes. The ϕ_5 function was realized with nine logic instructions. The additional logic instruction caused by the fact that MSP430 only has two-operand machine instructions.

Unfortunately the MSP430 instruction set only has single-bit shifts and no multi-bit rotation instruction, which created a bottleneck for "horizontal" λ.

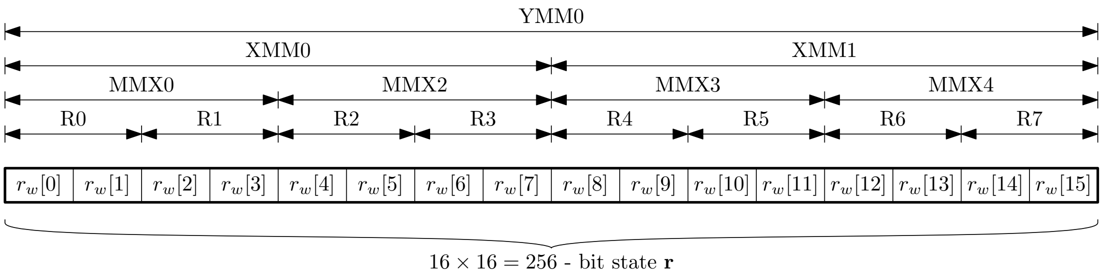
YMM0
XMM0
XMM1
MMX0
MMX2
MMX3
MMX4
R0
R1
R2
R3
R4
R5
R6
R7
$r_w[0]$ $r_w[1]$ $r_w[2]$ $r_w[3]$ $r_w[4]$ $r_w[5]$ $r_w[6]$ $r_w[7]$ $r_w[8]$ $r_w[9]$ $r_w[10]$ $r_w[11]$ $r_w[12]$ $r_w[13]$ $r_w[14]$ $r_w[15]$
$16 \times 16 = 256$ - bit state $\mathbf{r}$

Figure 4.3: Illustration on how to fit the 256-bit state into a single Haswell+ AVX2 YMM register, two Pentium 3+ SSE XMM registers, four Pentium+ MMX or ARM NEON registers or eight ARM general purpose registers for bit-slicing computation.

```
        /* r14 = Phi5(r15,r14,r13,r12,r11) */
        bic     r12, r11
        inv     r13
        and     r13, r12
        and     r11, r13
        xor     r12, r11
        and     r11, r15
        bis     r12, r14
        bic     r15, r14
        xor     r13, r14
```

The cipher is as fast as the very fastest AES implementations on this platform but has significantly smaller implementation footprint. The following numbers are only for cores, modes of operation not included. The IAIK [22] implementation is commercial and written in hand-optimized assember. The Texas Instruments [33] implementation is recommended by the SoC vendor. Note that GCM is nontrivial to implement on this platform and we can expect it to add a significant overhead to the AES numbers.

Code	**Flash**	**RAM**	**Encryption**	**Decryption**	**Cycles / Byte**
CBEAM	386 B	32 B	4369 C	4404 C	550.5
AES-128 [22]	2536 B	?	5432 C	8802 C	550.1
AES-128 [33]	2423 B	80 B	6600 C	8400 C	525.0
AES-256 [22]	2830 B	?	7552 C	12258 C	766.1

4.6 Latest Server / Desktop / Laptop Systems: x86-64 with AVX2

The Intel Haswell (Generation 4 Core) and later x86-64 CPUs support 256-bit AVX2 (Advanced Vector Extensions 2) SIMD instructions. The AVX2 platform provides shuffle and vector shift instructions for 16-bit vector sub-units in addition to 256-bit Boolean logic for the nonlinear function ϕ_5 (Equation 1.5). We can implement full 256-bit ϕ_5 with only eight instructions. This roughly doubles the overall execution speed when compared to optimized 64-bit gcc versions.

Here is a code snippet written in AVX2 C intrinsics for implementing the ϕ_5 function with 8 logical instructions on 256-bit registers:

```
        // t0 = Phi5(x0,x1,x2,x3,x4)
        t0 = _mm256_andnot_si256(x3, x4);
        t1 = _mm256_andnot_si256(x2, x3);
        t2 = _mm256_andnot_si256(x2, t0);
        t3 = _mm256_or_si256(x1, t1);
        t0 = _mm256_xor_si256(t0, t1);
        t1 = _mm256_and_si256(x0, t0);
        t0 = _mm256_andnot_si256(t1, t3);
        t0 = _mm256_xor_si256(t0, t2);
```

Please see the reference implementation file `ymm/mx6-avx2.c` for tricks on how to implement λ and various shifts efficiently on this platform.

The following speeds were measured on a MacBook Air (Q3/2013) with Intel Core i5 - 4250U CPU @1.30 GHz running Ubuntu Linux 13.04. Linux reported internal clock frequency as 1.90 GHz during the tests. We compare to the OpenSSL 1.0.1e AES implementation, which is the de facto standard AES implementation. Generic optimizations were enabled but we disabled the full hardware AES for fairness.

Implementation	**Troughput**	**Cycles / Byte**
CBEAM-GCC	58.5 MB/s	32.5
CBEAM-AVX2	117.5 MB/s	16.1
OpenSSL AES-128	106.5 MB/s	17.8
OpenSSL AES-192	86.0 MB/s	22.1
OpenSSL AES-256	71.9 MB/s	26.4

These are wall-clock measurements. Note that the cycles / byte numbers are calculated directly from the internal clock frequency and throughput, and are therefore influenced by I/O and other factors.

Chapter 5

Design Rationale

CBEAM design and analysis were interdependent and were largely performed concurrently. Therefore detailed design rationale is more completely given in Chapter 3, "Security Analysis". Here we just give some broad rationale to our overall component and parameter selection.

5.1 Design of $\pi = \mathbf{mx}^6$

See Section 3.2:

- The round constants were optimized to offer resistance against rotational and slide (Section 3.2.1).
- We exhaustively examined all 2^{32} 5-input Boolean functions when selecting ϕ_5, which was optimal in that class. (Section 3.2.2).
- The design offers resistance to classical Differential and Linear cryptanalysis (Section 3.2.3). CBEAM has exceptional Algebraic properties (Section 3.2.4).
- Six rounds provides very efficient mixing of data (Tables 3.3 and 3.4). However this is not a block cipher and a distinguisher for the mixing function (isolated from the mode of operation) should not be considered a fatal weakness.

We also wanted to avoid data-conditional processing and lookup tables to minimize opportunities for timing attacks. CBEAM can be implemented entirely in straight-line code without table lookups, making it more resistant to side-channel attacks than some alternatives.

5.2 Generic notes on Sponge Parameters

- The Sponge construct is a flexible method for building cryptographic algorithms of different kinds from a single permutation.
- The Sponge parameters (Section 1.3) were derived from well-established theorems regarding Sponge functions and a large body of research. [4, 5, 6, 7, 8, 10].
- The BLNK padding variant as used in CBEAM allows flexible later extensions such as "Parallelized Tree AEAD" without breaking the core.

5.3 Hidden Weaknesses

The designer of CBEAM has not hidden any weaknesses in this cipher. We are not aware of any method for hiding weaknesses to an algorithm of this type, as it has no large constants, tables, or other special cases. Shift-Invariance of Theorem 2 (Section 3.2.1) indicates that the π function output is *homogeneous*. Therefore there are no "special bits" that would leak out secret information.

Chapter 6

Intellectual Property and Consent

6.1 Intellectual Property

A patent application was filed by the submitter in December 2013 with USPTO. The application was filed to protect CBEAM against claims by third parties and to secure its free usage by anyone, should a version get accepted into the CAESAR portfolio.

This application covers the design and some software and hardware implementation techniques of CBEAM, and thereby CBEAM is "patent pending." However, we will withdraw this application if required by the CAESAR selection committee.

The author/inventor grants permission to use CBEAMr1 (as specified in this document) in any way, in any application, free of charge, indefinitely. However, this permission does not apply to derived or modified works without separate, specific consent.

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

6.2 Consent to CAESAR Selection Committee

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 they disagree with published analyses then they are 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.

Dr. Markku-Juhani O. Saarinen
Trondheim, NORWAY
March 15, 2014

Bibliography

[1] Aciiçmez, O., Schindler, W., and Ç. K. Koç. Cache based remote timing attack on the AES. In *CT-RSA 2007* (2007), M. Abe, Ed., vol. 4377 of *LNCS*, Springer, pp. 271--286.

[2] Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., and Wingers, L. The SIMON and SPECK families of lightweight block ciphers. IACR ePrint 2013/404, `http://eprint.iacr.org/2013/404`, June 2013.

[3] Bernstein, D. J. Cache-timing attacks on AES. Tech. rep., University of Chigaco, 2005.

[4] Bertoni, G., Daemen, J., Peeters, M., and Assche, G. V. Sponge functions. In *Ecrypt Hash Workshop 2007* (May 2007).

[5] Bertoni, G., Daemen, J., Peeters, M., and Assche, G. V. Sponge-based pseudo-random number generators. In *CHES 2010* (2010), S. Mangard and F.-X. Standaert, Eds., vol. 6225 of *LNCS*, Springer, pp. 33--47.

[6] Bertoni, G., Daemen, J., Peeters, M., and Assche, G. V. Cryptographic sponge functions, version 0.1. `http://sponge.noekeon.org/`, STMicroelectronics and NXP Semiconductors, January 2011.

[7] Bertoni, G., Daemen, J., Peeters, M., and Assche, G. V. Duplexing the sponge: Single-pass authenticated encryption and other applications. In *SAC 2011* (2011), A. Miri and S. Vaudenay, Eds., vol. 7118 of *LNCS*, Springer, pp. 320--337.

[8] Bertoni, G., Daemen, J., Peeters, M., and Assche, G. V. On the security of the keyed sponge construction. In *SKEW 2011 Symmetric Key Encryption Workshop* (February 2011).

[9] Bertoni, G., Daemen, J., Peeters, M., and Assche, G. V. Permutation-based encryption, authentication and authenticated encryption. In *DIAC 2012* (2012). `http://keccak.noekeon.org/KeccakDIAC2012.pdf`.

[10] Bertoni, G., Daemen, J., Peeters, M., and Assche, G. V. Sakura: a flexible coding for tree hashing. IACR ePrint 2013/213, `http://eprint.iacr.org/2013/213`, April 2013.

[11] Biham, E. A fast new DES implementation in software. In *FSE 1997* (1997), E. Biham, Ed., vol. 1267 of *LNCS*, Springer, pp. 260--272.

[12] Biham, E., and Shamir, A. *Differential Cryptanalysis of the Data Encryption Standard.* Springer, 1993.

[13] Biryukov, A., and Wagner, D. Slide attacks. In *FSE 1999* (1999), L. R. Knudsen, Ed., vol. 1636 of *LNCS*, Springer, pp. 245--259.

[14] Biryukov, A., and Wagner, D. Advanced slide attacks. In *EUROCRYPT 2000* (2000), B. Preneel, Ed., vol. 1807 of *LNCS*, Springer, pp. 589--606.

[15] Bogdanov, A., Knudsen, L. R., Leander, G., Paar, C., Poschmann, A., Robshaw, M. J. B., Seurin, Y., and Vikkelsoe, C. PRESENT: An ultra-lightweight block cipher. In *CHES 2007* (2007), P. Paillier and I. Verbauwhede, Eds., vol. 4727 of *LNCS*, Springer, pp. 450--466.

[16] Boura, C., and Canteaut, A. On the influence of the algebraic degree of F^{-1} on the algebraic degree of $G \circ F$. *IEEE Transactions on Information Theory 59*, 1 (January 2013).

[17] Cannière, C. D., O.Dunkelman, and zević., M. K. KATAN & KTANTAN -- a family of small and efficient hardware-oriented block ciphers. In *CHES 2009* (2009), C. Clavier and K. Gaj, Eds., vol. 5747 of *LNCS*, Springer, pp. 272--288.

[18] Cazorla, M., Marquet, K., and Minier, M. Survey and benchmark of lightweight block ciphers for wireless sensor networks. In *SECRYPT 2013* (May 2013). `http://eprint.iacr.org/2013/295`.

[19] Daemen, J. *Cipher and Hash Function Design Strategies based on linear and differential cryptanalysis*. PhD thesis, K.U. Leuven, March 1995.

[20] Ferguson, N., and Schneier, B. *Practical Cryptography*. John Wiley & Sons, 2003.

[21] Hiltgen, A. P. Towards a better understanding of one-wayness: Facing linear permutations. In *EUROCRYPT '98* (1998), K. Nyberg, Ed., vol. 1403 of *LNCS*, Springer, pp. 319--333.

[22] IAIK. AES for Texas Instruments MSP430 microcontrollers. Tech. rep., IAIK SIC T. U. Graz. `http://jce.iaik.tugraz.at/sic/Products/Crypto_Software_for_Microcontrollers`.

[23] Khovratovich, D., and Nikolić, I. Rotational cryptanalysis of ARX. In *FSE 2010* (2010), S. Hong and T. Iwata, Eds., vol. 6147 of *LNCS*, Springer, pp. 333--346.

[24] Matsui, M. The first experimental cryptanalysis of the data encryption standard. In *CRYPTO '94* (1994), Y. Desmedt, Ed., vol. 839 of *LNCS*, Springer, pp. 1--11.

[25] Matsui, M. Linear cryptoanalysis method for DES cipher. In *EUROCRYPT '93* (1994), T. Helleseth, Ed., vol. 765 of *LNCS*, Springer, pp. 386--397.

[26] Rogaway, P., Bellare, M., and Black, J. OCB: A block-cipher mode of operation for efficient authenticated encryption. *ACM Transactions on Information and System Security (TISSEC) 6*, 3 (August 2003), 365--403.

[27] Saarinen, M.-J. O. Chosen-IV statistical attacks against eSTREAM ciphers. In *Proc. SECRYPT 2006, International Conference on Security and Cryptography, Setubal, Portugal, August 7-10, 2006.* (2006).

[28] Saarinen, M.-J. O. Related-key attacks against full Hummingbird-2. In *FSE 2013: 20th International Workshop on Fast Software Encryption. 11-13 March 2013, Singapore, Singapore* (2013). To Appear.

[29] Saarinen, M.-J. O. Beyond modes: Building a secure record protocol from a cryptographic sponge permutation. In *CT-RSA 2014: Cryptographers' Track, RSA Conference USA, 25--28 February 2014, San Francisco, USA* (2014), Springer. To Appear.

[30] Saarinen, M.-J. O. CBEAM: Efficient authenticated encryption from feebly one-way phi functions. In *CT-RSA 2014: Cryptographers' Track, RSA Conference USA, 25--28 February 2014, San Francisco, USA* (2014), Springer. To Appear.

[31] Saarinen, M.-J. O. StriBob: Authenticated encryption from GOST R 34.11-2012 LPS permutation (extended abstract). Submitted for Publication, February 2014.

[32] Saarinen, M.-J. O. The STRIBOBr1 authenticated encryption algorithm. CAESAR First Round Candidate, `http://www.stribob.com`, March 2014.

[33] TI. AES128 - A C implementation for encryption and decryption. Tech. Rep. SLAA397A, Texas Instruments, July 2009. `http://www.ti.com/lit/an/slaa397a/slaa397a.pdf`.

[34] Wagner, D., and Schneier, B. Analysis of the SSL 3.0 protocol. In *The Second USENIX Workshop on Electronic Commerce Proceedings* (November 1996), USENIX Press, pp. 29--40.

[35] Weiss, M., Heinz, B., and Stumpf, F. A cache timing attack on AES in virtualization environments. In *FC 2012* (2013), A. Keromytis, Ed., vol. 7397 of *LNCS*, Springer, pp. 314--328.

[36] Yalçin, T., and Kavun, E. B. On the implementation aspects of sponge-based authenticated encryption for pervasive devices. In *CARDIS 2012* (2013), S. Mangard, Ed., vol. 7771 of *LNCS*, Springer, pp. 141--157.

Document version 20140315200000.