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
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Preface</strong></td>
<td>1</td>
</tr>
<tr>
<td>1 Specification</td>
<td></td>
</tr>
<tr>
<td>1.1 STRIBOBr1 Family and Parameters</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Structure of the $\pi$ Permutation</td>
<td>2</td>
</tr>
<tr>
<td>1.2.1 Matrix Notation and Finite Field Arithmetic</td>
<td>3</td>
</tr>
<tr>
<td>1.2.2 Round Constants $C_i$</td>
<td>4</td>
</tr>
<tr>
<td>1.2.3 Substitution $m' = S(m)$</td>
<td>5</td>
</tr>
<tr>
<td>1.2.4 Permutation $m' = P(m)$</td>
<td>5</td>
</tr>
<tr>
<td>1.2.5 Linear transform $m' = L(m)$</td>
<td>5</td>
</tr>
<tr>
<td>1.2.6 Example Computation of $\pi$</td>
<td>6</td>
</tr>
<tr>
<td>1.3 BLNK Sponge Mode and Padding</td>
<td>7</td>
</tr>
<tr>
<td>1.3.1 BLNK Block Operations</td>
<td>7</td>
</tr>
<tr>
<td>1.3.2 The CAESAR encrypt() and decrypt() AEAD API</td>
<td>8</td>
</tr>
<tr>
<td>1.4 Trace of strlib192r1 Computation</td>
<td>9</td>
</tr>
<tr>
<td>2 Security Goals</td>
<td></td>
</tr>
<tr>
<td>2.1 Specific Goals</td>
<td>10</td>
</tr>
<tr>
<td>2.2 Nonce Re-Use</td>
<td>10</td>
</tr>
<tr>
<td>2.3 General Goals</td>
<td>10</td>
</tr>
<tr>
<td>3 Security Analysis</td>
<td>11</td>
</tr>
<tr>
<td>3.1 Structure of GOST R 34.11-2012</td>
<td>11</td>
</tr>
<tr>
<td>3.1.1 SReebog Compression Function $g_N(h, m)$</td>
<td>11</td>
</tr>
<tr>
<td>3.2 Security of LPS Against Classical Attacks</td>
<td>12</td>
</tr>
<tr>
<td>3.3 Security Reduction Between StriBo's $\pi$ and SReebog's $g$</td>
<td>12</td>
</tr>
<tr>
<td>3.4 Sponge Functions</td>
<td>13</td>
</tr>
<tr>
<td>3.4.1 Absorbing and Squeezing</td>
<td>13</td>
</tr>
<tr>
<td>3.4.2 Duplexing</td>
<td>13</td>
</tr>
<tr>
<td>3.4.3 MAC-and-Continue</td>
<td>14</td>
</tr>
<tr>
<td>3.4.4 Duplex, Triplex, Multiplex</td>
<td>14</td>
</tr>
<tr>
<td>3.4.5 Multiplexing the Sponge</td>
<td>15</td>
</tr>
<tr>
<td>3.4.6 Domain Separation and Capacity Reduction</td>
<td>15</td>
</tr>
<tr>
<td>4 Features</td>
<td></td>
</tr>
<tr>
<td>4.1 Historical, Scientific, and Regulatory Context</td>
<td>16</td>
</tr>
<tr>
<td>4.2 Advantages</td>
<td>16</td>
</tr>
<tr>
<td>4.3 Implementation Issues</td>
<td>17</td>
</tr>
<tr>
<td>4.3.1 Low-Resource Software Platforms</td>
<td>17</td>
</tr>
<tr>
<td>4.3.2 Medium- to High-Resource Software Platforms</td>
<td>18</td>
</tr>
<tr>
<td>5 Design Rationale</td>
<td></td>
</tr>
<tr>
<td>5.1 Parameter and Component Selection</td>
<td>19</td>
</tr>
<tr>
<td>5.2 Hidden Weaknesses</td>
<td>19</td>
</tr>
</tbody>
</table>
6 Intellectual Property and Consent

6.1 Intellectual Property .......................................................... 20
6.2 Consent to CAESAR Selection Committee ............................ 20

Bibliography  ........................................................................... 21
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 \( \phi \) 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

STRIBO is an algorithm for Authenticated Encryption with Associated Data (AEAD). STRIBO accepts almost arbitrary ranges for its input parameters; however for CAESAR we propose a concrete parameter set “STRIBOBr192r1” as follows:

- Secret key size: 192 bits
- Secret sequence number: not used
- Public sequence number (nonce): 128 bits
- Authentication tag (message expansion): 128 bits

We first give a mathematical description of the cryptographic permutation \( S \) 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 \( \pi \) Permutation

STRIBO uses a \( 512 \times 512 \)-bit permutation \( \pi \) as its cryptographic foundation. \( \pi \) 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 STRIBO are lifted from the Russian GOST R 34.11-2012 “STREEBOC” 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 \( \pi(X_i) = X_{i+1} \) via:

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

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.](image-url)
1.2.1 Matrix Notation and Finite Field Arithmetic

We use C-style indexing for the 512-bit state as a matrix of $8 \times 8$ 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:

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

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

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

(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

StruBoG 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

\[ \text{val}(f) = \sum_{i=0}^7 2^{7-i} f_i. \]  

(1.3)

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

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 $t$-degree monomial canceling each out as this is a binary field:

\[ \left( \sum_{i=0}^7 f_i x^i \right) \left( \sum_{j=0}^7 g_j x^j \right) = \sum_{i=0}^7 \sum_{j=0}^7 f_i g_j x^{i+j} \pmod{2}. \]  

(1.4)

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

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

Example. Let’s multiply $\text{val}(f) = 45$ with $\text{val}(g) = 8E$. 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 $\mathbb{F}_2$:

\[ fg = x^{10} + x^{12} + 2x^{11} + x^{10} + x^9 + 2x^7 + x^6 + x^5 + x \]

\[ fg \equiv x^{13} + x^{12} + x^{10} + x^9 + x^6 + x \pmod{2} \]

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

\[ fg \text{ mod } p(x) = x^6 + x^3 + x^2 + x + 1 \]

\[ 45 \times 8E = \text{val}(x^6 + x^3 + x^2 + x + 1) = F2. \]

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

²This “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 Salsa specification, here given as byte matrices [14, 18]:

<table>
<thead>
<tr>
<th>$C_1$</th>
<th>$C_2$</th>
<th>$C_3$</th>
<th>$C_4$</th>
<th>$C_5$</th>
<th>$C_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1 08 5B DA 1E CA DA E9</td>
<td>6F A3 B5 8A A9 9D 2F 1A</td>
<td>EF 1F DF B3 E8 15 66 D2</td>
<td>4B EA 6B AC AD 47 47 99</td>
<td>AE 4F AE AE 1D 3A D3 D9</td>
<td></td>
</tr>
<tr>
<td>EB CB 2F B1 C0 65 7C 1F</td>
<td>4F E3 9D 46 0F 70 B5 D7</td>
<td>F9 48 E1 A0 5D 71 E4 DD</td>
<td>9A 3F 41 0C 6C A9 23 63</td>
<td>6F A4 C3 3B 7A 30 39 C0</td>
<td></td>
</tr>
<tr>
<td>2F 6A 76 43 2E 45 D0 16</td>
<td>F3 FE EA 72 0A 23 2B 98</td>
<td>61 D5 5E 0F 16 B5 01 31</td>
<td>9D 72 1C AD 68 5E 35 3F</td>
<td>2D 66 C4 F9 51 42 A4 6C</td>
<td></td>
</tr>
<tr>
<td>71 4E B8 BD 75 85 C4 FC</td>
<td>9A B5 17 6B 12 D6 99 58</td>
<td>5C B5 61 C2 DB 0A A7 CA</td>
<td>A9 D7 2C 82 ED 03 D6 75</td>
<td>1B 7F 64 C3 3B 7A 1E 6C</td>
<td></td>
</tr>
<tr>
<td>4B 7C E0 91 92 67 69 01</td>
<td>55 DD A2 1B D7 CB CD 56</td>
<td>D8 B7 13 33 93 52 03 BE</td>
<td>34 53 EA A1 93 E8 37 F1</td>
<td>7A 56 A2 7E A9 EA 63 F5</td>
<td></td>
</tr>
<tr>
<td>A2 42 2A 08 A4 60 D3 15</td>
<td>EE 79 04 70 21 B9 B7</td>
<td>60 17 58 FD 7C 6C FE 57</td>
<td>22 0C BE BC 84 E3 D1 2E</td>
<td>60 17 58 FD 7C 6C FE 57</td>
<td></td>
</tr>
<tr>
<td>05 76 74 36 CC 74 4D 23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DD 80 65 59 F2 A6 45 07</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$C_7$</th>
<th>$C_8$</th>
<th>$C_9$</th>
<th>$C_{10}$</th>
<th>$C_{11}$</th>
<th>$C_{12}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>F4 C7 OE 16 EE AA C5 EC</td>
<td>9B 1F 5B 42 D4 9D 93 C9 A7</td>
<td>37 8F 5A 54 16 31 22 9B</td>
<td>7B CD 9E D0 EF CB 89 FB</td>
<td>3E 1D 34 E4 32 29 8D 9F</td>
<td></td>
</tr>
<tr>
<td>51 AC 86 FE BF 24 09 54</td>
<td>03 E7 AA 02 0C 6E 41 41</td>
<td>94 4C 9A D8 EC 16 5F DE</td>
<td>3A 7D 3A 1B 25 89 42 24</td>
<td>3F 1B 2D 32 00 9D 0E 12</td>
<td></td>
</tr>
<tr>
<td>39 9E C6 C7 E6 BF 87 C9</td>
<td>0B 4E F7 F8 71 9C 36 DE 1E</td>
<td>3C D9 55 B7 E0 09 84</td>
<td>30 0A 44 0B DB BE B1</td>
<td>30 0A 44 0B DB BE B1</td>
<td></td>
</tr>
<tr>
<td>D3 47 3E 33 19 7A 93 C9</td>
<td>89 B4 44 3B 4D DB C4 9A</td>
<td>7B 2B 8A 9A 06 9C 54</td>
<td>7F B4 8A 9A 06 9C 54</td>
<td>7B 2B 8A 9A 06 9C 54</td>
<td></td>
</tr>
<tr>
<td>06 47 69 83 24 4A 05 04</td>
<td>F4 89 2B CB 92 98 06 90</td>
<td>69 D1 8D 2D B1 A5 C4 2F</td>
<td>36 AC A2 35 59 51 A8 D9</td>
<td>72 61 44 51 83 5A DB</td>
<td></td>
</tr>
<tr>
<td>35 17 45 4C A2 3C 4A F3</td>
<td>0E 38 DC 9C CB 1F 2A 60</td>
<td>A4 7F 0D D4 BF 02 E7 1E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>88 86 56 4D 3A 14 D4 93</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$C_{13}$</th>
<th>$C_{14}$</th>
<th>$C_{15}$</th>
<th>$C_{16}$</th>
<th>$C_{17}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1.2.3 Substitution \( m' = S(m) \)

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

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

(1.5)

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

| \( x \) | 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 | 2A | 68 | A2 | FD | 3A | CE | CC | 1F |
| 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 | 03 | 81 | 1C | 3C | 42 | 2A | FB | 2B |
| 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 | 03 | 29 | 18 | 6B | 04 | 80 | C3 | BD |
| 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 | 05 | 81 | 69 | D5 | 9S | 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 | DB | 81 | 61 |
| Ex    | 20 | 71 | 67 | A4 | 2D | 2B | 09 | 5B | CB | 98 | 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 \times 8 byte matrix. Transposition is a reflection along the main diagonal, which can also be seen as writing rows as columns:

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

(1.6)

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

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

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

\[
m' = m \cdot L, \quad L = \begin{pmatrix}
0E & 20 & FA & A7 & 2B & A0 & B4 & 70 \\
20 & D0 & 1B & 2A & 7A & 8E & 0F & 40 \\
7A & 40 & 1B & BA & 8E & 70 & 0F & 20 \\
40 & 20 & BA & DB & 0F & 1B & 7A & 40 \\
1B & BA & DB & 0F & 20 & 40 & 7A & 40 \\
7A & 8E & 0F & 20 & BA & DB & 0F & 7A \\
8E & 0F & 20 & BA & DB & 0F & 7A & 8E \\
0F & 20 & BA & DB & 0F & 7A & 8E & 0F
\end{pmatrix}
\]  

(1.7)

\[
m'[i][j] = \sum_{k=0}^{7} m[i][k] \cdot L[k][j].
\]  

(1.8)
1.2.6 Example Computation of $\pi$

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

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:

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

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

\[
S(X_1 \oplus C_1) = \begin{bmatrix}
45 & C4 & 72 & 53 & F1 & 89 & Ca & 6C \\
A4 & B3 & 5C & D6 & 88 & F3 & 19 & E9 \\
1F & 80 & 1B & 55 & ED & B5 & 1D & EE \\
91 & 12 & 60 & B0 & 91 & DC & A4 & A4 \\
6F & B7 & 33 & 46 & 55 & 22 & CC & EF \\
D6 & 6F & FC & 5A & 7D & 3A & 4B & ED \\
6A & AB & 46 & 6E & D1 & 34 & 80 & 93 \\
2B & A3 & 87 & 96 & ET & 50 & 80 & 06 \\
\end{bmatrix} \quad P(S(X_1 \oplus C_1)) = \begin{bmatrix}
45 & A4 & 1F & 91 & 6F & D6 & 6A & 2B \\
C4 & B3 & 80 & 1B & 76 & 6F & AB & A3 \\
72 & 5C & 10 & 60 & B3 & FC & 48 & 87 \\
53 & D6 & B5 & B0 & 46 & 5A & 6E & 96 \\
F1 & 88 & ED & 91 & 55 & D7 & 1E & E7 \\
89 & F3 & B5 & DC & 2C & 3A & 34 & 50 \\
CA & 19 & 1D & AA & CC & 4B & 80 & 80 \\
6C & EE & A4 & EF & ED & 93 & 06 \\
\end{bmatrix}
\]

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 $1 \times 8 + 3 = 11$. 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):

\[
m[1][3] = (C4 \ B3 \ 80 \ 12 \ B7 \ 6F \ AB \ A3) \cdot (A7 \ 80 \ 2A \ 5D \ F0 \ 88 \ 54 \ 6E)^T
\]

\[
= (C4 \ A7) \oplus (B3 \ 80) \oplus (80 \ 2A) \oplus (12 \ 5D) \oplus (B7 \ F0) \oplus (6F \ 88) \oplus (AB \ + \ 54) \oplus (A3 \ 6E)
\]

\[
= 66 \oplus B3 \oplus 2A \oplus 2D \oplus F4 \oplus D5 \oplus D2 \oplus 26
\]

\[
= 07.
\]

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

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

Further eleven rounds yields the final result of $\pi$:

\[
X_3 = \begin{bmatrix}
D3 & A7 & 1B & D2 & 2F & 79 & 0D & B7 \\
DB & 15 & 9B & 6A & 51 & EA & 8F & 54 \\
C9 & FA & F5 & 66 & DC & E6 & C3 & 4B \\
34 & 7A & 4E & 5D & DE & E8 & 58 \\
79 & 15 & 83 & B6 & 64 & 48 & DB & 68 \\
8E & 78 & 6F & 4A & 09 & B6 & 1A & D0 \\
AF & D7 & 5F & F3 & CC & B6 & 22 & D2 \\
44 & 23 & 4C & A0 & AD & E1 & B9 & C4 \\
\end{bmatrix} \quad \pi(X_1) = X_{13} = \begin{bmatrix}
16 & 8A & 86 & 7D & 30 & DB & 56 & 6D \\
57 & D5 & 30 & BE & D9 & 22 & 08 & 82 \\
37 & 0C & E2 & 79 & FB & A4 & E5 & 87 \\
A3 & 20 & E6 & ED & A2 & A3 & BA & 10 \\
17 & 34 & 62 & B6 & 23 & 0E & C5 & 67 \\
86 & 7C & 34 & 37 & 5E & 2E & 46 & D9 \\
A7 & FB & 06 & 19 & 27 & A3 & F5 & 49 \\
53 & 19 & BD & F9 & EC & 94 & 1A & 95 \\
\end{bmatrix}
\]
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. \( \pi \) 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.
- \( \delta \) Capacity consumed by padding. For STRIBOB192r1 we can bound this to \( < 2 \) 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 \( \text{pad} \), invoking the Sponge permutation \( \pi \) a total of \( \max(\lceil n/32 \rceil, 1) \) times.

The four basic operations are:

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

In the following generic pseudocode \( op \in \{ \text{put, get, enc, dec} \} \) and \( V[0 \cdots 63] \) is the state.

\[
\begin{align*}
1: & \quad i \leftarrow 0 \quad \text{state index, initialized to first byte} \\
2: & \quad \text{for } j = 0 \text{ to } n - 1 \text{ do} \\
3: & \quad \text{if } i = 32 \text{ then} \\
4: & \quad \quad V[32] \leftarrow V[32] \oplus \text{BLNK\_END} \oplus \text{pad} \quad \text{full block padding with block end marker} \\
5: & \quad \quad V \leftarrow \pi(V) \quad \text{cryptographic permutation} \\
6: & \quad \quad i \leftarrow 0 \quad \text{zero index} \\
7: & \quad \text{end if} \\
8: & \quad \text{if } op = \text{put then} \\
9: & \quad \quad V[i] \leftarrow V[i] \oplus D[j] \quad \text{XOR input data to the state} \\
10: & \quad \text{else if } op = \text{get then} \\
11: & \quad \quad D[j] \leftarrow V[i] \quad \text{simply save the data} \\
12: & \quad \text{else if } op = \text{enc then} \\
13: & \quad \quad C[j] \leftarrow V[i] \oplus P[j] \quad \text{encrypt as in a stream cipher} \\
14: & \quad \quad V[i] \leftarrow C[j] \quad \text{store ciphertext in state} \\
15: & \quad \text{else if } op = \text{dec then} \\
16: & \quad \quad P[j] \leftarrow V[i] \oplus C[j] \quad \text{decrypt as in a stream cipher} \\
17: & \quad \quad V[i] \leftarrow C[j] \quad \text{store ciphertext in state} \\
18: & \quad \text{end if} \\
19: & \quad i \leftarrow i + 1 \quad \text{advance block index} \\
20: & \quad \text{end for} \\
21: & \quad V[i] \leftarrow V[i] \oplus \text{BLNK\_END} \quad \text{end marker (note: } i = 32 \text{ possible)} \\
22: & \quad V[32] \leftarrow V[32] \oplus \text{BLNK\_FIN} \oplus \text{pad} \quad \text{final padding} \\
23: & \quad V \leftarrow \pi(V) \quad \text{final cryptographic permutation}
\end{align*}
\]
The confidential state can be cleared in order to minimize leakage.

performs full processing regardless of fail condition in order to minimize the risk of a timing attack. Also

F A I L

of these is used as a C-style zero-indexed byte vector in the descriptions that follow. Furthermore,

Input and output parameters to the encryption and decryption primitives are given below. Each one

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

V[ 0 ⋅⋅⋅ 63] is the 64-byte internal state of SrixBo.

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

A[a] Associated Authenticated data, 0 ≤ a bytes. Only integrity is protected for this data and the

P[n] Plaintext payload, 0 ≤ n bytes. Integrity and confidentiality is protected for this data.

C[n + 16] Ciphertext, 16 ≤ n + 16 bytes. Integrity and confidentiality is protected for this data.

Pseudocode for implementing standard AEAD API encryption:

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

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

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

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
1.4 Trace of strribob192r1 Computation

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

- $K[24] = "192-bit Secret Key value"
- $N[16] = "Nonces Used Once"
- $A[32] = "AAD Test Vector Exact Block 32 B"
- $P[38] = "This is a Test Vector for strribob192r1"

Steps 1-2: Keying. After zeroing V, the first input to $\pi$ 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
```

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

```
08 4D 08 20 FA 4A C8 C7 E0 0D CB 0A 3E 2F 5B 20 A2 77 05 7A 29 87 39 75
```

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
```

The state $V$ after nonce mixing $\pi$ is:

```
94 09 14 15 F3 64 F7 4B 22 A9 4A BE E7 31 14 D0 5F E3 BA 4B 1D 98 51
```

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 73 74 20 56 65 63 74 73 74 20
```

The state $V$ after AAD mixing $\pi$ 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
```

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

```
54 68 69 73 20 69 73 20 61 20 54 65 73 74 20 56 65 63 74 73 74 20 61 20
```

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
```

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

```
6D 3A 39 32 37 31 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

The state $V$ after encryption and final $\pi$:

```
16 5B D9 D6 2B 3C 7B 7D 6D C4 23 44 6B E7 60 82 BB 3A 5C A8 D2 50 C4
```

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
``
Chapter 2

Security Goals

2.1 Specific Goals

With the "strobob192r1" set of parameters (as specified in Sections 1.1 and 1.3) we have the following security claims and goals:

<table>
<thead>
<tr>
<th>Category</th>
<th>Effort</th>
<th>Attack Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confidentiality for the plaintext.</td>
<td>$2^{191}$</td>
<td>To recover to the plaintext from ciphertext or vice versa.</td>
</tr>
<tr>
<td>Integrity for the plaintext.</td>
<td>$2^{127}$</td>
<td>To forge plaintext payload.</td>
</tr>
<tr>
<td>Integrity for the associated data.</td>
<td>$2^{127}$</td>
<td>To forge Associated Data.</td>
</tr>
<tr>
<td>Integrity for the public message number.</td>
<td>$2^{127}$</td>
<td>To forge public message number.</td>
</tr>
</tbody>
</table>

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 $\pi$ permutation.

2.2 Nonce Re-Use

StriboBos 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. StriboBos 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 $C = \text{encrypt}(K, N, A, P)$ 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 $\text{decrypt}(K, N, A, C) = \text{FAIL}$ 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

StriBoo is based on the new Russian GOST R 34.11-2012 “StriiBoo” hash standard [18] and a variant of Saarinen’s Blinker padding for the Sponge Authenticated Encryption construct of [4, 29].

StriiBoo 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. StriiBoo 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 \( M = m_0 | m_1 | \cdots | m_n \) 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 \( \epsilon \), which is computed over all input blocks mod \( 2^{512} \).

3.1.1 StriiBoo 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 performs a total of 25 invocations of LPS:

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

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

\[
\sum_{i=0}^{n} m_i \pmod{2^{512}}
\]

Figure 3.1: Operation of StriiBoo 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 $P = \frac{256}{128}$. 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 $64 \times 64$ 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. Streebog therefore has 6-round “Squarepants”, as this is the best theoretical Square attack we know of [20].

3.3 Security Reduction Between StriBob’s $\pi$ and Streebog’s $g$

Only a single keyless permutation $\pi$ 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 $\pi_C(X_i) = X_{i3}$ with iteration

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

Structure of $\pi$ 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 $\pi_C$, 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 $\pi$ used by StriBob.
There’s a straightforward security reduction from the indistinguishably of $\pi_C$ 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 $K_i$, $\pi_C$ is as secure as $\pi_K$ and any distinguisher should have the same complexity.

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

### 3.4 Sponge Functions

Sponge constructions generally consist of a state $S = (S^r || S^c)$ which has $b = r + c$ bits and a $b$-bit keyless cryptographic permutation $\pi$. 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:

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

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

$$H = H || S_i^c$$

$$S_{i+1} = \pi(S_i).$$

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:

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

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

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:

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

\[
\text{Adv}^{\text{priv}}_{\text{enc}}(A) < q2^{-k} + \frac{N(N + 1)}{2^{c+1}} \quad (3.5)
\]

\[
\text{Adv}^{\text{auth}}_{\text{enc}}(A) < q2^{-k} + 2^{-t} + \frac{N(N + 1)}{2^{c+1}} \quad (3.6)
\]

against any single adversary A if \( K \leftarrow \{0, 1\}^k \), tags of \( l \geq t \) bits are used, \( \pi \) is a randomly chosen permutation, \( q \) is the number of queries and \( N \) is the number of times \( \pi \) 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
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:

$$S_{i+1} = \pi(S_i^c \oplus M_i \parallel S_i^d \oplus D_i \parallel S_i^{c'}) \quad \text{(3.7)}$$

For decryption we have the following update function:

$$S_{i+1} = \pi(C_i \parallel S_i^d \oplus D_i \parallel S_i^{c'}) \quad \text{(3.8)}$$

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 Skein, 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] (Skein) constructions in that they are based on a single keyless permutation rather than 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 Skein 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 Skein 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 Skein 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 Striebeckog, we see that the new construct is faster. Furthermore, the operational memory requirement is shrunk to approximately 25% when compared to Striebeckog.

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

```c
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 Striebeckog and therefore Striebeckog uses a special bit-inverted representation for field elements [19].

One can combine the S-Box lookup and discrete logarithm table into a single \( 8 \times 8 \) - bit lookup table \( \log(S(x)) \). The \( 8 \times 8 \) matrix over \( \mathbb{F}_{2^8} \) \( M \) (representing \( L \)) can be stored in log form. Required addition \( x + y \mod 2^8 \) can be implemented by adding carry bit \( \lfloor \frac{x+y}{2^8} \rfloor \) of addition \( x + y \mod 2^8 \) 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:

```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 option).

### 4.3.2 Medium- to High-Resource Software Platforms

A software implementation on systems with 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 implementation ref/sbob_pi64.c and ref/sbob_tab64.c:

```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:

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

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 SrixBos reference implementation is by author.
Chapter 5

Design Rationale

Serglass 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 Glue.

- The BLNK padding variant as used in Glass 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 Glass is equivalent to that of the Glue 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 Glass has not hidden any weaknesses in this cipher.

The designers of Glue 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 $\pi_C$ function, or a collision for the $g$ function. In the context of Glass, 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 Glass. The use of large random incompressible tables was also one of 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


