The Secret Structure of the S-Box of Streebog, Kuznechik and Stribob

Alex Biryukov, Léo Perrin, Aleksei Udovenko
first.last-name@uni.lu

SnT, University of Luxembourg

August 27, 2015

Abstract

The last hash function and block cipher standardized by the Russian standardization body (GOST) both use the same S-Box. It is also used by an independent CAESAR candidate. This transformation is only specified as a look up table and the reason behind its choice is unknown.

We managed to reverse-engineer this S-Box and describe its unpublished structure. Our decomposition allows a much more efficient hardware implementation but the choice of the components used is puzzling from a cryptographic perspective.

This extended abstract does not explain *how* we found this decomposition. We will describe our process in an extended version of this paper.

1 Our Target

We consider the permutation of 8 bits given in Appendix A and denote it π. This S-Box is used by three distinct algorithms.

Streebog Sometimes spelled Stribog, it is the new standard hash function for the Russian Federation [Fed12]. Several cryptanalyses against this algorithm have been published. In particular, it is actually possible to find collisions for a modified version of this hash function where only the round constants are changed [AY15]. To show that the constants were not chosen with malicious intentions, the designers published a note describing how they were derived from a modified version of the hash function [Rud15]. While puzzling looking at first glance, the seeds used actually correspond to Russian names written backward and endoded in `cp1251`.

Kuznyechik Sometimes spelled Kuznechik, it is the last block cipher standardized by the GOST. It was first mentionned in [SDL+14] and is now available at [Fed15].

STRIBOBr1 The CAESAR candidate [Saa14] STRIBOB made it to the second round of the competition. Its designer is not affiliated with the GOST but STRIBOBr1, the the first round version, is based on the permutation used in Streebog.[1]

Using the statistical approach described in [BP15], we could rule out that this S-Box was generated at random. Indeed, the probability for a random permutation to have at most 15 occurrences of 8 in its difference distribution table (which is the case for π) is

[1] The second round candidate STRIBOBr2 does *not* use this S-Box because of the secrecy of its design process (among other things).

equal to $2^{-82.7}$. Thus, this S-Box was neither generated at random nor picked from a feasibly large pool of random S-Boxes according to some criteria.

2 A Secret Structure

Very little is known about the criteria used to choose or build this S-Box. In [SB14], Saarinen et al. summarize a discussion they had with some of the designers of the GOST algorithms at conference in Moscow:

> We had brief informal discussions with some members of the Streebog and Kuznyechik design team [...] the aim was to choose a "randomized" S-Box that meets the basic differential, linear, and algebraic requirements. Randomization was simply iterated until a "good enough" permutation was found. This was seen as an effective countermeasure against yet-unknown attacks [as well as algebraic attacks].

Actually, this S-Box was built using a secret structure which we managed to reverse-engineer. Figure 1 shows how to compute π and its components are described below.

Finite field multiplication "$\odot$" denotes a multiplication in the finite field $\mathrm{GF}(2^4)$ defined by the irreducible polynomial $x^4 + x^3 + 1$.

8-bit linear permutations The linear functions correspond to multiplications by the following binary matrices:

$$\alpha = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \ \omega = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

4-bit non-linear functions The non-linear functions used are all given in Table 1. Note that $\mathcal{I}$ is the inversion in the finite field used for the multiplications and that ϕ is not a bijection.

Multiplexer The outputs of ν_0 and ν_1 go in a multiplexer. If the value on the right branch at that moment is equal to 0 then the output of ν_0 is selected. Otherwise, this component returns the output of ν_1.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$\mathcal{I}$	0	1	c	8	6	f	4	e	3	d	b	a	2	9	7	5
ν_0	2	5	3	b	6	9	e	a	0	4	f	1	8	d	c	7
ν_1	7	6	c	9	0	f	8	1	4	5	b	e	d	2	3	a
ϕ	b	2	b	8	c	4	1	c	6	3	5	8	e	3	6	b
σ	c	d	0	4	8	b	a	e	3	9	5	2	f	1	6	7

Table 1: The non-linear functions needed to compute π.

We also provide a SAGE [S+13] script performing those computations in Appendix B.

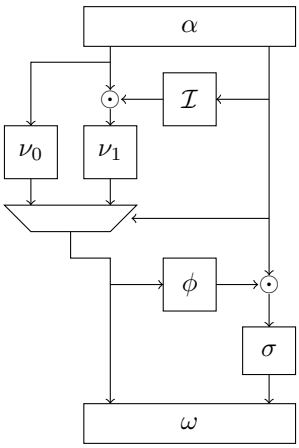

Figure 1: Our decomposition of π. The "wires" transmit 4 bit.

3 Preliminary Comments on our Decomposition

To the best of our knowledge, it is the first time that finite field multiplications are used to build an S-Box. The only exceptions are the compact implementations of S-Boxes based on the inverse function like in the AES [DR02, Can05]. However, this S-Box cannot correspond to a monomial because if it were the case then each line of its difference distribution table would be a permutation of the first one (see [BCC10]). It is not the case.

An obvious issue with this structure is that multiplication by 0 is not invertible. The designers of π tackled in a different way for each multiplication. First, a different datapath is used when a multiplication by 0 would occur. In the second multiplication, the problem is prevented by using a function which has no pre-image for 0.

Table 2 summarizes the properties of the non-linear components of our decomposition. While it is not hard to find 4-bit permutations with a differential uniformity of 4, we see that none of the components chosen do except the inverse function. We can thus discard the idea that the strength of π against differential and linear attacks relies on the individual resilience of each of its components.

	1-to-1	Best differentials and their probabilities	Best linear approximations and their probabilities
ϕ	No	$\mathtt{1} \to \mathtt{d}$ (8/16)	$\mathtt{3} \to \mathtt{8}$ (2/16), $\mathtt{7} \to \mathtt{d}$ (2/16)
σ	Yes	$\mathtt{f} \to \mathtt{b}$ (6/16)	$\mathtt{1} \to \mathtt{f}$ (14/16)
ν_0	Yes	$\mathtt{6} \to \mathtt{c}$ (6/16), $\mathtt{e} \to \mathtt{e}$ (6/16)	30 approximations $(8 \pm 4)/16$
ν_1	Yes	$\mathtt{9} \to \mathtt{2}$ (16/16)	8 approximations $(8 \pm 6)/16$
$\mathcal{I}$	Yes	15 differentials (4/16)	30 approximations $(8 \pm 4)/16$

Table 2: Linear and differential properties of the components of π.

Furthermore, we note that there exists a probability 1 differential in ν_1: $\nu_1(x \oplus \mathtt{9}) \oplus \nu_1(x) = \mathtt{2}$. Besides, a difference equal to $\mathtt{2}$ on the left branch corresponds to a 1 bit difference on bit 5 of the input of ω, a bit which is left unchanged by this transformation.

We also simulated the implementation of π in hardware.[2] We used two different

[2] We used `Synopsys design_compiler` (version J-2014.09-SP2) along with digital library `SAED_EDK90_CORE` (version 1.11).

definitions of π: the look up table given by the designers and our decomposition. Table 3 contains both the area taken by our implementations and the delay, i.e. the time taken to compute the output of the S-Box. For both quantities, the lower is the better. As we can see, the knowledge of the decomposition allows us to divide the delay by 8 and the area by 2.5.

Structure	Area (μm^2)	Delay (ns)
naive implementation	3889.6	362.52
using the decomposition	1530.1	46.11

Table 3: Results on the hardware implementation of π.

4 Conclusion

The S-Box used by the last two Russian standards in symmetric cryptography has a hidden structure which we managed to completely recover. The knowledge of this decomposition gives us a significantly more efficient hardware implementation. However, it is based on sub-components whose lack of cryprographic strength is puzzling.

A future paper will provide detailed explanations on the process we used to reverse-engineer this S-Box.

5 Acknowledgement

We thank Yann Le Corre for studying the hardware implementation of the S-Box. We also thank Oleksandr Kazymyrov for suggesting this target. The work of Léo Perrin is supported by the CORE ACRYPT project (ID C12-15-4009992) funded by the *Fonds National de la Recherche* (Luxembourg). The work of Aleksei Udovenko is supported by the *Fonds National de la Recherche*, Luxembourg (project reference 9037104).

References

[AY15] Riham AlTawy and Amr M Youssef. Watch your constants: Malicious streebog. *IET Information Security*, 2015.

[BCC10] Céline Blondeau, Anne Canteaut, and Pascale Charpin. Differential properties of power functions. *International Journal of Information and Coding Theory*, 1(2):149–170, 2010.

[BP15] Alex Biryukov and Léo Perrin. On Reverse-Engineering S-Boxes with Hidden Design Criteria or Structure. In *Advances in Cryptology – CRYPTO 2015*, Lecture Notes in Computer Science, page (to appear). Springer Berlin Heidelberg, 2015.

[Can05] D. Canright. A very compact s-box for aes. In JosyulaR. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 441–455. Springer Berlin Heidelberg, 2005.

[DR02] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer, 2002.

[Fed12] Federal Agency on Technical Regulation and Metrology (GOST). Gost r 34.11-2012: Streebog hash function, 2012. https://www.streebog.net/.

[Fed15] Federal Agency on Technical Regulation and Metrology (GOST). Block ciphers, 2015. http://www.tc26.ru/en/standard/draft/ENG_GOST_R_bsh.pdf.

[Rud15] V. Rudskoy. Note on streebog constants origin, 2015. http://www.tc26.ru/en/ISO_IEC/streebog/streebog_constants_eng.pdf.

[S+13] W. A. Stein et al. *Sage Mathematics Software (Version 5.10)*. The Sage Development Team, 2013. http://www.sagemath.org.

[Saa14] Markku-Juhani O Saarinen. Stribob: Authenticated encryption from gost r 34.11-2012 lps permutation. *IACR Cryptology ePrint Archive*, 2014:271, 2014.

[SB14] Markku-Juhani O. Saarinen and Billy Bob Brumley. Whirlbob, the whirlpool variant of stribob. Cryptology ePrint Archive, Report 2014/501, 2014. http://eprint.iacr.org/.

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

A Definition of the S-Box

Table 4 contains the definition of π.

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

Table 4: The S-Box in hexadecimal. For example, $\pi(0x7a) = 0xc6$.

B Generating π from our Decomposition

```
from sage.all import *

X = GF(2).polynomial_ring().gen()
F = GF(2**4, name="a", modulus=X**4+X**3+1)

inv   = [0x0,0x1,0xc,0x8,0x6,0xf,0x4,0xe,0x3,0xd,0xb,0xa,0x2,0x9,0x7,0x5]
nu_0  = [0x2,0x5,0x3,0xb,0x6,0x9,0xe,0xa,0x0,0x4,0xf,0x1,0x8,0xd,0xc,0x7]
nu_1  = [0x7,0x6,0xc,0x9,0x0,0xf,0x8,0x1,0x4,0x5,0xb,0xe,0xd,0x2,0x3,0xa]
sigma = [0xc,0xd,0x0,0x4,0x8,0xb,0xa,0xe,0x3,0x9,0x5,0x2,0xf,0x1,0x6,0x7]
phi   = [0xb,0x2,0xb,0x8,0xc,0x4,0x1,0xc,0x6,0x3,0x5,0x8,0xe,0x3,0x6,0xb]

alpha = Matrix(GF(2), 8, 8, [
    0,0,0,0,1,0,0,0,  0,1,0,0,0,0,0,1,
    0,1,0,0,0,0,1,1,  1,1,1,0,1,1,1,1,
    1,0,0,0,1,0,1,0,  0,1,0,0,0,1,0,0,
    0,0,0,1,1,0,1,0,  0,0,1,0,0,0,0,0,
])
omega = Matrix(GF(2), 8, 8, [
    0,0,0,0,1,0,1,0,  0,0,0,0,0,1,0,0,
    0,0,1,0,0,0,0,0,  1,0,0,1,1,0,1,0,
    0,0,0,0,1,0,0,0,  0,1,0,0,0,1,0,0,
    1,0,0,0,0,0,1,0,  0,0,0,0,0,0,0,1,
])

def applymat8(x, mat):
    y = mat * Matrix(GF(2), 8, 1, map(int, bin(x)[2:].zfill(8)))
    return int("".join(map(str, y.T[0][:8])), 2)

def F_mult(x, y):
    return (F.fetch_int(x) * F.fetch_int(y)).integer_representation()

pi = []
for x in xrange(256):
    x = applymat8(x, alpha)
    l, r = x >> 4, x & 0xf
    l = (r == 0) * nu_0[l] + (r != 0) * nu_1[F_mult(l, inv[r])]
    r = sigma[F_mult(r, phi[l])]
    x = applymat8((l << 4) | r, omega)
    pi.append(x)
print pi
```