STRIBOBr2: "WHIRLBOB"

Second Round CAESAR Algorithm Tweak Specification

Pricipal Submitter
Markku-Juhani O. Saarinen
m.saarinen@qub.ac.uk
Queen's University Belfast, UK

Billy B. Brumley
billy.brumley@tut.fi
Tampere University of Technology, Finland

August 28, 2015

For updates and further information:

http://www.stribob.com

Contents

Preface and Introduction

This document describes the STRIBOBr2 ("WHIRLBOB") Authenticated Encryption with Associated Data (AEAD) algorithm.

- **STRIBOBr1** was Markku's first-round proposal. For details of STRIBOBr1, we refer to round one documentation [56, 57]. Analysis of STRIBOBr1 is still encouraged; this variant will continue to coexist with STRIBOBr2.
- **STRIBOBr2** is our current **primary recommendation** for CAESAR Round 2 and is equivalent to the **WHIRLBOB** algorithm discussed in [58].

Note that version naming convention has been simplified from one previously proposed.

This 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`

Chapter 6, "Changes", details the technical differences between STRIBOBr1 and STRIBOBr2. The tweak is motivated purely by performance and other implementation factors; no attacks have been discovered against STRIBOBr1.

STRIBOBr2 borrows its S-Box and MDS matrix design from the well known hash function Whirlpool, which has been around for 15 years. Whirlpool has received a substantial amount of public cryptanalysis, much of which is directly applicable to STRIBOBr2. As in STRIBOBr1 [56], the number of rounds of STRIBOBr2 is 12 (rather than 10 as in original Whirlpool) making it resistant to all known attacks.

Another advantage of STRIBOBr2 is in implementation techniques. The new (or rather, old, Whirlpool-originated) S-Box design allows very efficient constant-time implementation on current ARM, Intel, and AMD architectures. We are happy to report that these constant time implementations are faster than latest OpenSSL AES-192 (matching security level) "de facto standard" implementation on typical PC hardware when AES NI instructions are not used. The design also facilitates efficient lightweight and hardware implementation.

This is the version 2.20150828143000 of this document. We urge the reader to check for updates, revisions, and reference data at:

`http://www.stribob.com`

If you find bugs, typos, obvious security blunders, or clever cryptanalytic attacks, we are very interested to hear about that. Our e-mail address can be found on the front page.

Cheers and have fun,

- **Markku**, Belfast and **Billy**, Tampere

Chapter 1

Specification

STRIBOBr2 ("WhirlBob") is an algorithm for Authenticated Encryption with Associated Data (AEAD). The STRIBOBr2 design is flexible enough to accept almost arbitrary data ranges as its input parameters; however for CAESAR we propose a concrete parameter set as follows:

Secret key size	192 bits	`CRYPTO_KEYBYTES`	24
Secret sequence number	*not used*	`CRYPTO_NSECBYTES`	0
Public sequence number (nonce)	128 bits	`CRYPTO_NPUBBYTES`	16
Authentication tag (message expansion)	128 bits	`CRYPTO_ABYTES`	16

We note that this is a specification for STRIBOBr2 a.k.a "WHIRLBOB" **only**. For details of STRIBOBr1, we refer to round one documentation [55, 56, 57]. STRIBOBr2 is our current primary recommendation.

STRIBOBr2 is defined by a 512×512 - bit permutation π and BLNK mode of operation and padding. The permutation $\pi(x)$ does not take any other input variables apart from x. Computation of π follows almost exactly the operation of the "internal key schedule" of Whirlpool 3.0 [4]. The only modification is that the number of rounds is increased from $R = 10$ to $R = 12$ for extra security margin against Rebound Attacks [36, 37]. [1] BLNK is a simple Sponge AEAD mode derived from the Blinker protocol [53].

The algorithm specification is structured as follows. We first define the π permutation in Section 1.1, then the BLNK Sponge mode of operation in Section 1.2, and finally offer some test vectors in Section 1.3.

1.1 Structure of the π Permutation

To compute $\pi(x_0) = x_{12}$ we iterate a $L \circ P \circ S$ composite mixing function with round constants C_r.

$$x_{r+1} = L(P(S(x_r))) \oplus C_r \quad for\ rounds \quad 0 \leq r < 12. \tag{1.1}$$

We write the 512-bit state as a matrix $M[\,0 \cdots 7\,][\,0 \cdots 7\,]$ of 8×8 bytes, which can be serialized as a 64-byte sequence $V[\,0 \cdots 63\,]$ with $V[\,8i + j\,] = M[\,i\,][\,j\,]$.

1.1.1 Substitution S or `SubBytes`

In this step each of the 64 bytes in the state is substituted using a 8×8 - bit S-Box S, defined by Table 1.1.

$$M'[\,i\,][\,j\,] \leftarrow S(M[\,i\,][\,j\,]) \quad for \quad 0 \leq i, j < 8. \tag{1.2}$$

Note that 8-bit S can be implemented using 4-Bit "miniboxes" of Table 1.2 as shown in Figure 1.1.

1.1.2 Permutation P or `ShiftColumns`

In this step the state bytes are moved around as follows:

$$M'[\,(\,i + j\,) \bmod 8\,][\,j\,] \leftarrow M[\,i\,][\,j\,] \quad for \quad 0 \leq i, j < 8. \tag{1.3}$$

[1] These attacks would not be directly applicable with the BLNK mode anyway. This is because the attacker can never access more than r bits of the internal state. [40]

Table 1.1: STRIBOBr2 S-Box S.

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	18	23	C6	E8	87	B8	01	4F	36	A6	D2	F5	79	6F	91	52
1x	60	BC	9B	8E	A3	0C	7B	35	1D	E0	D7	C2	2E	4B	FE	57
2x	15	77	37	E5	9F	F0	4A	DA	58	C9	29	0A	B1	A0	6B	85
3x	BD	5D	10	F4	CB	3E	05	67	E4	27	41	8B	A7	7D	95	D8
4x	FB	EE	7C	66	DD	17	47	9E	CA	2D	BF	07	AD	5A	83	33
5x	63	02	AA	71	C8	19	49	D9	F2	E3	5B	88	9A	26	32	B0
6x	E9	0F	D5	80	BE	CD	34	48	FF	7A	90	5F	20	68	1A	AE
7x	B4	54	93	22	64	F1	73	12	40	08	C3	EC	DB	A1	8D	3D
8x	97	00	CF	2B	76	82	D6	1B	B5	AF	6A	50	45	F3	30	EF
9x	3F	55	A2	EA	65	BA	2F	C0	DE	1C	FD	4D	92	75	06	8A
Ax	B2	E6	0E	1F	62	D4	A8	96	F9	C5	25	59	84	72	39	4C
Bx	5E	78	38	8C	D1	A5	E2	61	B3	21	9C	1E	43	C7	FC	04
Cx	51	99	6D	0D	FA	DF	7E	24	3B	AB	CE	11	8F	4E	B7	EB
Dx	3C	81	94	F7	B9	13	2C	D3	E7	6E	C4	03	56	44	7F	A9
Ex	2A	BB	C1	53	DC	0B	9D	6C	31	74	F6	46	AC	89	14	E1
Fx	16	3A	69	09	70	B6	D0	ED	CC	42	98	A4	28	5C	F8	86

Table 1.2: Three 4×4 miniboxes that are used to build the 8×8 S-Box S.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$E(x)$	1	B	9	C	D	6	F	3	E	8	7	4	A	2	5	0
$E^{-1}(x)$	F	0	D	7	B	E	5	A	9	2	C	1	3	4	8	6
$R(x)$	7	C	B	D	E	4	9	F	6	3	8	A	2	5	1	0

1.1.3 Linear operation L or `MixRows`

Each of the 8 row vectors

$$W_i = (\, M[\, i\,][\, 0\,], M[\, i\,][\, 1\,], \cdots M[\, i\,][\, 7\,]\,) \tag{1.4}$$

is individually multiplied by a circulant, low-weight 8×8 MDS matrix in the finite field GF(2^8) characterized by primitive polynomial $p(x) = x^8 + x^4 + x^3 + x^2 + 1$.

$$W_i' = W_i \cdot \begin{pmatrix} 01 & 01 & 04 & 01 & 08 & 05 & 02 & 09 \\ 09 & 01 & 01 & 04 & 01 & 08 & 05 & 02 \\ 02 & 09 & 01 & 01 & 04 & 01 & 08 & 05 \\ 05 & 02 & 09 & 01 & 01 & 04 & 01 & 08 \\ 08 & 05 & 02 & 09 & 01 & 01 & 04 & 01 \\ 01 & 08 & 05 & 02 & 09 & 01 & 01 & 04 \\ 04 & 01 & 08 & 05 & 02 & 09 & 01 & 01 \\ 01 & 04 & 01 & 08 & 05 & 02 & 09 & 01 \end{pmatrix} \quad for \quad 0 \le i < 8. \tag{1.5}$$

1.1.4 Constants C_r or `AddRoundKey`

Blocks of eight bytes from the S-Box are used as round keys C_r for the first row. The rest of the rows are unaffected by C_r. For round $0 \le r < 12$:

$$M'[\, 0\,][\, j\,] \;\leftarrow\; M[\, 0\,][\, j\,] \oplus S(8r + j) \quad for \quad 0 \le j < 8. \tag{1.6}$$

$$M'[\, i\,][\, j\,] \;\leftarrow\; M[\, i\,][\, j\,] \quad for \quad 1 \le i < 8 \; and \; 0 \le j < 8. \tag{1.7}$$

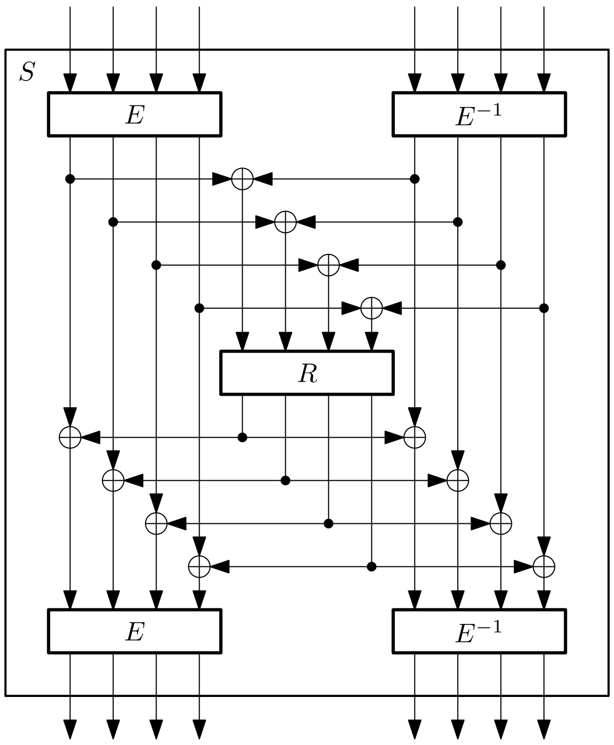

Figure 1.1: The STRIBOBr2 8×8 - bit S-Box is constructed from three 4×4 - bit "miniboxes". In this diagram the most significant bits are on the left: E operates on the higher nibble.

1.1.5 Code for π Computation

Rather than repeating the definition of the above sections in some ad hoc pseudocode, we offer a platform-independent ANSI C code for π to help implementors. It is reproduced here just for reference – this code should not be used for production as much more efficient implementations exist. See Chapter 4 for information about optimization techniques for various targets.

The function `wbob_pi()` computes π in place on byte vector `st[64]`. The only external variable referenced is `wbob_sbox[256]`, which has equivalent contents to Table 1.1.

```
void wbob_pi(uint8_t st[64])                                    // STRIBOBr2 Pi
{
    int r, i, j;                                                // loop variables
    uint8_t t[64], x, *pt;                                      // work variables

    for (r = 0; r < 12; r++) {                                  // 12 rounds
        for (i = 0; i < 64; i++) {
            t[(i & 7) + ((i + (i << 3)) & 070)] =               // P Permutation..
                wbob_sbox[st[i]];                               // ..with S-Box
        }

        // Round constants C come from the S-box
        pt = (uint8_t *) &wbob_sbox[8 * r];
        for (i = 0; i < 8; i++)
            st[i] = pt[i];                                      // C in first 8
        for (i = 8; i < 64; i++)
            st[i] = 0;                                          // zero the rest

        // Apply the circular, low weight MDS matrix
        for (i = 0; i < 64; i += 8) {
            pt = &st[i];                                // start of row
            for (j = 0; j < 8; j++) {
                x = t[i + j];                           // Circular MDS
                pt[j & 7] ^= x;                         // 01
                pt[(j + 1) & 7] ^= x;                   // 01
                pt[(j + 3) & 7] ^= x;                   // 01
                pt[(j + 5) & 7] ^= x;                   // odd
                pt[(j + 7) & 7] ^= x;                   // odd

                x = (x << 1) ^ (x & 0x80 ? 0x1D : 0x00); // double x
                pt[(j + 6) & 7] ^= x;                   // 02

                x = (x << 1) ^ (x & 0x80 ? 0x1D : 0x00); // redouble to 4x
                pt[(j + 2) & 7] ^= x;                   // 04
                pt[(j + 5) & 7] ^= x;                   // 01 + 04 = 05

                x = (x << 1) ^ (x & 0x80 ? 0x1D : 0x00); // redouble to 8x
                pt[(j + 4) & 7] ^= x;                   // 08
                pt[(j + 7) & 7] ^= x;                   // 01 + 08 = 09
            }
        }
    }
}
```

1.2 BLNK Sponge Mode and Padding

BLNK (”Blink”) is a general and highly flexible Sponge mode of operation modified from the Sponge padding used in the original Blinker [53] lightweight two-party protocol.

In this section we describe only how it is used specifically in the STRIBOBr2 Authenticated Encryption with Associated Data (AEAD) algorithm, ignoring many of its more advanced uses and features.

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

b	State size. π has $b = 512$ bits.
r	Data rate or block size. $r = 256$ bits.
c	Capacity, the amount of secret information in the state. $c = b - r - \delta$ bits.
δ	Information Theoretic capacity consumed by padding. We can bound this to $\delta \leq 2$ bits.

Furthermore, we fix the key size to $k = 192$ bits and the authentication tag to $t = 128$ bits. Authentication tags are concatenated with ciphertext for final output.

1.2.1 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($\lceil n/32 \rceil, 1$) times.

The four basic operations are:

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

Table 1.3: Constant bytes used in STRIBOBr2 BLNK Padding.

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_AAD`	`0x40`	Authenticated Associated Data (in).
`BLNK_MSG`	`0x50`	Confidential Message Payload (in/out).
`BLNK_MAC`	`0x60`	Message Authentication Code (out).

In the following generic pseudocode $\mathsf{op} \in \{\mathsf{put}, \mathsf{get}, \mathsf{enc}, \mathsf{dec}\}$ and $V[\,0 \cdots 63\,]$ is the state. The padding argument pad is made up as a combination of constants given in Table 1.3.

```
1:  $i \leftarrow 0$                                          state index, initialized to first byte
2:  for $j = 0$ to $n - 1$ do
3:    if $i = 32$ then
4:      $V[32] \leftarrow V[32] \oplus$ 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 $\mathsf{op} = \mathsf{put}$ then
9:      $V[i] \leftarrow V[i] \oplus D[j]$                     XOR input data to the state
10:   else if $\mathsf{op} = \mathsf{get}$ then
11:     $D[j] \leftarrow V[i]$                                 simply save the data
12:   else if $\mathsf{op} = \mathsf{enc}$ 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 $\mathsf{op} = \mathsf{dec}$ 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: $i = 32$ possible)
22: $V[32] \leftarrow V[32] \oplus$ BLNK_FIN $\oplus pad$      final padding
23: $V \leftarrow \pi(V)$                                      final cryptographic permutation
```

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

We describe how STRIBOBr2 is used with the Application Programming Interface (API) defined in the CAESAR call [21]. 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.

$K[\,24\,]$	Secret key of $k = 192$ bits, or 24 bytes.
$N[\,16\,]$	A 128-bit public nonrepeating sequence number (nonce) for the message. Only integrity is protected for this data. Contents are not transmitted in ciphertext.
$A[\,a\,]$	Associated Authenticated data, a bytes. Only integrity is protected for this data. The contents are not transmitted in ciphertext. If unused, set $a = 0$.
$P[\,n\,]$	Plaintext payload, $0 \leq n$ bytes. Integrity and confidentiality is protected for this data.
$C[\,n + 16\,]$	Ciphertext, $16 \leq n + 16$ bytes. Integrity and confidentiality is protected for this data.

Pseudocode for implementing standard CAESAR AEAD API encryption.

```
$C[\,n+16\,] \leftarrow$ encrypt( $K[\,24\,]$, $N[\,16\,]$, $A[\,a\,]$, $P[\,n\,]$ )
V[64] ← (0,0,···,0)                        initialize the state with zeros
put( K[24], BLNK_KEY )                      secret key, always a single π op
put( N[16], BLNK_NPUB )                     public nonce, always a single π op
put( A[a], BLNK_AAD )                       authenticated data, ⌈a/32⌉ π ops
C[0···n−1] ← enc( P[n], BLNK_MSG )          encryption, ⌈n/32⌉ π ops
C[n···n+15] ← get( 16 , BLNK_MAC )          message authentication code, π not necessary
return C[n+16]                              authenticated ciphertext
```

Inverse operation by the recipient:

```
{ P[n] or FAIL } ← decrypt( K[24], N[16], A[a], C[n+16] )
V[64] ← (0,0,···,0)                        initialize the state with zeros
put( K[24], BLNK_KEY )                      secret key, always a single π op
put( N[16], BLNK_NPUB )                     public nonce, always a single π op
put( A[a], BLNK_AAD )                       associated authenticated data
P[n] ← dec( P[0···n−1], BLNK_MSG )          decryption
if C[n···n+15] = get( 16 , BLNK_MAC ) then
    return P[n]                             auth match: C[n···n+15] = V[0···15]
else
    return FAIL                             plaintext should be ignored (and cleared)
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. Confidential state should be cleared in order to minimize leakage before exit.

1.3 Test Vectors

1.3.1 The 12-round π transform

These vectors are derived from ISO Test vectors of Whirlpool 3.0 [30]. We give the input x_0 and results after 1, 10 (as in Whirlpool), and full 12 rounds of processing.

$$x_0 = \begin{pmatrix} \texttt{77 38 E1 B5 41 A0 36 EA} \\ \texttt{45 8D 50 F8 0F A0 1C 44} \\ \texttt{72 88 CE 97 D1 A0 DC F0} \\ \texttt{16 95 FF D6 E7 1D 09 25} \\ \texttt{33 BE 30 9F 01 2A 59 09} \\ \texttt{72 91 14 59 5F 08 6E 76} \\ \texttt{07 18 AF E3 65 BC 09 DE} \\ \texttt{B6 AF A1 80 BC EC 2A 98} \end{pmatrix} \quad x_1 = \begin{pmatrix} \texttt{1A 78 4D 7D BD 4C 17 E6} \\ \texttt{27 31 10 AA 63 C5 9E 25} \\ \texttt{7A 2E B7 48 C4 5D E0 23} \\ \texttt{6D 0D 61 9F 6C 1D 80 AE} \\ \texttt{01 A2 D5 6E DB 41 D9 A0} \\ \texttt{E9 06 4C D1 27 95 FA 86} \\ \texttt{77 62 31 BC B4 4E C6 01} \\ \texttt{6F CD BC 98 10 78 6F EC} \end{pmatrix}$$

$$x_{10} = \begin{pmatrix} \texttt{B4 74 E1 56 96 31 B9 6C} \\ \texttt{21 A1 B6 33 CC 89 68 1A} \\ \texttt{B1 97 25 86 7B 2B 3F 09} \\ \texttt{4C 73 C7 62 93 A8 15 CF} \\ \texttt{55 15 C0 C0 9A 05 05 16} \\ \texttt{23 44 8D 8D D3 5F B3 6E} \\ \texttt{7E 6C 2D 37 12 D0 F3 3E} \\ \texttt{CE B8 04 F2 8D 9F C9 99} \end{pmatrix} \quad x_{12} = \begin{pmatrix} \texttt{3F 72 C2 60 EE 28 EF EA} \\ \texttt{42 8E B5 3A FB 8A 33 A2} \\ \texttt{03 E4 72 31 90 A5 1A D3} \\ \texttt{3E 68 E6 46 FC 94 3C C7} \\ \texttt{80 42 9E 2E CB 32 75 93} \\ \texttt{30 AA E2 21 21 C8 99 ED} \\ \texttt{86 1E 06 9E 91 1F 89 6C} \\ \texttt{D2 99 EC 7E E9 0B 01 10} \end{pmatrix}$$

The last entry corresponds to the final output $\pi(x_0) = x_{12}$.

1.3.2 Authenticated Encryption

Inputs are plain ASCII.

K	= `"192-bit Secret Key value"`	(24 Bytes)
N	= `"Nonces Used Once"`	(16 Bytes)
A	= `"AAD Test Vector Exact Block 32 B"`	(32 Bytes)
P	= `"2 Block Test Vector for stribob192r2d2"`	(38 Bytes)

Trace of inputs and outputs for the π permutation:

Block 0	31 39 32 2D 62 69 74 20 53 65 63 72 65 74 20 4B
	65 79 20 76 61 6C 75 65 01 00 00 00 00 00 00 00
	12 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
π out 0	FA A0 45 77 16 DB 55 A3 82 13 BD E2 44 EE D7 8C
	A1 57 5E 12 8D 0D 0D FB 5D FF 8C 67 D4 EA 6B 31
	8E 52 09 AD 17 8F 72 A4 2A D8 39 CC B9 88 40 A9
	00 8B B6 D9 FB 76 EB B1 A5 FC F4 01 AF F2 2E 27
Block 1	B4 CF 2B 14 73 A8 75 F6 F1 76 D9 C2 0B 80 B4 E9
	A0 57 5E 12 8D 0D 0D FB 5D FF 8C 67 D4 EA 6B 31
	AC 52 09 AD 17 8F 72 A4 2A D8 39 CC B9 88 40 A9
	00 8B B6 D9 FB 76 EB B1 A5 FC F4 01 AF F2 2E 27
π out 1	30 75 A4 58 67 4D 4C F0 3B AE AE 84 1A 5B 3C 06
	84 18 18 70 C5 3D E7 A6 CE 04 63 60 33 5D 5E 96
	2F E5 66 5B B4 9A AF CE B2 1B 45 3D 3E 1B FC 13
	F5 A7 02 1F 31 CC D6 5F 74 C5 30 F9 6A BD A1 2E
Block 2	71 34 E0 78 33 28 3F 84 1B F8 CB E7 6E 34 4E 26
	C1 60 79 13 B1 1D A5 CA A1 67 08 40 00 6F 7E D4
	6C E5 66 5B B4 9A AF CE B2 1B 45 3D 3E 1B FC 13
	F5 A7 02 1F 31 CC D6 5F 74 C5 30 F9 6A BD A1 2E
π out 2	6B BC 1D 05 10 75 5B 27 E0 B0 21 44 04 5A 4E 18
	7E FC 3C 6C 05 76 3D 8C 99 59 A7 27 DF 6A 03 1D
	40 3D 29 10 8D 01 56 49 DB EC 58 D9 B0 62 84 98
	19 73 3F 3F 18 EC E5 49 89 8E 05 18 9F 32 C2 4F
Block 3	59 9C 5F 69 7F 16 30 07 B4 D5 52 30 24 0C 2B 7B
	0A 93 4E 4C 63 19 4F AC EA 2D D5 4E BD 05 61 2C
	10 3D 29 10 8D 01 56 49 DB EC 58 D9 B0 62 84 98
	19 73 3F 3F 18 EC E5 49 89 8E 05 18 9F 32 C2 4F
π out 3	20 A0 35 CE C5 A5 DE 90 C6 D8 79 BA D4 B4 2D F5
	25 C7 0C D1 CB BF 86 EB B8 3F E5 23 D2 4E D1 60
	BD 76 13 59 E1 CD A9 98 97 A0 6F 34 D9 03 8D 43
	CE 95 12 87 F1 F6 A4 58 DA DD 6E CA F0 C9 4F 8F
Block 4	19 92 47 FC A1 97 DF 90 C6 D8 79 BA D4 B4 2D F5
	25 C7 0C D1 CB BF 86 EB B8 3F E5 23 D2 4E D1 60
	EF 76 13 59 E1 CD A9 98 97 A0 6F 34 D9 03 8D 43
	CE 95 12 87 F1 F6 A4 58 DA DD 6E CA F0 C9 4F 8F
π out 4	AE AE 71 0F 0D ED 3E 56 5B D0 26 FE 20 F6 4A 4F
	12 81 B7 3C BB 63 65 62 A5 D9 8B E1 72 EE A0 8A
	78 C2 9E 84 2C 91 BF 5D 39 1C 9A 5C 2B 34 7E 1B
	24 DC 45 DC 71 EB 2D 04 C2 EE AE 39 17 0C 60 20

Authenticated ciphertext has 38 message bytes + 16 for MAC = 54 (0x36) bytes:

Offset	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	59	9C	5F	69	7F	16	30	07	B4	D5	52	30	24	0C	2B	7B
1x	0A	93	4E	4C	63	19	4F	AC	EA	2D	D5	4E	BD	05	61	2C
2x	19	92	47	FC	A1	97	AE	AE	71	0F	0D	ED	3E	56	5B	D0
3x	26	FE	20	F6	4A	4F										

Chapter 2

Security Goals

2.1 Specific Goals

STRIBOBr2 security parameters, claims, and goals remain unmodified from the original submission:

Category	Effort	Attack Goal
Confidentiality for the plaintext.	2^{191}	Recover plaintext from ciphertext.
Integrity for the plaintext.	2^{127}	Forge valid ciphertext.
Integrity for the associated data.	2^{127}	Forge Associated Data.
Integrity for the public message number.	2^{127}	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 / nonce pair. The "unit" for the effort is equivalent to the effort required to compute the π permutation.

2.2 Nonce Re-Use

STRIBOBr2 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**. STRIBOBr2 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 [51] and Duplex Sponges in particular – proofs in [9, 12] are applicable. For the primitives of Section 1.2.2:

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 $\mathsf{decrypt}(K,\ N,\ A,\ C) =$ 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

3.1 Attacks on the Sponge AEAD Mode BLNK

STRIBOBr1 [55, 56, 57] and STRIBOBr2 [58] use exactly the same padding mechanism, BLNK. This padding mechanism is a variant of Saarinen's Blinker [53] padding, but limited to the CAESAR use case.

Some independent analysis of the mode has been recently published [32, 60], essentially validating the claimed security bounds. No attacks have been reported against the padding mechanism, and hence the security bounds derived from DuplexWrap [9, 14] directly apply.

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 [13] 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. BLNK uses simpler embedded encoding. However, for our "effective capacity" c [53] the following Theorem holds:

Theorem 1 (Derived from Theorem 4 from [14])**.** *The DuplexWrap and BLNK authenticated encryption modes satisfy the following privacy and authentication security bounds:*

$$\begin{aligned}\mathrm{Adv}^{\mathsf{priv}}_{\mathsf{sbob}}(\mathcal{A}) &< \frac{D+T}{2^k} + \frac{D^2+4DT}{2^{c+1}}\\ \mathrm{Adv}^{\mathsf{auth}}_{\mathsf{sbob}}(\mathcal{A}) &< \frac{D+T}{2^k} + \frac{D^2+4DT}{2^{c+1}} + \frac{D}{2^t}.\end{aligned}$$

against any single adversary $\mathcal{A}$ if $K \xleftarrow{\$} \{0,1\}^k$, tags of t bits are used, π is a randomly chosen permutation, D is the data complexity (number of queries to target), and T is the offline attack time complexity.

Proof. See Theorem 4 of [14] and related work [3, 9]. See also [32]. □

Since $b = 512$, we choose a Sponge rate of $r = 256$ bits, which leaves capacity $c = b - r = 256$. We choose key size $k = 192$ and limit $D < 2^{56}$ (2^{64} bits) and $T < 2^{k-1}$. As our actual effective capacity is $c \approx 254$ ($\delta \leq 2$ effective capacity bits are lost due to domain separation bits [53]), a 192-bit security level is reached.

3.2 The π Permutation

The STRIBOBr2 π permutation is lifted from Whirlpool 3.0 hash function [4] in straightforward fashion, and a security relationship exists between the two; a structural distinguisher against π would also indicate a weakness in Whirlpool. Whirlpool is the only hash funcion (in addition to SHA [44]) in the NESSIE final portfolio [42]. Whirlpool is also an ISO standard [30]. The Whirlpool compression function is among the best understood, and most closely scrutinized cryptographic primitives in existence.

The π permutation from Whirlpool was designed using the wide trail design strategy [23]. LPS gets all of its non-linearity from the 8-bit S-Box S, which has been designed to offer resistance against classical methods of cryptanalysis. Its differential bound [15] is $P = \frac{8}{256}$ and best linear approximation [38] holds with $P = \frac{28}{128}$. No algebraic weaknesses have been discovered.

Many structural observations on AES-like ciphers also apply to LPS [5]. Adjusted to its state size, LPS has similar per-round avalanche to AES (each input byte affects each output byte after two rounds) and similar resistance to Square attacks. The best theoretical Square attack is effective against six rounds [35]. Due to shared design principles, vast AES research literature is also largely applicable to both Whirlpool and STRIBOBr2 [5, 24].

The only effective attack against 10-round variant of Whirlpool is the Rebound Attack [36, 37, 41]. We firmly believe that increase of rounds from 10 to 12 makes STRIBOBr2 resistant to these attacks. Even addition of a single round would increase the work factor of those attacks by a significant factor. Since we are using a sponge mode with $r = 256$ $c = 256$, an attacker only has control over half of the state, and therefore STRIBOBr1 and STRIBOBr2 designs have a good security margin against these attacks [40].

3.3 Side-Channel Attacks

Due to the minibox structure, we may load the 4×4 - bit tables in registers and access them via constant-time shuffles on Intel SSSE3 and ARM NEON SIMD targets as noted in Section 4.2.2. STRIBOBr2 is also suited for bitsliced implementation due to its particular S-Box and MDS design.

Being unconditional straight-line code without data-dependent table lookups, bitsliced and byte shuffling implementations are effective countermeasures against cache timing attacks, which can be mounted against cryptographic primitives with large tables such as AES [1, 6, 47, 63].

A non-constant-time implementation of the S-Box on Whirlpool, Streebog, or StriBob on 64-bit platforms typically requires lookup tables of up to $8 \times 256 \times 8 = 16384$B. Even though this size easily fits into the Level 2 cache of any 64-bit system, one may see that timing attacks are possible as L2 caches are not always shared even between different execution cores within a single CPU unit. This is due to the process switching operation of most 64-bit operating systems.

3.4 Conclusions

Structural similarity and equivalent differential and linear properties indicate that the security levels of STRIBOBr1 and STRIBOBr2 are essentially the same. However, the Whirlpool structure has received even more of cryptanalysis [34, 59] than Streebog [26]. STRIBOBr2 has more implementation options, facilitating constant-time operation. To summarize main security arguments for STRIBOBr2:

- **Well-understood Structure.** STRIBOBr2 is a classical sponge-based design; a large body of literature and theorems exist to justify its particular security claims.
- **Recycling strong components.** The π permutation from Whirlpool has also received a large amount of public analysis, and its shared design principles with AES increase our confidence in it.
- **Security Reduction.** It is relatively straightforward to show that a *structural distinguisher* against STRIBOBr2 is also a structural distinguisher against a 12-round extended variant of Whirlpool.
- **Fast, Constant-time software implementation.** STRIBOBr2 allows very efficient constant-time implementation on many platforms, making it resistant to side-channel attacks (that make secure software implementation even of AES more difficult.) Hardware implementations have also been shown to be efficient.

Extensive, directly applicable research into the security of Sponge functions, DuplexWrap, Whirlpool, Streebog, and AES-like structures warrants an exceptional level of confidence for the long-term security of both STRIBOBr1 and STRIBOBr2. Based on our review of latest research, no attacks are known against corresponding versions of these ciphers, and a comfortable security margin remains.

Chapter 4

Features

4.1 Advantages over AES-GCM

The main technical advantages of STRIBOBr2 over AES-GCM are:

- Unlike AES-GCM, the STRIBOBr2 authentication tags offer a level of integrity protection commensurate with its length, rather than half of it [52].
- STRIBOBr2 allows easier constant-time implementation on targets without special AES instructions. Software speeds are often faster than AES-192.
- The same hybrid STRIBOBr2/Whirlpool core can also be used for unkeyed hashing in digital signatures (standards-compliant certificate handling) and other applications.

STRIBOBr2 has superb implementation characteristics on FPGA, SIMD, and lightweight targets.

4.2 Software Implementations

STRIBOBr2 has extremely small implementation footprint on resource-limited software platforms – typically under half a kilobyte. Its particular S-Box and MDS design allows STRIBOBr2 to have efficient constant-time bitsliced and SIMD byte shuffling implementations. These are an effective countermeasure against cache timing attacks, which are a concern against AES. The $b = 8 \times 64$ - bit state size is particularly suitable for bitslicing of a byte-oriented algorithm on 64-bit platforms and byte slicing for SIMD platforms.

4.2.1 Lightweight Targets

The entire byte-oriented implementation of π fits onto a single page; See Section 1.1.5. Remarkably, in addition to π, only the S-Box `wbob_sbox[256]` (Table 1.1) together with minimal BLNK logic are required for full AEAD implementation. On many microcontrollers targets, STRIBOBr2's entire software footprint is in the 500B range. Slightly more is required for a shared secret handshake protocol and two-way secure BLINKER protocol [53].

This is a significant improvement over STRIBOBr1, which typically needs almost 2kB. STRIBOBr1 is also much slower and larger on low-end microcontrollers due to the "heavy" MDS matrix. The reference implementation is written for compactness and clarity; it is not optimal when it comes to speed or size. We refer to section 7.3 of [4] for techniques that greatly reduce the number of XORs required, resulting in increased processing speed. Additional tables will be required, however, and this will increase the overall implementation size.

4.2.2 Constant-Time SIMD Implementation

Due largely to Whirlpool's S-Box structure and generous parallelism, STRIBOBr2's π is well suited for high speed, constant-time implementation on Single Instruction Multiple Data (SIMD) architectures.

Here we focus on ARM's NEON as the reference architecture since the state layout fits the registers nicely, but also consider Intel's SSSE3 as another explicit example. The goal is to improve performance, while at the same time avoiding memory-resident table lookups that cause execution time to depend on the data cache state and thus algorithm state (the crux of cache timing attacks).

Related work in this area includes simulated ISA extensions to a RISC architecture for parallel table lookups to speed up Whirlpool [28]. These extensions are then used to build essentially a hardware-assisted analogue of the traditional T tables software implementation – storing the state in rows and issuing a single instruction to perform 8 parallel lookups from the 8-bit S-Box input to the 64-bit linear layer output and XOR-summing the results, repeated for each row. AES [27] and Anubis [20] can also take advantage of SSSE3's variable byte shuffle instruction for fast and secure implementations.

NEON has 32×64 - bit SIMD registers and SSSE3 16×128 - bit. We store the state column-wise (one column per NEON register, two columns per SSSE3 register), i.e. byte position j of register i contains the state byte in column i and row j. The `SubBytes` step is not sensitive to this ordering, but both `ShiftColumns` and `MixRows` are. Since both of these architectures feature variable byte shuffle instructions (`vtbl.u8` for NEON and `pshufb` for SSSE3), implementing `SubBytes` is a direct translation of Figure 1.1 to these instructions. This amounts to 40 NEON shuffles and half as many SSSE3 shuffles. For `ShiftColumns`, NEON uses `vext` for byte-wise register rotation and SSSE3 `pshufb` with constant rotation distances since each register holds two columns. For `MixRows`, we use the row formula from the Whirlpool specification [4, Sec. 7.3] where the multiplications by x are a simple left shift (native on NEON, integer addition on SSSE3) and conditional XOR (operand masked by signed right shift on NEON, comparison on SSSE3). The formula is fairly symmetric around even and odd byte positions – while NEON implements it as written with 24 multiplications, SSSE3 slightly rearranges a few registers to parallelize across the full 128-bit register width and use half as many multiplications.

4.2.3 Generic Constant-Time Bitsliced Implementation

The byte-oriented $8 \times 64 = 512$ - bit state can be rapidly split into eight 64-bit registers. The parallelism evident in Figure 1.1 helps to speed up bitsliced implementation. We see that for 2/3 of the time, the S-Box has effectively two independent 4-bit execution paths. Interleaving these may greatly reduce wait states due to the superscalar architecture employed by most modern CPUs.

Appendix B of the current 2003 Whirlpool specification [4] gives listings with 14-16 instructions/gates for each of the miniboxes (if ANDN instruction is allowed). Those were used in our reference bitsliced implementations.

4.3 Implementation Summary

We currently have six implementations of STRIBOBr2. They mainly differ in the implementation technique used for the π cryptographic permutation.

- **C 8-bit**: This is the minimal reference implementation which is optimized for clarity and low-resource platforms, corresponding to Section 1.1.5.
- **C 64-bit**: Standard speed-optimized implementation for most platforms, utilizing large lookup tables. Apart from Whirlpool-derived tables, equivalent to the implementation of STRIBOBr1 [56].
- **C Bitsliced**: Straight-line, fully bitsliced implementation without data-dependent branches or lookups. Resistant to timing attacks.
- **NEON Intrinsics**: Fast constant-time version that avoids table lookups by storing 4×4 - bit miniboxes in SIMD registers.
- **SSSE3 Intrinsics**: Similar but for 128-bit SIMD registers.
- **Verilog 12-cycle**: This is the hardware reference implementation. Source code is about 350 lines. Additional logic is required for AXI Bus integration.

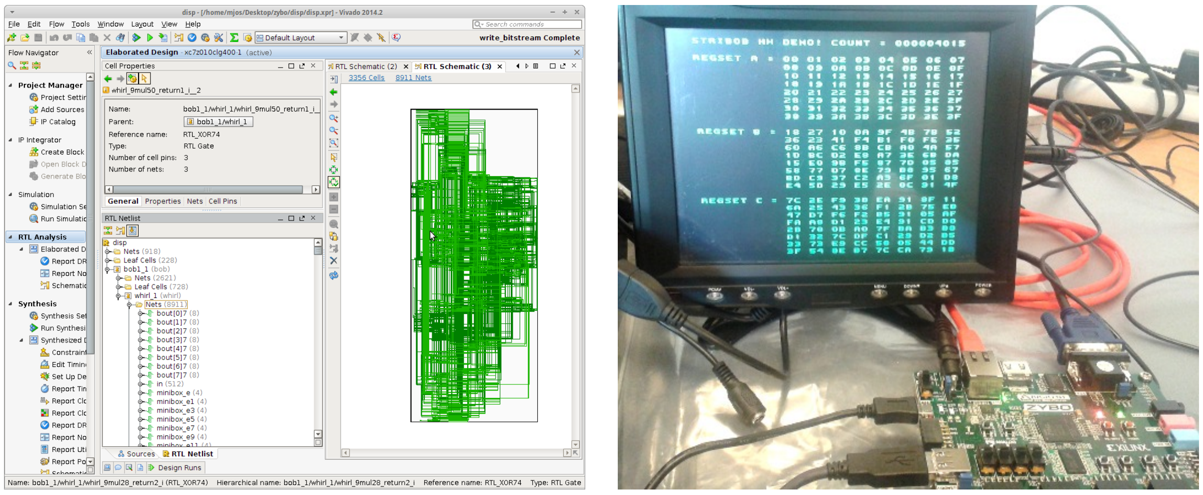

Figure 4.1: STRIBOBr2 was implemented on the FPGA logic fabric of Xilix Zynq 7010. The implementation integrates with the AXI bus of ARM Cortex A9 on the SoC chip.

4.3.1 Software Implementations

The first three implementations use only C99, and are hence easily portable. The 64-bit reference implementation is almost exactly as fast as OpenSSL's Whirlpool on the same platform. See Table 4.1 for implementation metrics. Note that STRIBOBr2 is faster then the (out-of-box, Ubuntu 14.04 LTS) OpenSSL implementation of AES-192 on the same target. We also have various embedded implementations.

Table 4.1: Comparing software implementations of STRIBOBr2's π.

Target	**Speed**	**Footprint**		**Source**
	MB/sec	Code	Data	C lines
Single Core of 3.4GHz Core i7-4770				
8-bit C99 Reference	7.772	326	256	97
Bitsliced C99 Reference	49.02	4592	768	345
64-bit C99 Reference	139.2	1942	16512	128
SSSE3 (Constant-Time)	**162.3**	1290	1152	256
OpenSSL 1.0.1f AES-192 CBC	*145.6*			
BeagleBone Black 1.0GHz Cortex-A8				
8-bit C99 Reference	0.828	352	256	97
64-bit C99 Reference	3.343	6524	16512	128
Bitsliced C99 Reference	1.435	15704	768	345
NEON (Constant-Time)	9.208	1528	1072	320

4.3.2 Hardware Implementation

The hardware implementation has been proven on FPGA (Figure 4.1). The SÆHI proposal reports total post place-and-route utilization on Artix-7 of 4,946 logic units for a single round of STRIBOBr2, which compares favorably to 7,972 required for Keccak/Keyak [54]. Throughput is roughly 2 MB/s for each MHz.

Chapter 5

Design Rationale

STRIBOBr2 design and analysis were interdependent and were performed concurrently. Design rationale is therefore complemented by Chapter 3, "Security Analysis". Here we just give some broad rationale to our overall component and parameter selection.

5.1 Block Cipher Modes are Limited by the Birthday Paradox

Authenticated Encryption modes based on block ciphers such as AES Rijndael [24] were deemed unsatisfactory as the security of most of these is bound by the limit imposed by 128-bit block size and the birthday paradox; this applies to GCM [43], OCB [50, 51], and similar synthesized modes [29]. Furthermore, we wanted to explore a different design paradigm.

The Sponge Construct is a flexible method for building cryptographic algorithms of different kinds from a single permutation. The Sponge parameter selections were derived from well-established theorems regarding Sponge functions and a large body of research. [7, 8, 11, 9, 10, 13].

A random-indistinguishable π and appropriate padding rules are sufficient to construct Sponge-based hashes [7], Tree Hashes [13], MACs [10], Authenticated Encryption (AE) algorithms [9, 14], and pseudorandom extractors (SHAKEs, PRFs, and PRNGs) [8, 45].

As a starting point, we wanted to marry Blinker-style Sponge padding [53] with a well-understood, secure cryptographic permutation. The permutation should be large enough to allow both, efficient operation and sufficient security level.

5.2 Implementation Synergy with Standard Hash Functions

It was observed that some dedicated hash function standards, the Russian standard hash function GOST R 34.11-2012 [26] and the (ISO Standard and NESSIE final portfolio algorithm) Whirlpool hash function [4] contain within them 512-bit keyless permutations. We took those permutations and used them in a Sponge-based design. The shared S-Boxes and codebase with corresponding hashes would be helpful in embedded and lightweight applications – there would be implementation synergy.

Whirlpool has received a significant amount of analysis in the almost 15 years since its original publication. Whirlpool was the only hash function in the final NESSIE portfolio in addition to SHA-2 hashes [42]. Whirlpool has also been standardized by ISO as part of ISO/IEC 10118-3:2004 [30].

The amended MDS matrix used by current ('03) Whirlpool is also used by STRIBOBr2 as a countermeasure to the structural observations given in [61]. Our design is based on Whirlpool 3.0.

Whirlpool was found to be vulnerable to a Rebound Distinguisher [36, 37, 41]. That 2^{188} attack applies to the 10-round variant; our 12-round version should offer a comfortable security margin, especially as our security target is 2^{192}. The way the round constants are derived from the S-Box allows this change to be made in a straightforward manner.

After careful analysis, we concluded that the π permutations obtained contain no structural distinguishers that are not based on some trivial property such as a priori knowledge of output value of $\pi(x)$ for some particular x.

5.3 Hidden Weaknesses

The original design criteria for Streebog (on which STRIBOBr1 is partially based) has never been published, so various teams have attempted to determine those. Even though some of the "mysterious constants" of Streebog have been recently revealed [18, 33, 49] in response to potential attacks [2], the rationale behind many of the design choices still remains hidden. STRIBOBr2 has no such magical constants; the entire design process is open and repeatable. See [4] for S-Box and MDS details for STRIBOBr2.

The designers of STRIBOBr2 have not hidden any weaknesses in this cipher.

5.4 Updated Information on STRIBOBr1 π Design

The 8-bit S-Box used by STRIBOBr1 was directly lifted from Streebog so that hardware and software components developed for Streebog could be shared or recycled when implementing STRIBOBr1. The same S-Box is also used by the recently proposed Russian Encryption Standard "Kuznyechik" [25, 62].

The GOST R 34.11-2012 "Streebog" standard text [26] does not describe the linear step as a 8×8 matrix-vector multiplication with $\mathsf{GF}(2^8)$ elements like the STRIBOBr1 spec [56], but as a 64×64 binary matrix multiplication. One can see that $8 \times 8 \times 8 = 512$ bits are required to describe the former, but $64 \times 64 = 4096$ bits are required for the latter. The more effective description was discovered by Kazymorov and Kazymorova in [33] by exhaustively testing all 30 irreducible polynomial bases, revealing an AES-like MDS structure. The origin of the particular numerical values of that MDS matrix is still a mystery. They do not appear to offer similar avenues for size or performance optimization like those in Whirlpool 3.0 and STRIBOBr2 do.

Not much about the particular design criteria of the Streebog S-Box has been published. That S-Box was apparently selected at least 5 years ago as Streebog already appeared in RusCrypto '10 proceedings [39]. Very recent ongoing work has revealed it to also have an optimized representation [18], after all.

We can easily observe that the S-Box offers reasonable resistance against basic methods of cryptanalysis. Its differential bound [15] is $P = \frac{8}{256}$ and best linear approximation [38] holds with $P = \frac{28}{128}$. There does not seem to be any exploitable algebraic weaknesses. These are the exact same bounds as can be found for the Whirlpool and STRIBOBr2 S-Box, but fall short from the bounds of the AES S-Box.

By comparison, the Rijndael AES S-Box is constructed from finite field inversion x^{-1} operation in $\mathsf{GF}(2^8)$ (inspired by the Nyberg construction [46]) and an affine bit transform that serves as a countermeasure against, among other things, Interpolation Attacks [31] on the AES predecessor SHARK [48]. We refer to [24] for more information about the AES design process.

We had brief informal discussions with some members of the Streebog and Kuznyechik design team at the CTCrypt '14 workshop (05-06 June 2014, Moscow RU). Their recollection was that the aim was to choose a "randomized" S-Box that meets the basic differential, linear, and algebraic requirements. Randomization using various building blocks was simply iterated until a "good enough" permutation was found. This was seen as an effective countermeasure against yet-unknown attacks. At the time of Streebog S-Box selection (before 2010's) the emergence of allegedly effective AES Algebraic Attacks such as [22] was a major concern for much of the symmetric cryptographic community. Hence it was felt appropriate to avoid too much algebraic structure in either the S-Box or MDS matrix while also ensuring necessary resistance against known attacks such as DC and LC. Algebraic attack attempts of this type against AES have since largely fizzled out. We feel confident that the Whirlpool S-Box should be sufficient for our claimed security level, especially as it offers significantly better speeds in constant-time implementations when compared to an AES-Style S-Box.

One is left with the impression that Streebog is a "whitened" or randomized copy of the original Whirlpool design. Despite its partially unknown origins and relative shortcomings on some implementation targets, we consider STRIBOBr1 to be at least as secure as STRIBOBr2 if appropriately implemented. Indeed some of the more successful attacks on AES and Whirlpool have been based on their deep structural self-similarities and simplistic key schedules [16, 17, 19], so STRIBOBr1 may have some security advantages against "unknown" attacks.

Chapter 6

Changes

In order to facilitate better implementation techniques, we have produced a new variant of the cipher with STRIBOBr2. We still encourage security evaluation of STRIBOBr1.

The changes for STRIBOBr2 have not been done in response to any specific cryptanalytic attacks; we are confident that both STRIBOBr1 and STRIBOBr2 offer essentially equivalent security.

The new variant, STRIBOBr2, is structurally identical to the original STRIBOBr1 design, but differs in its S-Box and linear mixing selection. In fact the source code for the 64-bit reference implementations of the two ciphers are nearly identical, differing only in tables.

- Section 1.1.1: The 8×8 - bit S-Box from [26, 62] has been replaced with that from Whirlpool 3.0 [4].
- Section 1.1.2: The byte transpose shuffle of [26] has been replaced with `ShiftColumns` shuffle of [4].
- Section 1.1.3: The 8×8 - byte matrix and finite field from [33] has been replaced with that from [4].
- Section 1.1.4: New Round constants C. Rather than using the Magical constants from [26, 49], we are using ones derived from the S-Box, as in [4].

We note that these Whirlpool components were revised several times during the NESSIE evaluation process; its MDS matrix [61] and S-Box were both redesigned. We are using the final 3.0 specifications, with the difference that number of rounds has been increased from 10 to 12 in response to Rebound Distinguishers [36, 37, 41]. That 2^{188} attack applies to the 10-round variant; our 12-round version should offer a comfortable security margin, especially as our security target is 2^{192} [40]. STRIBOBr1 also has 12 rounds.

Chapter 7

Intellectual Property and Consent

7.1 Intellectual Property

The authors are not aware of any patents or patent applications that directly cover the work described in this document, nor are planning to submit any.

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

7.2 Consent to CAESAR Selection Committee

The submitters hereby consent 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 submitters understand 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 submitters understand 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 submitters acknowledge that the committee decisions reflect the collective expert judgments of the committee members and are not subject to appeal.

The submitters understand 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 submitters understand that this statement is required as a condition of consideration of this submission by the CAESAR selection committee.

Submitters:

Dr. Markku-Juhani O. Saarinen	**Dr. Billy Bob Brumley**

Bibliography

[1] Onur Aciiçmez, Werner Schindler, and Çetin Kaya Koç. Cache based remote timing attack on the AES. In Masayuki Abe, editor, *CT-RSA 2007*, volume 4377 of *LNCS*, pages 271–286. Springer, 2007. doi:10.1007/11967668_18.

[2] Riham AlTawy and Amr M. Youssef. Watch your constants: Malicious streebog. IACR ePrint 2014/879, 2014. URL: https://eprint.iacr.org/2014/879.

[3] Elena Andreeva, Bart Mennink, and Bart Preneel. Security reductions of the second round SHA-3 candidates. IACR ePrint 2010/381, July 2010. URL: https://eprint.iacr.org/2010/381.

[4] Paulo S. L. M. Barreto and Vincent Rijmen. The Whirlpool hashing function. NESSIE Algorithm Specification, 2000, Revised May 2003. URL: http://www.larc.usp.br/~pbarreto/WhirlpoolPage.html.

[5] Christof Beierle, Philipp Jovanovic, Martin M. Lauridsen, Gregor Leander, and Christian Rechberger. Analyzing permutations for AES-like ciphers: Understanding ShiftRows. In Kaisa Nyberg, editor, *CT-RSA 2015*, volume 9048 of *LNCS*, pages 37–58. Springer, 2015. doi:10.1007/978-3-319-16715-2_3.

[6] D. J. Bernstein. Cache-timing attacks on AES. Technical report, University of Chigaco, 2005. URL: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[7] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge functions. In *Ecrypt Hash Workshop 2007*, May 2007. URL: http://events.iaik.tugraz.at/HashWorkshop07/program.html.

[8] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge-based pseudo-random number generators. In Stefan Mangard and François-Xavier Standaert, editors, *CHES 2010*, volume 6225 of *LNCS*, pages 33–47. Springer, 2010. doi:10.1007/978-3-642-15031-9_3.

[9] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Duplexing the sponge: Single-pass authenticated encryption and other applications. In A. Miri and S. Vaudenay, editors, *SAC 2011*, volume 7118 of *LNCS*, pages 320–337. Springer, 2011. doi:10.1007/978-3-642-28496-0_19.

[10] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The Keccak reference, version 3.0. NIST SHA3 Submission Document, January 2011. URL: http://keccak.noekeon.org/Keccak-reference-3.0.pdf.

[11] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. On the security of the keyed sponge construction. In *SKEW 2011 Symmetric Key Encryption Workshop*, February 2011. URL: http://skew2011.mat.dtu.dk/proceedings/On%20the%20security%20of%20the%20keyed%20sponge%20construction.pdf.

[12] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Permutation-based encryption, authentication and authenticated encryption. In *DIAC 2012*, 2012. URL: http://keccak.noekeon.org/KeccakDIAC2012.pdf.

[13] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sakura: a flexible coding for tree hashing. IACR ePrint 2013/231, April 2013. URL: https://eprint.iacr.org/2013/231.

[14] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. CAESAR submission: Keyak v1. CAESAR First Round Submission, March 2014. URL: `http://competitions.cr.yp.to/round1/keyakv1.pdf`.

[15] Eli Biham and Adi Shamir. *Differential Cryptanalysis of the Data Encryption Standard*. Springer, 1993. `doi:10.1007/978-1-4613-9314-6`.

[16] Alex Biryukov and Dmitry Khovratovich. Related-key cryptanalysis of the full AES-192 and AES-256. In Mitsuru Matsui, editor, *ASIACRYPT '09*, volume 5912 of *LNCS*, pages 1–18. Springer, 2009. `doi:10.1007/978-3-642-10366-7_1`.

[17] Alex Biryukov, Dmitry Khovratovich, and Ivica Nikolić. Distinguisher and related-key attack on the full AES-256. In Sjai Halevi, editor, *CRYPTO '09*, volume 5677 of *LNCS*, pages 231–249. Springer, 2009. `doi:10.1007/978-3-642-03356-8_14`.

[18] Alex Biryukov, Lèo Perrin, and Aleksei Udovenko. The secret structure of the S-Box of Streebog, Kuznechik and StriBob. IACR ePrint 2015/812, August 2015. URL: `https://eprint.iacr.org/2015/812`.

[19] Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger. Biclique cryptanalysis of the full AES. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT '11*, volume 7073 of *LNCS*, pages 344–371. Springer, 2011. `doi:10.1007/978-3-642-25385-0_19`.

[20] Billy B. Brumley. Secure and fast implementations of two involution ciphers. In T. Aura, K. Järvinen, and K. Nyberg, editors, *NordSec '10*, volume 7127 of *LNCS*, pages 269–282. Springer, 2012. `doi:10.1007/978-3-642-27937-9_19`.

[21] CAESAR. CAESAR call for submissions, January 2014. URL: `http://competitions.cr.yp.to/caesar-call.html`.

[22] N. Courtois. How fast can be algebraic attacks on block ciphers? IACR ePrint 2006/168, May 2006. URL: `https://eprint.iacr.org/2006/168`.

[23] Joan Daemen and Vincent Rijmen. The wide trail design strategy. In Bahram Honary, editor, *Cryptography and Coding 2001*, volume 2260 of *LNCS*, pages 222–238. Springer, 2001. `doi:10.1007/3-540-45325-3_20`.

[24] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - the Advanced Encryption Standard*. Springer, 2002. `doi:10.1007/978-3-662-04722-4`.

[25] Denis M. Dygin, Ivan V. Lavrikov, Grigory B. Marshalko, Vladimir I. Rudskoy, Dmitry I. Trifonov, and Vasily A. Shishkin. On a new Russian Encryption Standard. *Mathematical Aspects of Cryptography*, 6(2):29–34, 2015. (Abstract In Russian). URL: `http://www.mathnet.ru/php/archive.phtml?wshow=paper&jrnid=mvk&paperid=142&option_lang=eng`.

[26] GOST. Information technology. cryptographic protection of information, hash function. GOST R 34.11-2012, 2012. (In Russian). URL: `http://protect.gost.ru/v.aspx?control=7&id=180209`.

[27] M. Hamburg. Accelerating AES with vector permute instructions. In C. Clavier and K. Gaj, editors, *CHES '09*, volume 5747 of *LNCS*, pages 18–32. Springer, 2009. `doi:10.1007/978-3-642-04138-9_2`.

[28] Y. Hilewitz, Y. L. Yin, and R. B. Lee. Accelerating the Whirlpool hash function using parallel table lookup and fast cyclical permutation. In K. Nyberg, editor, *FSE '08*, volume 5086 of *LNCS*, pages 173–188. Springer, 2008. `doi:10.1007/978-3-540-71039-4_11`.

[29] Viet Tung Hoang, Jonathan Katz, and Alex J. Malozemoff. Automated analysis and synthesis of authenticated encryption schemes. IACR ePrint 2015/624, June 2015. URL: `https://eprint.iacr.org/2015/624`.

[30] ISO/IEC. Information technology - security techniques - hash-functions - part 3: Dedicated hash-functions. ISO/IEC 10118-3:2004, 2004. URL: `https://www.iso.org/obp/ui/#iso:std:iso-iec:10118:-3:ed-3:v1:en`.

[31] T. Jakobsen and Lars R. Knudsen. The interpolation attack on block cipers. In Eli Biham, editor, *FSE '97*, volume 1267 of *LNCS*, pages 99–112. Springer, 1996.

[32] Philipp Jovanovic, Atul Luykx, and Bart Mennink. Beyond $2^{c/2}$ security in sponge-based authenticated encryption modes. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014*, volume 8873 of *LNCS*, pages 85–104. Springer, 2014. doi:10.1007/978-3-662-45611-8_5.

[33] Oleksandr Kazymyrov and Valentyna Kazymyrova. Algebraic aspects of the Russian hash standard GOST R 34.11-2012. In *CTCrypt '13, June 23-24, 2013, Ekaterinburg, Russia*, 2013. IACR ePrint 2013/556. URL: https://eprint.iacr.org/2013/556.

[34] Lars R. Knudsen. Non-random properties of reduced-round Whirlpool. NESSIE public report, NES/DOC/UIB/WP5/016/2, August 2002. URL: https://www.cosic.esat.kuleuven.be/nessie/reports/phase2/uibwp5-016-2.pdf.

[35] Lars R. Knudsen and David Wagner. Integral cryptanalysis (extended abstract). In Joan Daemen and Vincent Rijmen, editors, *FSE 2002*, volume 2365 of *LNCS*, pages 112–127. Springer, 2002. doi:10.1007/3-540-45661-9_9.

[36] Mario Lamberger, Florian Mendel, Christian Rechberger, Vincent Rijmen, and Martin Schläffer. Rebound distinguishers: Results on the full whirlpool compression function. In Mitsuru Matsui, editor, *ASIACRYPT '09*, volume 5912 of *LNCS*, pages 126–143. Springer, 2009. doi:10.1007/978-3-642-10366-7_8.

[37] Mario Lamberger, Florian Mendel, Martin Schläffer, Christian Rechberger, and Vincent Rijmen. The rebound attack and subspace distinguishers: Application to Whirlpool. *J. Cryptology*, 28:257–296, 2015. doi:10.1007/s00145-013-9166-5.

[38] Mitsuru Matsui. Linear cryptoanalysis method for DES cipher. In T. Helleseth, editor, *EUROCRYPT '93*, volume 765 of *LNCS*, pages 386–397. Springer, 1994. doi:10.1007/3-540-48285-7_33.

[39] D. V. Matyuhin, V. I. Rudskoy, and V. A. Shishkin. Promising hashing algorithm. RusCrypto '10 Workshop, 02 April 2010, 2010. (In Russian).

[40] Florian Mendel. Personal Communication, June 2014.

[41] Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. The rebound attack: Cryptanalysis of reduced Whirlpool and Grøstl. In Orr Dunkelman, editor, *FSE 2009*, volume 5665 of *LNCS*, pages 260–276. Springer, 2009. doi:10.1007/978-3-642-03317-9_16.

[42] NESSIE. *Final report of European project number IST-1999-12324, named New European Schemes for Signatures, Integrity, and Encryption*. NESSIE, April 2004. URL: https://www.cosic.esat.kuleuven.be/nessie/Bookv015.pdf.

[43] NIST. Recommendation for block cipher modes of operation: Galois/counter mode (GCM) and GMAC. NIST Special Publication 800-38D, 2007. URL: http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf.

[44] NIST. Secure Hash Standard (SHS). Federal Information Processing Standards Publication FIPS 180-4, March 2012. URL: http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf.

[45] NIST. SHA-3 standard: Permutation-based hash and extendable-output functions. FIPS 202, August 2015. doi:10.6028/NIST.FIPS.202.

[46] Kaisa Nyberg. Differentially uniform mappings for cryptography. In Tor Helleseth, editor, *EUROCRYPT '93*, volume 765 of *LNCS*, pages 55–64. Springer, 1994. doi:10.1007/3-540-48285-7_6.

[47] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In David Pointcheval, editor, *CT-RSA 2006*, volume 3860 of *LNCS*, pages 1–20. Springer, 2006. doi:10.1007/11605805_1.

[48] Vincent Rijmen, Joan Daemen, Bart Preneel, Antoon Bosselaers, and Erik De Win. The cipher SHARK. In Dieter Gollmann, editor, *FSE '96*, volume 1039 of *LNCS*, pages 99–111. Springer, 1996. doi:10.1007/3-540-60865-6_47.

[49] Vladimir Rodskoy. Note on Streebog constants origin. Preprint, 2015. URL: http://tk26.ru/en/ISO_IEC/streebog/streebog_constants_eng.pdf.

[50] Phillip Rogaway, M. Bellare, J. Black, and T. Krovetz. OCB: A block-cipher mode of operation for efficient authenticated encryption. In M. K. Reiter and P. Samarati, editors, *CCS '01: Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 196–205. ACM, 2001. doi:10.1145/501983.502011.

[51] Phillip Rogaway, Mihir Bellare, and Jogn Black. OCB: A block-cipher mode of operation for efficient authenticated encryption. *ACM Transactions on Information and System Security (TISSEC)*, 6(3):365–403, August 2003. URL: http://web.cs.ucdavis.edu/~rogaway/papers/ocb-full.pdf, doi:10.1145/937527.937529.

[52] Markku-Juhani O. Saarinen. Cycling attacks on GCM, GHASH and other polynomial MACs and hashes. In Anne Canteaut, editor, *FSE 2012*, volume 7549 of *LNCS*, pages 216–225. Springer, 2012. doi:10.1007/978-3-642-34047-5_13.

[53] Markku-Juhani O. Saarinen. Beyond modes: Building a secure record protocol from a cryptographic sponge permutation. In J. Benaloh, editor, *CT-RSA 2014*, volume 8366 of *LNCS*, pages 270–285. Springer, 2014. doi:10.1007/978-3-319-04852-9_14.

[54] Markku-Juhani O. Saarinen. Simple AEAD hardware interface (SÆHI) in a SoC: Implementing an on-chip Keyak/WhirlBob coprocessor. In *TrustED '14 Proceedings of the 4th International Workshop on Trustworthy Embedded Device*, pages 51–56. ACM, 2014. doi:10.1145/2666141.2666144.

[55] Markku-Juhani O. Saarinen. StriBob: Authenticated encryption from GOST R 34.11-2012 LPS permutation. In *CTCrypt '14, 05-06 June 2014, Moscow, Russia. Preproceedings.*, pages 170–182, June 2014. URL: https://eprint.iacr.org/2014/271.

[56] Markku-Juhani O. Saarinen. The STRIBOBr1 authenticated encryption algorithm. CAESAR, 1st Round Candidate, March 2014. URL: http://www.stribob.com.

[57] Markku-Juhani O. Saarinen. StriBob: Authenticated encryption from GOST R 34.11-2012 LPS permutation. *Mathematical Aspects of Cryptography*, 6(2):67–78, 2015. (Abstract In Russian). URL: http://www.mathnet.ru/php/archive.phtml?wshow=paper&jrnid=mvk&paperid=146&option_lang=eng.

[58] Markku-Juhani O. Saarinen and Billy B. Brumley. Lighter, Faster, and Constant-Time: WHIRLBOB, the Whirlpool variant of STRIBOB. In *NORDSEC 2015*, LNCS. Springer, October 2015. To appear. Also IACR ePrint 2014/501. URL: https://eprint.iacr.org/2014/501.

[59] Yu Sasaki, Lei Wang, Shuang Wu, and Wenling Wu. Investigating fundamental security requirements on Whirlpool: Improved preimage and collision attacks. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 562–379. Springer, 2012. doi:10.1007/978-3-642-34961-4_34.

[60] Yu Sasaki and Kan Yasuda. How to incorporate associated data in sponge-based authenticated encryption. In Kaisa Nyberg, editor, *CT-RSA 2015*, volume 9048 of *LNCS*, pages 353–370. Springer, 2015. doi:10.1007/978-3-319-16715-2_19.

[61] T. Shirai and K. Shibutani. On the diffusion matrix employed in the Whirlpool hashing function. NESSIE Public Report, 2003. URL: http://www.cosic.esat.kuleuven.be/nessie/reports/phase2/whirlpool-20030311.pdf.

[62] Vasily Shishkin, Denis Dygin, Ivan Lavrikov, Grigory Marshalko, Vladimir Rudskoy, and Dmitry Trifonov. Low-weight and hi-end: Draft Russian Encryption Standard. In *CTCrypt '14, 05-06 June 2014, Moscow, Russia. Preproceedings.*, pages 183–188, June 2014.

[63] Michael Weiß, Benedikt Heinz, and Frederic Stumpf. A cache timing attack on AES in virtualization environments. In Angelos D. Keromytis, editor, *FC 2012*, volume 7397 of *LNCS*, pages 314–328. Springer, 2013. doi:10.1007/978-3-642-32946-3_23.

Document version 20150828143000.