The STRIBOBr1 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.stribob.com

Norwegian University of Science and Technology

Contents

Preface **1**

1 Specification **2**
1.1 STRIBOBr1 Family and Parameters . . . 2
1.2 Structure of the π Permutation . . . 2
1.2.1 Matrix Notation and Finite Field Arithmetic . . . 3
1.2.2 Round Constants C_i . . . 4
1.2.3 Substitution $m' = S(m)$ . . . 5
1.2.4 Permutation $m' = P(m)$ . . . 5
1.2.5 Linear transform $m' = L(m)$ . . . 5
1.2.6 Example Computation of π . . . 6
1.3 BLNK Sponge Mode and Padding . . . 7
1.3.1 BLNK Block Operations . . . 7
1.3.2 The CAESAR encrypt() and decrypt() AEAD API . . . 8
1.4 Trace of `stribob192r1` Computation . . . 9

2 Security Goals **10**
2.1 Specific Goals . . . 10
2.2 Nonce Re-Use . . . 10
2.3 General Goals . . . 10

3 Security Analysis **11**
3.1 Structure of GOST R 34.11-2012 . . . 11
3.1.1 Streebog Compression Function $g_N(h, m)$ . . . 11
3.2 Security of LPS Against Classical Attacks . . . 12
3.3 Security Reduction Between StriBob 's π and Streebog's g . . . 12
3.4 Sponge Functions . . . 13
3.4.1 Absorbing and Squeezing . . . 13
3.4.2 Duplexing . . . 13
3.4.3 MAC-and-Continue . . . 14
3.4.4 Duplex, Triplex, Multiplex . . . 14
3.4.5 Multiplexing the Sponge . . . 15
3.4.6 Domain Separation and Capacity Reduction . . . 15

4 Features **16**
4.1 Historical, Scientific, and Regulatory Context . . . 16
4.2 Advantages . . . 16
4.3 Implementation Issues . . . 17
4.3.1 Low-Resource Software Platforms . . . 17
4.3.2 Medium- to High-Resource Software Platforms . . . 18

5 Design Rationale **19**
5.1 Parameter and Component Selection . . . 19
5.2 Hidden Weaknesses . . . 19

6 Intellectual Property and Consent **20**
6.1 Intellectual Property 20
6.2 Consent to CAESAR Selection Committee 20

Bibliography **21**

Preface

This is a specification for STRIBOBr1, 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 [31] and STRIBOB, 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, 32].

This is the version 1.20140315200000 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, 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 STRIBOBr1 Family and Parameters

StriBob is an algorithm for Authenticated Encryption with Associated Data (AEAD). StriBob accepts almost arbitrary ranges for its input parameters; however for CAESAR we propose a concrete parameter set ``STRIBOB192r1'' 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 first give a mathematical description of the cryptographic permutation π 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.

1.2 Structure of the π Permutation

StriBob is uses a 512×512 - bit permutation π as its cryptographic foundation. π in turn is built from twelve iterations of LPS transformation, interleaved with exclusive-or operation with round constants. The LPS core component and round constants of StriBob are lifted from the Russian GOST R 34.11-2012 "Streebog" hash standard [18], which is also specified in IETF RFC 6986 [14].

With twelve 512-bit round constants C_i (Section 1.2.2) we define the permutation $\pi(X_1) = X_{13}$ via:

$$X_{i+1} = \mathsf{LPS}(X_i \oplus C_i) \text{ for } 1 \leq i \leq 12. \tag{1.1}$$

LPS consists of three steps, which are (in order of execution): S substitution, P permutation, and L linear step; see Figure 1.1. We abbreviate the composite function $L(P(S(V))) = (L \circ P \circ S)(V)$ as LPS .

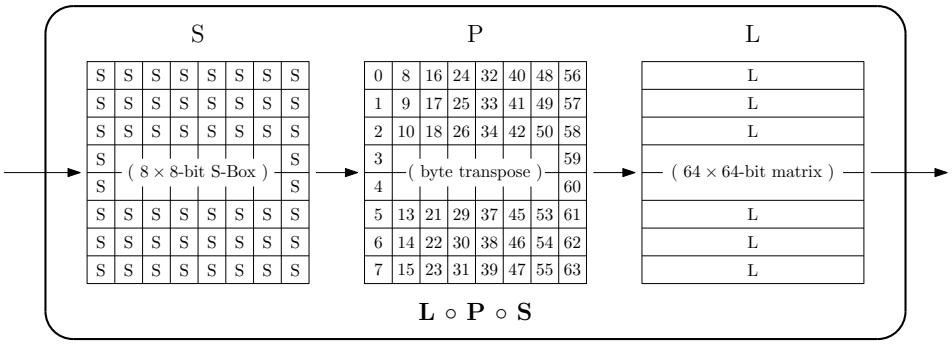

Figure 1.1: LPS consists of a byte substitution layer S, byte transpose P, and linear layer L.

1.2.1 Matrix Notation and Finite Field Arithmetic

We use C-style indexing for the 512-bit state as a matrix of 8×8 bytes (octets) $m[\,0 \cdots 7\,][\,0 \cdots 7\,]$. Here $m[\,i\,][\,j\,]$ indicates a byte at row i and column j, both indexed from zero:

$$\begin{pmatrix} m[\,0\,][\,0\,] & m[\,0\,][\,1\,] & m[\,0\,][\,2\,] & \dots & m[\,0\,][\,7\,] \\ m[\,1\,][\,0\,] & m[\,1\,][\,1\,] & m[\,1\,][\,2\,] & \dots & m[\,1\,][\,7\,] \\ m[\,2\,][\,0\,] & m[\,2\,][\,1\,] & m[\,2\,][\,2\,] & \dots & m[\,2\,][\,7\,] \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ m[\,7\,][\,0\,] & m[\,7\,][\,1\,] & m[\,7\,][\,2\,] & \dots & m[\,7\,][\,7\,] \end{pmatrix}$$

The matrices are serialized as byte sequences for transmission in straightforward fashion:

$$V[\,i\,] = m[\,\lfloor i/8 \rfloor\,][\,i \mod 8\,] \text{ for } i = 0, 1, \cdots, 63. \tag{1.2}$$

Implementors on mid- and high-end software target platforms will typically store each row in a 64-bit register for computation. For storage and transmission, the big-endian (network) byte order of Equation 1.2 should be used.[1]

StriBob performs arithmetic in the finite field $\mathbb{F}_2^8$, where elements are binary polynomials of degree 7 (or less). Each polynomial may be encoded as a byte by their coefficient vector of 8 binary digits. Encoding of a binary polynomial $f(x) = \sum_{i=0}^{7} f_i x^i,\ f_i \in \{0, 1\}$ as an 8-bit byte is performed with low-order monomial on the left: [2]

$$\mathsf{val}(f) = \sum_{i=0}^{7} 2^{7-i} f_i. \tag{1.3}$$

In hexadecimal we have $\mathsf{val}(x^7) = \mathtt{01}$, $\mathsf{val}(x^6) = \mathtt{02}$, $\cdots$, $\mathsf{val}(x) = \mathtt{40}$, $\mathsf{val}(1) = \mathtt{80}$. Since $x + x \equiv 0 \pmod 2$, addition of two polynomials is equivalent to bitwise exclusive-or operation $\mathsf{val}(f + g) = \mathsf{val}(f) \oplus \mathsf{val}(g)$.

For field multiplication the irreducible polynomial basis is $p(x) = x^8 + x^6 + x^5 + x^4 + 1$. Polynomial multiplication is carried out normally - but with each i-degree monomial canceling each out as this is a binary field:

$$\Big(\sum_{i=0}^{7} f_i x^i\Big)\Big(\sum_{i=0}^{7} g_i x^i\Big) = \sum_{i=0}^{7}\sum_{j=0}^{7} f_i g_j x^{i+j} \pmod 2. \tag{1.4}$$

The result is reduced mod $p(x)$, limiting its degree to 7.

$$\begin{aligned} x^i \bmod p(x) &= x^i \ \text{ for } 0 \le i \le 7 & x^{11} \bmod p(x) &= x^4 + x^3 + x + 1 \\ x^8 \bmod p(x) &= x^6 + x^5 + x^4 + 1 & x^{12} \bmod p(x) &= x^5 + x^4 + x^2 + x \\ x^9 \bmod p(x) &= x^7 + x^6 + x^5 + x & x^{13} \bmod p(x) &= x^6 + x^5 + x^3 + x^2 \\ x^{10} \bmod p(x) &= x^7 + x^5 + x^4 + x^2 + 1 & x^{14} \bmod p(x) &= x^7 + x^6 + x^4 + x^3. \end{aligned}$$

Example. Let's multiply $\mathsf{val}(f) = \mathtt{45}$ with $\mathsf{val}(g) = \mathtt{8E}$. From Eqn. 1.3 we see that corresponding polynomials are $f = x^7 + x^5 + x$ and $g = x^6 + x^5 + x^4 + 1$. Formal product fg and its equivalent in $\mathbb{F}_2$:

$$\begin{aligned} fg &= x^{13} + x^{12} + 2x^{11} + x^{10} + x^9 + 2x^7 + x^6 + x^5 + x \\ fg &\equiv x^{13} + x^{12} + x^{10} + x^9 + x^6 + x \pmod 2 \end{aligned}$$

Further reducing mod $p(x)$ and converting to a byte value we have:

$$\begin{aligned} fg \mod p(x) &= x^6 + x^3 + x^2 + x + 1 \\ \mathtt{45} * \mathtt{8E} &= \mathsf{val}(x^6 + x^3 + x^2 + x + 1) = \mathtt{F2}. \end{aligned}$$

[1] The Streebog hash standard does not specify the byte order for transmission [18].

[2] This "bit-reverse" convention is carried over from the GOST standard, but is also used in other cryptographic standards such as NIST's GCM Mode [26], but not in AES [25].

1.2.2 Round Constants C_i

The twelve round constants C_i are from the Streebog specification, here given as byte matrices [14, 18]:

$$C_1 = \begin{pmatrix} \texttt{B1 08 5B DA 1E CA DA E9} \\ \texttt{EB CB 2F 81 C0 65 7C 1F} \\ \texttt{2F 6A 76 43 2E 45 D0 16} \\ \texttt{71 4E B8 8D 75 85 C4 FC} \\ \texttt{4B 7C E0 91 92 67 69 01} \\ \texttt{A2 42 2A 08 A4 60 D3 15} \\ \texttt{05 76 74 36 CC 74 4D 23} \\ \texttt{DD 80 65 59 F2 A6 45 07} \end{pmatrix} \quad C_7 = \begin{pmatrix} \texttt{F4 C7 0E 16 EE AA C5 EC} \\ \texttt{51 AC 86 FE BF 24 09 54} \\ \texttt{39 9E C6 C7 E6 BF 87 C9} \\ \texttt{D3 47 3E 33 19 7A 93 C9} \\ \texttt{09 92 AB C5 2D 82 2C 37} \\ \texttt{06 47 69 83 28 4A 05 04} \\ \texttt{35 17 45 4C A2 3C 4A F3} \\ \texttt{88 86 56 4D 3A 14 D4 93} \end{pmatrix}$$

$$C_2 = \begin{pmatrix} \texttt{6F A3 B5 8A A9 9D 2F 1A} \\ \texttt{4F E3 9D 46 0F 70 B5 D7} \\ \texttt{F3 FE EA 72 0A 23 2B 98} \\ \texttt{61 D5 5E 0F 16 B5 01 31} \\ \texttt{9A B5 17 6B 12 D6 99 58} \\ \texttt{5C B5 61 C2 DB 0A A7 CA} \\ \texttt{55 DD A2 1B D7 CB CD 56} \\ \texttt{E6 79 04 70 21 B1 9B B7} \end{pmatrix} \quad C_8 = \begin{pmatrix} \texttt{9B 1F 5B 42 4D 93 C9 A7} \\ \texttt{03 E7 AA 02 0C 6E 41 41} \\ \texttt{4E B7 F8 71 9C 36 DE 1E} \\ \texttt{89 B4 44 3B 4D DB C4 9A} \\ \texttt{F4 89 2B CB 92 9B 06 90} \\ \texttt{69 D1 8D 2B D1 A5 C4 2F} \\ \texttt{36 AC C2 35 59 51 A8 D9} \\ \texttt{A4 7F 0D D4 BF 02 E7 1E} \end{pmatrix}$$

$$C_3 = \begin{pmatrix} \texttt{F5 74 DC AC 2B CE 2F C7} \\ \texttt{0A 39 FC 28 6A 3D 84 35} \\ \texttt{06 F1 5E 5F 52 9C 1F 8B} \\ \texttt{F2 EA 75 14 B1 29 7B 7B} \\ \texttt{D3 E2 0F E4 90 35 9E B1} \\ \texttt{C1 C9 3A 37 60 62 DB 09} \\ \texttt{C2 B6 F4 43 86 7A DB 31} \\ \texttt{99 1E 96 F5 0A BA 0A B2} \end{pmatrix} \quad C_9 = \begin{pmatrix} \texttt{37 8F 5A 54 16 31 22 9B} \\ \texttt{94 4C 9A D8 EC 16 5F DE} \\ \texttt{3A 7D 3A 1B 25 89 42 24} \\ \texttt{3C D9 55 B7 E0 0D 09 84} \\ \texttt{80 0A 44 0B DB B2 CE B1} \\ \texttt{7B 2B 8A 9A A6 07 9C 54} \\ \texttt{0E 38 DC 92 CB 1F 2A 60} \\ \texttt{72 61 44 51 83 23 5A DB} \end{pmatrix}$$

$$C_4 = \begin{pmatrix} \texttt{EF 1F DF B3 E8 15 66 D2} \\ \texttt{F9 48 E1 A0 5D 71 E4 DD} \\ \texttt{48 8E 85 7E 33 5C 3C 7D} \\ \texttt{9D 72 1C AD 68 5E 35 3F} \\ \texttt{A9 D7 2C 82 ED 03 D6 75} \\ \texttt{D8 B7 13 33 93 52 03 BE} \\ \texttt{34 53 EA A1 93 E8 37 F1} \\ \texttt{22 0C BE BC 84 E3 D1 2E} \end{pmatrix} \quad C_{10} = \begin{pmatrix} \texttt{AB BE DE A6 80 05 6F 52} \\ \texttt{38 2A E5 48 B2 E4 F3 F3} \\ \texttt{89 41 E7 1C FF 8A 78 DB} \\ \texttt{1F FF E1 8A 1B 33 61 03} \\ \texttt{9F E7 67 02 AF 69 33 4B} \\ \texttt{7A 1E 6C 30 3B 76 52 F4} \\ \texttt{36 98 FA D1 15 3B B6 C3} \\ \texttt{74 B4 C7 FB 98 45 9C ED} \end{pmatrix}$$

$$C_5 = \begin{pmatrix} \texttt{4B EA 6B AC AD 47 47 99} \\ \texttt{9A 3F 41 0C 6C A9 23 63} \\ \texttt{7F 15 1C 1F 16 86 10 4A} \\ \texttt{35 9E 35 D7 80 0F FF BD} \\ \texttt{BF CD 17 47 25 3A F5 A3} \\ \texttt{DF FF 00 B7 23 27 1A 16} \\ \texttt{7A 56 A2 7E A9 EA 63 F5} \\ \texttt{60 17 58 FD 7C 6C FE 57} \end{pmatrix} \quad C_{11} = \begin{pmatrix} \texttt{7B CD 9E D0 EF C8 89 FB} \\ \texttt{30 02 C6 CD 63 5A FE 94} \\ \texttt{D8 FA 6B BB EB AB 07 61} \\ \texttt{20 01 80 21 14 84 66 79} \\ \texttt{8A 1D 71 EF EA 48 B9 CA} \\ \texttt{EF BA CD 1D 7D 47 6E 98} \\ \texttt{DE A2 59 4A C0 6F D8 5D} \\ \texttt{6B CA A4 CD 81 F3 2D 1B} \end{pmatrix}$$

$$C_6 = \begin{pmatrix} \texttt{AE 4F AE AE 1D 3A D3 D9} \\ \texttt{6F A4 C3 3B 7A 30 39 C0} \\ \texttt{2D 66 C4 F9 51 42 A4 6C} \\ \texttt{18 7F 9A B4 9A F0 8E C6} \\ \texttt{CF FA A6 B7 1C 9A B7 B4} \\ \texttt{0A F2 1F 66 C2 BE C6 B6} \\ \texttt{BF 71 C5 72 36 90 4F 35} \\ \texttt{FA 68 40 7A 46 64 7D 6E} \end{pmatrix} \quad C_{12} = \begin{pmatrix} \texttt{37 8E E7 67 F1 16 31 BA} \\ \texttt{D2 13 80 B0 04 49 B1 7A} \\ \texttt{CD A4 3C 32 BC DF 1D 77} \\ \texttt{F8 20 12 D4 30 21 9F 9B} \\ \texttt{5D 80 EF 9D 18 91 CC 86} \\ \texttt{E7 1D A4 AA 88 E1 28 52} \\ \texttt{FA F4 17 D5 D9 B2 1B 99} \\ \texttt{48 BC 92 4A F1 1B D7 20} \end{pmatrix}$$

1.2.3 Substitution $m' = S(m)$

In this step, a 8×8 - bit S-Box is applied to each byte (octet) of data on vector.

$$m'[\,i\,][\,j\,] = S(m[\,i\,][\,j\,]) \text{ for } 0 \leq i, j \leq 7 \tag{1.5}$$

The substitute bytes $S(0) = 252, S(1) = 238, \cdots S(255) = 182$ are given by the following table (in hex):

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

1.2.4 Permutation $m' = P(m)$

A permutation of bytes in the state, equivalent of transposing the 8×8 byte matrix. Transposition is a reflection along the main diagonal, which can also be seen as writing rows as columns:

$$m'[\,i\,][\,j\,] = m[\,j\,][\,i\,] \text{ for } 0 \leq i, j \leq 7. \tag{1.6}$$

$$\begin{pmatrix} 00\ 01\ 02\ 03\ 04\ 05\ 06\ 07 \\ 08\ 09\ 0A\ 0B\ 0C\ 0D\ 0E\ 0F \\ 10\ 11\ 12\ 13\ 14\ 15\ 16\ 17 \\ 18\ 19\ 1A\ 1B\ 1C\ 1D\ 1E\ 1F \\ 20\ 21\ 22\ 23\ 24\ 25\ 26\ 27 \\ 28\ 29\ 2A\ 2B\ 2C\ 2D\ 2E\ 2F \\ 30\ 31\ 32\ 33\ 34\ 35\ 36\ 37 \\ 38\ 39\ 3A\ 3B\ 3C\ 3D\ 3E\ 3F \end{pmatrix}^T = \begin{pmatrix} 00\ 08\ 10\ 18\ 20\ 28\ 30\ 38 \\ 01\ 09\ 11\ 19\ 21\ 29\ 31\ 39 \\ 02\ 0A\ 12\ 1A\ 22\ 2A\ 32\ 3A \\ 03\ 0B\ 13\ 1B\ 23\ 2B\ 33\ 3B \\ 04\ 0C\ 14\ 1C\ 24\ 2C\ 34\ 3C \\ 05\ 0D\ 15\ 1D\ 25\ 2D\ 35\ 3D \\ 06\ 0E\ 16\ 1E\ 26\ 2E\ 36\ 3E \\ 07\ 0F\ 17\ 1F\ 27\ 2F\ 37\ 3F \end{pmatrix}$$

1.2.5 Linear transform $m' = L(m)$

Even though L is specified in the form of a 64×64 - bit matrix in the Standard text [18], it is in fact built from a matrix multiplication in the finite field $\mathbb{F}_{2^8}$ [19]. The mixing operation and matrix constant L are:

$$m' = m \cdot L, \quad L = \begin{pmatrix} 8E\ 20\ FA\ A7\ 2B\ A0\ B4\ 70 \\ A0\ 11\ D3\ 80\ 81\ 8E\ 8F\ 40 \\ 90\ DA\ B5\ 2A\ 38\ 7A\ E7\ 6F \\ 9D\ 4D\ F0\ 5D\ 5F\ 66\ 14\ 51 \\ 86\ 27\ 5D\ F0\ 9C\ E8\ AA\ A8 \\ 45\ 6C\ 34\ 88\ 7A\ 38\ 05\ B9 \\ E4\ FA\ 20\ 54\ A8\ 0B\ 32\ 9C \\ 70\ A6\ A5\ 6E\ 24\ 40\ 59\ 8E \end{pmatrix} \tag{1.7}$$

$$m'[\,i\,][\,j\,] = \bigoplus_{k=0}^{7} m[\,i\,][\,k\,] * L[\,k\,][\,j\,]. \tag{1.8}$$

1.2.6 Example Computation of π

We will give an example of computation of π in this section. One may also utilize pages 20-24 of RFC 6986 [14] where the computation of `K[i]` is mathematically equivalent to X_i iteration with $\pi(\texttt{K[1]}) = \texttt{K[13]}$. That example also computes many quantities that are specific to Streebog and not needed in StriBob .

We begin iteration of Equation 1.1 with initial value set to ascending byte sequence $X_1 = (\,0, 1, 2, \cdots, 63\,)$. First round constant addition yields:

$$X_1 \oplus C_1 = \begin{pmatrix} \texttt{00 01 02 03 04 05 06 07} \\ \texttt{08 09 0A 0B 0C 0D 0E 0F} \\ \texttt{10 11 12 13 14 15 16 17} \\ \texttt{18 19 1A 1B 1C 1D 1E 1F} \\ \texttt{20 21 22 23 24 25 26 27} \\ \texttt{28 29 2A 2B 2C 2D 2E 2F} \\ \texttt{30 31 32 33 34 35 36 37} \\ \texttt{38 39 3A 3B 3C 3D 3E 3F} \end{pmatrix} \oplus \begin{pmatrix} \texttt{B1 08 5B DA 1E CA DA E9} \\ \texttt{EB CB 2F 81 C0 65 7C 1F} \\ \texttt{2F 6A 76 43 2E 45 D0 16} \\ \texttt{71 4E B8 8D 75 85 C4 FC} \\ \texttt{4B 7C E0 91 92 67 69 01} \\ \texttt{A2 42 2A 08 A4 60 D3 15} \\ \texttt{05 76 74 36 CC 74 4D 23} \\ \texttt{DD 80 65 59 F2 A6 45 07} \end{pmatrix} = \begin{pmatrix} \texttt{B1 09 59 D9 1A CF DC EE} \\ \texttt{E3 C2 25 8A CC 68 72 10} \\ \texttt{3F 7B 64 50 3A 50 C6 01} \\ \texttt{69 57 A2 96 69 98 DA E3} \\ \texttt{6B 5D C2 B2 B6 42 4F 26} \\ \texttt{8A 6B 00 23 88 4D FD 3A} \\ \texttt{35 47 46 05 F8 41 7B 14} \\ \texttt{E5 B9 5F 62 CE 9B 7B 38} \end{pmatrix}$$

Applying the S-Box substitution and the transpose of P permutation we have:

$$S(X_1 \oplus C_1) = \begin{pmatrix} \texttt{45 C4 72 53 F1 89 CA 6C} \\ \texttt{A4 B3 5C D6 88 F3 19 E9} \\ \texttt{1F 80 10 B5 ED B5 1D EE} \\ \texttt{91 12 60 B0 91 DC AA A4} \\ \texttt{6F B7 B3 46 55 2C CC EF} \\ \texttt{D6 6F FC 5A D7 3A 4B ED} \\ \texttt{6A AB 48 6E D1 34 80 93} \\ \texttt{2B A3 87 96 E7 50 80 06} \end{pmatrix} \quad P(S(X_1 \oplus C_1) = \begin{pmatrix} \texttt{45 A4 1F 91 6F D6 6A 2B} \\ \texttt{C4 B3 80 12 B7 6F AB A3} \\ \texttt{72 5C 10 60 B3 FC 48 87} \\ \texttt{53 D6 B5 B0 46 5A 6E 96} \\ \texttt{F1 88 ED 91 55 D7 D1 E7} \\ \texttt{89 F3 B5 DC 2C 3A 34 50} \\ \texttt{CA 19 1D AA CC 4B 80 80} \\ \texttt{6C E9 EE A4 EF ED 93 06} \end{pmatrix}.$$

We illustrate the finite field matrix multiplication (Eqn. 1.8) of the L step by computing one element of $m = P(S(X_1 \oplus C_1)) \cdot L$. Let's consider, say, $m[\,1\,][\,3\,]$, the byte at offset $1 * 8 + 3 = 11$. The value of this byte will equal the dot product of row 1 of $P(S(X_1 \oplus C_1))$ and column 3 of the L matrix (Eqn. 1.7):

$$\begin{aligned} m[\,1\,][\,3\,] &= (\ \texttt{C4 B3 80 12 B7 6F AB A3}\ ) \cdot (\ \texttt{A7 80 2A 5D F0 88 54 6E}\ )^T \\ &= (\texttt{C4} * \texttt{A7}) \oplus (\texttt{B3} * \texttt{80}) \oplus (\texttt{80} * \texttt{2A}) \oplus (\texttt{12} * \texttt{5D}) \oplus (\texttt{B7} * \texttt{F0}) \oplus (\texttt{6F} * \texttt{88}) \oplus (\texttt{AB} * \texttt{54}) \oplus (\texttt{A3} * \texttt{6E}) \\ &= \texttt{66} \oplus \texttt{B3} \oplus \texttt{2A} \oplus \texttt{2D} \oplus \texttt{F4} \oplus \texttt{D5} \oplus \texttt{D2} \oplus \texttt{26} \\ &= \underline{\textbf{07}}. \end{aligned}$$

The result of the L step (and the first round) is

$$X_2 = P(S(X_1 \oplus C_1)) \cdot L = (L \circ P \circ S)(X_1 \oplus C_1) = \begin{pmatrix} \texttt{35 B0 E5 15 D4 2D CC D5} \\ \texttt{72 63 04 } \underline{\textbf{07}} \texttt{ 96 4A E5 16} \\ \texttt{6B EA FD 00 FF E3 A6 93} \\ \texttt{96 66 64 04 BF AE 69 5D} \\ \texttt{9A 09 63 C6 04 D4 BE 0E} \\ \texttt{9C 57 21 98 7D 19 8F 27} \\ \texttt{DB 2B F6 9D 02 43 5E AB} \\ \texttt{27 A6 4E 75 07 79 F3 89} \end{pmatrix}$$

Further eleven rounds yields the final result of π:

$$X_3 = \begin{pmatrix} \texttt{D3 A7 1B D2 2F 79 0D B7} \\ \texttt{DB 15 9B 6A 51 EA 8F 54} \\ \texttt{C9 FA F5 66 DC E6 C3 4B} \\ \texttt{34 74 A0 ED 5D DE E8 58} \\ \texttt{79 15 83 86 64 48 DE 68} \\ \texttt{0E 78 6F 4A 09 B6 1A D0} \\ \texttt{AF D7 5F F3 CC B6 22 D2} \\ \texttt{44 23 4C A0 AD E1 B9 C4} \end{pmatrix} \quad \pi(X_1) = X_{13} = \begin{pmatrix} \texttt{16 8A 86 7D 30 DB 56 6D} \\ \texttt{57 D5 30 BE D9 22 08 82} \\ \texttt{37 0C E2 79 FB A4 E5 87} \\ \texttt{A3 20 E6 ED A2 A3 BA 10} \\ \texttt{17 34 62 B6 23 0E C5 67} \\ \texttt{86 7C 34 37 5E 2E 46 D9} \\ \texttt{A7 FB 06 19 27 A3 F5 49} \\ \texttt{53 19 BD F9 EC 94 1A 95} \end{pmatrix}$$

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 `STRIBOB192r1` 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 $b = 512$ bits.
- r Data rate or block size. $r = 256$ bits.
- c Capacity, the amount of secret information in the state. $c = b - \delta$ bits.
- δ Capacity consumed by padding. For `STRIBOB192r1` we can bound this to $\delta < 2$ bits.

Furthermore, we fix the key size to $k = 192$ bits and the authentication tag to $t = 128$ 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 $\max(\lceil n/32 \rceil, 1)$ 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 $V[\,0 \cdots 63\,]$ is the state.

```
1:  i ← 0                                          state index, initialized to first byte
2:  for j = 0 to n − 1 do
3:    if i = 32 then
4:      V[ 32 ] ← V[ 32 ] ⊕ BLNK_END ⊕ pad         full block padding with block end marker
5:      V ← π(V)                                   cryptographic permutation
6:      i ← 0                                      zero index
7:    end if
8:    if op = put then
9:      V[ i ] ← V[ i ] ⊕ D[ j ]                   XOR input data to the state
10:   else if op = get then
11:     D[ j ] ← V[ i ]                            simply save the data
12:   else if op = enc then
13:     C[ j ] ← V[ i ] ⊕ P[ j ]                   encrypt as in a stream cipher
14:     V[ i ] ← C[ j ]                            store ciphertext in state
15:   else if op = dec then
16:     P[ j ] ← V[ i ] ⊕ C[ j ]                   decrypt as in a stream cipher
17:     V[ i ] ← C[ j ]                            store ciphertext in state
18:   end if
19:   i ← i + 1                                    advance block index
20: end for
21: V[ i ] ← V[ i ] ⊕ BLNK_END                     end marker (note: i = 32 possible)
22: V[ 32 ] ← V[ 32 ] ⊕ BLNK_FIN ⊕ pad             final padding
23: V ← π(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, $V[\,0 \cdots 63\,]$ is the 64-byte internal state of StriBob .

$K[\,24\,]$ Secret key of $k = 192$ bits, or 24 bytes.

$N[\,16\,]$ A 128-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, $0 \leq a$ 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, $0 \leq n$ 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 AEAD API encryption:

$C[\,n+16\,] \leftarrow$ encrypt($K[\,24\,]$, $N[\,16\,]$, $A[\,a\,]$, $P[\,n\,]$)

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

Inverse operation by the recipient:

{ $P[\,n\,]$ or FAIL } ← decrypt($K[\,24\,]$, $N[\,16\,]$, $A[\,a\,]$, $C[\,n+16\,]$)

```
1: V[ 64 ] ← ( 0, 0, ··· , 0 )          initialize the state with zeros
2: put( K[ 24 ], BLNK_KEY )             secret key, always a single π op
3: put( N[ 16 ], BLNK_NPUB )            public nonce, always a single π op
4: put( A[ a ], BLNK_AAD )              authenticated data, ⌈a/32⌉ π ops
5: P[ n ] ← dec( P[ 0 ··· n − 1 ], BLNK_MSG )   decryption, ⌈n/32⌉ π ops
6: if C[ n ··· n + 15 ] = get( 16 , BLNK_MAC ) then
7:    return P[ N ]                     auth match: C[ n ··· n + 15 ] = V[ 0 ··· 15 ]
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 `stribob192r1` Computation

To illustrate the operation with CAESAR parameters, we use following plain ASCII values for input to encrypt() with $n = 38$:

$$\begin{array}{rcl} K[\,24\,] & = & \texttt{"192-bit Secret Key value"} \\ N[\,16\,] & = & \texttt{"Nonces Used Once"} \\ A[\,32\,] & = & \texttt{"AAD Test Vector Exact Block 32 B"} \\ P[\,38\,] & = & \texttt{"This is a Test Vector for stribob192r1"} \end{array}$$

Steps 1-2: Keying. After zeroing V, the first input to π is the padded secret key value $K[\,24\,]$:

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

The state $V[\,64\,]$ after key mixing π is:

08 4D 08 20 FA 4A C8 C7 E0 0D CB 0A 3E 2F 5B 20 A2 77 0B A2 D2 90 02 BB 0C F2 D8 B7 7D 4E 9F 94
B4 E4 A6 C1 0D 3E 05 7A 29 87 39 75 9E 90 0E 0B 6B 06 5D 55 92 36 89 7A C0 CA 22 30 84 8D 5D 77

Step 3: Nonce mixing. The nonce $N[\,16\,]$ padded XOR value:

4E 6F 6E 63 65 73 20 55 73 65 64 20 4F 6E 63 65 **01** 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
22 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

The state V after nonce mixing π is:

94 09 14 15 F3 64 F7 4B 22 A9 4A BE E7 31 14 D0 5F E3 BA 4B 1D 98 51 12 0F 70 ED DA F1 F9 F5 B0
DF 6F 47 BD BF B9 F2 57 D0 B8 4A 4E DF AF 20 E5 19 FF EF 7B 6C B8 23 BE D4 FD F5 0A 5C F5 CD 43

Step 4: Associated Authenticated Data. Padded AAD $A[\,32\,]$ XOR value. Note how the end bit is embedded with domain indicator byte at position 32, 0x43 = BLNK_END $\oplus$ BLNK_FIN $\oplus$ BLNK_AAD.

41 41 44 20 54 65 73 74 20 56 65 63 74 6F 72 20 45 78 61 63 74 20 42 6C 6F 63 6B 20 33 32 20 42
43 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

The state V after AAD mixing π is:

39 E8 76 FD 1F A6 DB 05 FC 68 1E CA C8 03 A2 A4 8B 6C B3 0E 6B 47 D9 FE C9 4F E1 E8 CB 3E 02 D4
73 48 03 FB 16 F3 6A 56 53 DE FE BC 70 12 C2 8C 94 91 72 CA EC 12 74 E1 9A 7C 51 32 AF E5 8E AC

Step 5: Payload Encryption. First 32 bytes of plaintext $P[\,0 \cdots 31\,]$ with padding:

54 68 69 73 20 69 73 20 61 20 54 65 73 74 20 56 65 63 74 6F 72 20 66 6F 72 20 73 74 72 69 62 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 00 00 00 00 00 00 00 00

The state V after encrypting first 32 bytes of plaintext:

CA 39 E7 1B 5D BA DA 03 2D DB 04 19 25 22 AF 27 83 73 E6 C4 2E F1 D5 7F 4B 18 F7 01 DD 03 38 F9
C7 C4 F5 A4 F7 79 AB 4F FB A1 45 FC ED 0E 74 65 9A EA 3C E3 3A 72 25 5D 97 6B F9 76 B3 CA C8 CD

Remaining 6 bytes of plaintext $P[\,32 \cdots 37\,]$ with padding:

62 31 39 32 72 31 **01** 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
52 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

The state V after encryption and final π:

16 5B D9 D6 2B 3C 7B 7D 6D C4 23 44 6B E7 60 82 BB 3A 5C A8 D2 50 C4 E8 3A 58 58 F0 89 94 00 95
1B ED ED 65 63 9C 41 5B 6B 90 90 51 BA CD 55 EE 8E 63 4A C7 63 A7 55 B3 77 E0 7A CE B0 FD A8 CF

Step 6: Authentication code. Authenticated ciphertext bytes $C[\,54\,]$. The last 16 bytes correspond to first 16 bytes of the final state above.

6D 80 1F 8E 3F CF A8 25 9D 48 4A AF BB 77 82 F2 EE 0F C7 61 19 67 BF 91 BB 6F 92 9C B9 57 60 BB
A8 08 DE 29 2F 8B 16 5B D9 D6 2B 3C 7B 7D 6D C4 23 44 6B E7 60 82

Chapter 2

Security Goals

2.1 Specific Goals

With the "`stribob192r1`" 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^{191}	To recover to the plaintext from ciphertext or vice versa.
Integrity for the plaintext.	2^{127}	To forge plaintext payload.
Integrity for the associated data.	2^{127}	To forge Associated Data .
Integrity for the public message number.	2^{127}	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 / nonce pair. The "unit" for the effort is equivalent to the effort required to compute the π permutation.

2.2 Nonce Re-Use

StriBob 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**. StriBob 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 [27] and Duplex Sponges in particular -- proofs in [4, 7] 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

StriBob is based on the new Russian GOST R 34.11-2012 "Streebog" hash standard [18] and a variant of Saarinen's Blinker padding for the Sponge Authenticated Encryption construct of [4, 29]

Streebog is not a sponge-based construction and uses the LPS core in an entirely different way, yet the similarities allow certain types of security reductions between the two algorithms.

3.1 Structure of GOST R 34.11-2012

We first recall the structure of GOST R 34.11-2012 hash function. Streebog produces either a 256-bit or a 512-bit hash from a bit string of arbitrary size using the Merkle-Damgård [13, 24] iterative method without any randomization.

Figure 3.1 gives an overview of the hashing process. Padded message M is processed in 512-bit blocks $M = m_0 \mid m_1 \mid \cdots \mid m_n$ by a compression function $h' = g_N(h, m_i)$. The chaining variable h also has 512 bits and N denotes the index bit offset of the input block. After the last message block, there are finalization steps involving two invocations of the compression function, first on the total bit length of input, and then on checksum ϵ, which is computed over all input blocks mod 2^{512}.

3.1.1 Streebog Compression Function $g_N(h, m)$

The compression function $h' = g_N(h, m)$ takes in a chaining variable h, message block m, a position index variable N, and produces a new chaining value h'. The compression function is built from a keyless 512-bit nonlinear permutation LPS and 512-bit vector XOR operations. The compression function has 12 rounds and a performs a total of 25 invocations of LPS :

$$\begin{aligned} [K_1, X_1] &= [\, \mathsf{LPS}(h \oplus N),\ m\,] \\ [K_{i+1}, X_{i+1}] &= [\, \mathsf{LPS}(K_i \oplus C_i), \mathsf{LPS}(X_i \oplus K_i)\,] \ \text{ for } \ 1 \le i \le 12 \\ g_N(h, m) &= K_{13} \oplus X_{13} \oplus h \oplus m. \end{aligned}$$

Figure 3.2 shows the structure of g. We can view it as a two-track substitution-permutation network where input value $h \oplus N$ and a set of 12 round constants C_i is used to key (via K_i) another substitution-permutation network operating on h . The outputs of the two tracks are finally XORed together with

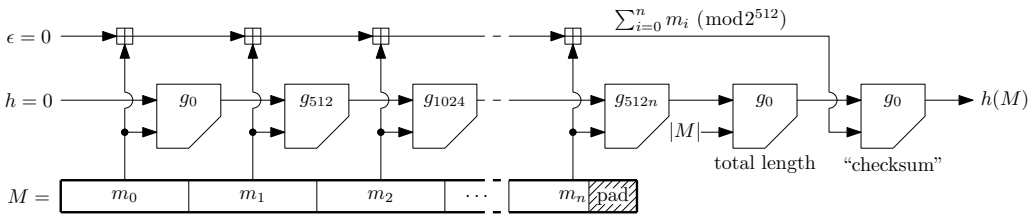

Figure 3.1: Operation of Streebog with 512-bit output. For 256-bit hashes, the initial h value is changed to `0x010101..01` and the output $h(M)$ is truncated to 256 bits.

original values of h and m. We note that h together with offset N uniquely defines all K_i subkey values for each invocation of g.

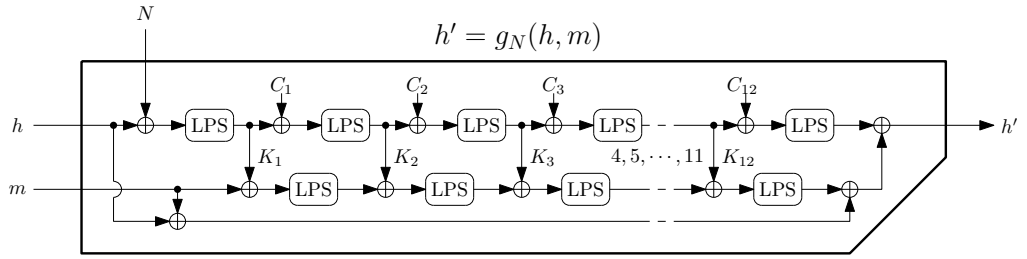

Figure 3.2: Streebog compression function. All data paths, inputs, and outputs are 512-bit vectors. Here the $\oplus$ symbol denotes the XOR operation between two 512-bit vectors.

3.2 Security of LPS Against Classical Attacks

LPS gets all of its non-linearity from the 8-bit S-box S, which apparently has been designed to offer resistance against classical methods of cryptanalysis. Its differential bound [10] is $P = \frac{8}{256}$ and best linear approximation [21] holds with $P = \frac{28}{128}$. There seems to be no exploitable algebraic weaknesses.

The linear transform L is not randomly constructed even though it is expressed without explanation as a 64×64 binary matrix in [18]. L in fact has a byte-oriented structure as an MDS matrix with $\mathbb{F}_{2^8}$ arithmetic in a similar fashion as AES, even though this is not mentioned in the standard specification [19, 25]. We use this equivalent description (Section 1.2.5). Many structural observations on AES-like ciphers also apply to LPS : S and L are effectively mix together the bits of the eight 64-bit rows. P swaps rows and columns and after two rounds each input bit affects each output bit of the 512-bit state. 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. StriBob therefore has 6-round "Squarepants", as this is the best theoretical Square attack we know of [20].

3.3 Security Reduction Between StriBob 's π and Streebog's g

Only a single keyless permutation π is required in a In a sponge function. We utilize the LPS transform and twelve round constants C_i of GOST R 34.11-2012 in our new design. For some vector of twelve 512-bit subkeys C_i we define a 512-bit permutation $\pi_C(X_1) = X_{13}$ with iteration

$$X_{i+1} = \mathsf{LPS}(X_i \oplus C_i) \text{ for } 1 \leq i \leq 12.$$

Structure of π is shown in Figure 3.3. One may find it helpful to compare it with Figure 3.2 while considering the first input block which always has $N = h = 0$; the subkey values K_i are always the same, regardless of the input message block m. The chaining value h after processing the first block (but before final XORs) is $h = \pi_K(m)$, which is equivalent to π_C, just with different random round constants.

The output truncation after the last invocation of g of Streebog-256 indicates that collision resistance is expected of half of the output as well, which is exactly what we need in a $r = 256$ Sponge mode.

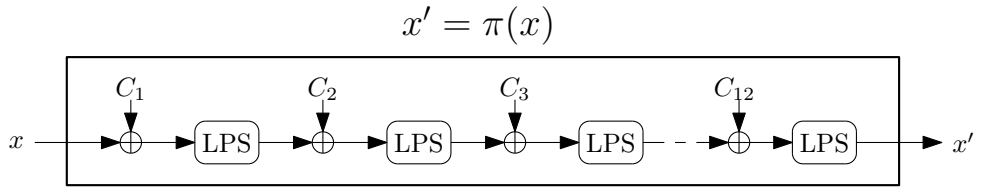

Figure 3.3: The 512-bit permutation π used by StriBob .

There's a straightforward security reduction from the indistinguishably of π_C to that of g. Note that the usual rules for the "distinguishing game" apply to our proof sketches.

Theorem 1. *If $\pi_C(x)$ can be effectively distinguished from a random permutation for any C, so can $g_N(h, x)$ for any h and N.*

Proof. If h is known, so are all of the subkeys K_i as those are a function of h alone. We have the equivalence

$$g_N(h, x) \oplus x \oplus h = \pi_K(x \oplus N).$$

Assuming that the round constants C_i offer no advantage over known round keys K_i, π_C is as secure as π_K and any distinguisher should have the same complexity. □

Theorem 1 indicates that a generic powerful attack against π is also an attack on g. A distinguishing attack against g of course does not imply a collision attack against Streebog as a whole. However as the security level expected of StriBob is lower than that of Streebog, Theorem 1 a significant level of confidence for our construction.

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 [1], Tree Hashes [8], Message Authentication Codes (MACs) [6], Authenticated Encryption (AE) algorithms [4], and pseudorandom extractors (PRFs and PRNGs) [2].

3.4.1 Absorbing and Squeezing

We recall the basic Sponge hash [1] 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:

$$S_{i+1} = \pi(\ S_i^r \oplus M_i \mid\mid S_i^c\ ). \tag{3.1}$$

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

$$\begin{aligned} H &= H \mid\mid S_i^r \\ S_{i+1} &= \pi(S_i). \end{aligned} \tag{3.2}$$

These constructions may be transformed into a keyed MAC by considering the state S_i as secret (keyed) [6]. 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 [4] 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 3.1. To encrypt plaintext blocks P_i to ciphertext blocks C_i we iterate:

$$\begin{aligned} C_i &= S_i^r \oplus P_i \\ S_{i+1} &= \pi(C_i \mid\mid S_i^c). \end{aligned} \tag{3.3}$$

The effect on the state is the same as that of Equation 3.1. The inverse -- decryption operation -- is almost equivalent to encryption, which in itself has significant implementation advantages:

$$P_i = S_i^r \oplus C_i$$

$$S_{i+1} = \pi(C_i \mid\mid S_i^c). \tag{3.4}$$

After encryption or decryption, a message authentication code for the message may be squeezed out as in Equation 3.2 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 3.4 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 as part of the CAESAR proposal.

3.4.4 Duplex, Triplex, Multiplex

The SpongeWrap [4] and monkeyDuplex [7] 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 [34].

The requirements laid out in [4] 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 [8] 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 [4] (Section 5.2 on Page 332) and [6] also hold for enc():

Theorem 2 (Theorem 1 from [4])**.** *The SpongeWrap and BLINKER authenticated encryption modes satisfy the following privacy and authentication security bounds:*

$$\mathrm{Adv}^{\mathsf{priv}}_{\mathsf{enc}}(\mathcal{A}) < q2^{-k} + \frac{N(N+1)}{2^{c+1}} \tag{3.5}$$

$$\mathrm{Adv}^{\mathsf{auth}}_{\mathsf{enc}}(\mathcal{A}) < q2^{-k} + 2^{-t} + \frac{N(N+1)}{2^{c+1}} \tag{3.6}$$

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

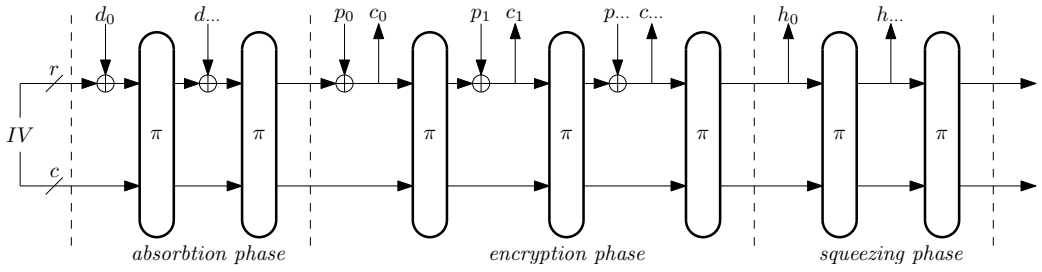

Figure 3.4: A simplified view of a Sponge-based AEAD. First the padded Secret Key, Nonce, and Associated Authenticated Data - all represented by d_u words - are "absorbed" or mixed into the Sponge state. The π permutation is then used to also encrypt data p_i into ciphertext c_i (or vice versa) and finally to "squeeze" out a Message Authentication Code h_i.

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 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 current this proposal.

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

$$S_{i+1} = \pi(\, S_i^r \oplus M_i \,||\, S_i^d \oplus D_i \,||\, S_i^{c'} \,). \tag{3.7}$$

For decryption we have the following update function:

$$S_{i+1} = \pi(\, C_i \,||\, S_i^d \oplus D_i \,||\, S_i^{c'} \,). \tag{3.8}$$

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 3.7 and 3.8). 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 [15, 33], *"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 2 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 [8]. 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 2 [4, 6] 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 Historical, Scientific, and Regulatory Context

Since January 1, 2013, the Russian Federation has mandated the use of new GOST R 34.11-2012 hash algorithm in digital signatures [14, 18]. This hash was designed apparently in response to cryptographic weaknesses reported in the previous hash standard GOST R 34.11-94 [17, 23].

The 2012 standard (which has been available since 2009), dubbed Streebog, has superficial similarities to the old 1994 standard but also features clearly AES-inspired design elements [12, 19, 25].

In contrast to the Russian approach, the U.S. NIST selected a novel Sponge-based design, Keccak, as the basis of future SHA-3 hash function standard [5, 11]. Sponge hashes diverge from more traditional Davies-Mayer [22] (SHA) and derived HAIFA [9] (Streebog) constructions in that they are based on a single keyless permutation π rather on a keyed permutation.

Sponge permutations can also be used to achieve provable Authenticated Encryption in straightforward manner (see Figure 3.4) [4, 7]. Here both the confidentiality and integrity of a message can be guaranteed with a single processing pass, without the use of a separate encryption algorithm such as GOST 28147-89 [16]. This has clear advantages for performance and implementation footprint, which are especially useful in limited-resource applications. Even full-featured secure communications suites can be constructed from a single permutation [29].

4.2 Advantages

The main technical advantages over AES-GCM are:

- Unlike AES-GCM, the StriBob authentication tags offer a level of integrity protection commensurate with its length (rather than half of it [28].)
- The round function is easily implementable on 64-bit platforms. The AES design is geared towards 32-bit platforms. With 12-round structure and double rate, the throughput of Hardware implementations can be expected to be even higher than AES.
- The same core can also be used for unkeyed hashing in digital signatures and other applications.

Also, selection of StriBob to the CAESAR portfolio would have additional advantages over AES-GCM in some security markets that are inclined to reject U.S. AES - based encryption monoculture:

- If StriBob gains community acceptance and CAESAR portfolio status, it would have potential for widespread practical use in some markets -- such as the Russian financial sector where the use of the GOST hash standard for digital signatures is mandatory anyway.
- The legally mandated status of GOST R 34.11-2012 in Russia means that efficient software and hardware implementations of the LPS core are bound to be available for multiple platforms. Applications that implement both algorithms can easily share resources.

4.3 Implementation Issues

Since the new construct requires 12 invocations of LPS per 256 bits processed in comparison to 25 invocations per 512 bits with Streebog, we see that the new construct is faster. Furthermore, the operational memory requirement is shrunk to approximately 25 % when compared to Streebog.

We use the following type to access the 512-bit state as both bytes and quadwords:

```
typedef union {
    uint8_t b[64];
    uint64_t q[8];
} sbob_w512;
```

4.3.1 Low-Resource Software Platforms

For a software implementation on a low-resource 8- or 16-bit CPUs and SoCs (e.g. RFID, Smart Card, Sensor, Ubiquitous / IoT category systems) it is advantageous to realize the linear layer L as a matrix multiplication in $\mathbb{F}_{2^8}$. Multiplication in a small finite field can be implemented via discrete logarithm and exponentiation tables: $AB = \exp(\log A + \log B)$. Note that Streebog and therefore Streebog uses a special bit-inverted representation for field elements [19].

One can combine the S-Box lookup and discrete logarithm table into a single 8×8 - bit lookup table $\log(S(x))$. The 8×8 matrix over $\mathbb{F}_{2^8}$ M (representing L) can be stored in log form. Required addition $x + y \pmod{2^8 - 1}$ can be implemented by adding carry bit $\lfloor \frac{x+y}{2^8} \rfloor$ of addition $x + y \pmod{2^8}$ to the 8-bit sum itself -- set $\exp(255) = \exp(0)$ in this case.

Here's the main routine of the 8-bit reference implementation `8bit/sbob_pi.c`:

```
const uint8_t sbob_pi8log[0x100] = { .. // log of bit-rev s-box
const uint8_t sbob_rv8exp[0x100] = { .. // bit-reversed exp table
const uint8_t sbob_matlog[8][8] = {  .. // logs of MDS matrix
const uint8_t sbob_rdcnst[12][64] = {.. // round constants

void sbob_pi(sbob_w512 *s512)
{
    int i, j, k, r;
    uint8_t x, y, t[64];

    for (r = 0; r < 12; r++) {

        for (i = 0; i < 64; i++) {
            t[(i >> 3) ^ ((i & 7) << 3)] =
                sbob_pi8log[s512->b[i] ^ sbob_rdcnst[r][i]];
            s512->b[i] = 0;
        }
        for (k = 0; k < 64; k += 8) {
            for (i = 0; i < 8; i++) {
                x = t[k + i];
                if (x > 0) {  // balance for side channel attacks!
                    for (j = 0; j < 8; j++) {
                        y = sbob_matlog[i][j] + x;
                        if (y >= x)  // side channel, balance!
                            y--;
                        s512->b[k + j] ^= sbob_rv8exp[y];
                    }
                }
            }
        }
    }
}
```

As the transpose P can be coded into the loops (switching the column and row indexes), the implementation of LPS requires a total of $256 + 256 + 8 \times 8 = 576$ bytes for storage. Unfortunately C_i round constants still require $12 \times 64 = 768$ bytes.

One may consider a variant that uses a fast pseudorandom generator such as some Fibonacci-based sequence or linear congruential generator instead of a truly random C to further compress the implementation. (see Section 5.2 for tweak opetion).

4.3.2 Medium- to High-Resource Software Platforms

A software implementation on system with a medium- or high-performance CPUs (e.g. server, desktop, laptop, or tablet category systems) can utilize $8 \times 8 \times 64$ - bit lookup tables that combine S and L, requiring a total of 16 kB and 768 B for round constants. The compression function code itself is very compact.

Here's code from our 64-bit reference implmementation `ref/sbob_pi64.c` and `ref/sbob_tab64.c`:

```
const uint64_t sbob_sl64[8][256] = { .. // s-box & l combined
const uint64_t sbob_rc64[12][8] = {  .. // round constants

// 64-bit version
void sbob_pi(sbob_w512 *s512)
{
    int i, r;
    sbob_w512 t;                        // temporary

    for (r = 0; r < 12; r++) {          // 12 rounds

        for (i = 0; i < 8; i++)         // t = x ^ rc
            t.q[i] = s512->q[i] ^ sbob_rc64[r][i];

        for (i = 0; i < 8; i++) {       // s-box and linear op
            s512->q[i] = sbob_sl64[0][t.b[i]] ^
                         sbob_sl64[1][t.b[i + 8]] ^
                         sbob_sl64[2][t.b[i + 16]] ^
                         sbob_sl64[3][t.b[i + 24]] ^
                         sbob_sl64[4][t.b[i + 32]] ^
                         sbob_sl64[5][t.b[i + 40]] ^
                         sbob_sl64[6][t.b[i + 48]] ^
                         sbob_sl64[7][t.b[i + 56]];
        }
    }
}
```

Results of wall-clock throughput measurements on a typical desktop system:

Algorithm	Throughput (MB/s)	Cycles / Byte
AES - 128/192/256	109.2 / 90.9 / 77.9	26.8 / 32.3 / 37.6
SHA - 256/512	212.7 / 328.3	13.8 / 8.92
GOST 28147-89	53.3	58.3
GOST R 34.11-1994	20.8	141
GOST R 34.11-2012	109.4	26.8
StriBob	115.7	25.3

These are wall-clock measurements. Measurements were made on a single core of an Intel Core i7 860 @ 2.80 GHz system running Ubuntu Linux 13.10 (amd64) with gcc 4.8.1. Linux reported internal clock frequency as 2.93 GHz during the tests. 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.

The AES, SHA, GOST 28147-89 and R 34.11-1994 timings with were measured with Ubuntu default OpenSSL (1.0.1e). A. Degtyarev's implementation (0.11) was used for the GOST 34.11-2012 benchmark. The StriBob reference implementation is by author.

Chapter 5

Design Rationale

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

5.1 Parameter and Component Selection

Regarding parameter selection:

- 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. [1, 2, 3, 4, 6, 8].
- Security relationship of Theorem 1 (Section 3.3) indicates that the number of rounds is appropriate and conservative as it is the same as is also used in Streebog.
- The BLNK padding variant as used in StriBob will allow flexible later extensions such as "Parallelized Tree AEAD" without breaking the core.

Specific notes regarding the LPS core:

- The LPS transform is a conservative choice as it can be analyzed with well-established tools developed for the AES round function. Increased size of the MDS Matrix guarantees efficient avalanche to cover the larger state size.
- The LPS transform of StriBob is equivalent to that of the Streebog standard, which has received years of analysis from the Russian cryptologic community and security authorities.
- LPS is a 512-bit transform and therefore perfectly suited for Sponge cryptography with given parameters. This is advantageous to, say, AES round transform that has been widened in some ad hoc fashion.

5.2 Hidden Weaknesses

The designer of StriBob has not hidden any weaknesses in this cipher.

The designers of Streebog have not published their detailed design criteria, at least not in English. We see that, with appropriate selection of constants C_i, one may hide at least one fixed point for the the π_C function, or a collision for the g function. In the context of StriBob , where the state is randomized by a secret key and a nonce, this should have a limited impact on security.

Tweak option. In order to save storage space on low-resource platforms, we reserve the option of switching from C_i to some on-the-fly generated pseudorandom sequence in future versions of StriBob . The use of large random incompressible tables was also one the most widely criticized design features of SHA-2 Family, making it relatively poorly suited for ultra-lightweight platforms.

Chapter 6

Intellectual Property and Consent

6.1 Intellectual Property

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

[2] 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.

[3] 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.

[4] 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.

[5] Bertoni, G., Daemen, J., Peeters, M., and Assche, G. V. The Keccak reference, version 3.0. NIST SHA3 Submission Document, January 2011.

[6] 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).

[7] 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`.

[8] 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.

[9] Biham, E., and Dunkelman, O. A framework for iterative hash functions - HAIFA. IACR ePrint 2007/278, `http://eprint.iacr.org/2007/278`, July 2007.

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

[11] Chang, S., R.Perlner, Burr, W. E., Turan, M. S., Kelsey, J. M., Paul, S., and Bassham, L. E. Third-round report of the SHA-3 cryptographic hash algorithm competition. Tech. Rep. NISTIR 7896, National Institute of Standards and Technology, November 2012.

[12] Daemen, J., and Rijmen, V. *The Design of Rijndael: AES -- the Advanced Encryption Standard*. Springer, 2002.

[13] Damgård, I. A design principle for hash functions. In *CRYPTO '89* (1989), G. Brassard, Ed., vol. 435 of *LNCS*, Springer, pp. 416--427.

[14] Dolmatov, V., and Degtyarev, A. *GOST R 34.11-2012: Hash Function*. No. RFC 6986. Internet Engineering Task Force, August 2013.

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

[16] GOST. *Cryptographic Protection for Data Processing System*. No. GOST 28147-89. Gosudarstvennyi Standard of USSR, 1989. (In Russian).

[17] GOST. *Cryptographic Protection of Information, Hash Function*. No. GOST R 34.11-94. Gosudarstvennyi Standard of Russian Federation, 1994. (In Russian).

[18] GOST. *Cryptographic Protection of Information, Hash Function*. No. GOST R 34.11-2012. Gosudarstvennyi Standard of Russian Federation, 2012. (In Russian).

[19] Kazymyrov, O., and Kazymyrova, V. Algebraic aspects of the russian hash standard GOST R 34.11-2012. In *Proc. CTCrypt '13, June 23--24, 2013, Ekaterinburg, Russia* (2013). IACR ePrint 2013/589 `http://eprint.iacr.org/2013/589`.

[20] Knudsen, L., and Wagner, D. Integral cryptanalysis (extended abstract). In *FSE 2002* (2002), J. Daemen and V. Rijmen, Eds., vol. 2365 of *LNCS*, Springer, pp. 112--127.

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

[22] Matyas, S., Meyer, C., and Ossas, J. Generating strong one-way functions with cryptographic algorithm. *IBM Technical Disclosure Bulletin*, 27 (1985), 5658--5659.

[23] Mendel, F., Pramstaller, N., Rechberger, C., Kontak, M., and Szmidt, J. Cryptanalysis of the GOST hash function. In *CRYPTO 2008* (2008), D. Wagner, Ed., vol. 5157 of *LNCS*, Springer, pp. 162--128.

[24] Merkle, R. *Secrecy, Authenticatication, and public key systems*. PhD thesis, Stanford University, 1979.

[25] NIST. *Advanced Encryption Standard (AES)*. No. FIPS-197. National Institute of Standards and Technology, 2001.

[26] NIST. Recommendation for block cipher modes of operation: Galois/counter mode (GCM) and GMAC. NIST Special Publication 800-38D, 2007.

[27] 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.

[28] Saarinen, M.-J. O. Cycling attacks on GCM, GHASH and other polynomial MACs and hashes. In *FSE 2012* (2012), vol. 7549 of *LNCS*, Springer, pp. 216--225.

[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. The CBEAMr1 authenticated encryption algorithm. CAESAR First Round Candidate, `http://www.cbeam.mx`, March 2014.

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

[33] 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.

[34] 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.