This is a previous version of the article published in Logic Journal of the IGPL. 2016, 24(1): 68-79. doi:10.1093/jigpal/jzv044

Randomness Analysis and Generation of Key-Derived S-Boxes*

Rafael Álvarez and Antonio Zamora

Dpt. of Computer Science and Artificial Intelligence (DCCIA)
University of Alicante (Campus de San Vicente)
Ap. 99, E-03080, Alicante, Spain
{ralvarez,zamora}@dccia.ua.es

Abstract. Although many ciphers use fixed, close to ideal, s-boxes (like AES for example), random s-boxes offer an interesting alternative since they have no underlying structure that can be exploited in cryptanalysis. For this reason, some cryptosystems generate pseudo-random s-boxes as a function of the key (key-derived).
We analyse the randomness properties of key-derived s-boxes generated by some popular cryptosystems like the RC4 stream cipher, and the Blowfish and Twofish block ciphers with the aim of establishing if this kind of s-boxes are indistinguishable from purely random s-boxes.
For this purpose we have developed a custom software framework to generate and evaluate random and key derived s-boxes.
We also detail and analyse several mechanisms for the generation of proper key-derived s-boxes, including fixed point filtering and different sizes based on 8x8 s-boxes.

Keywords: S-Boxes, Key-Derived, Random, RC4, Blowfish, Twofish

1 Introduction

Substitution boxes (or s-boxes) are simple substitution tables where an input value is transformed into a different output value. They are employed in many sizes, but the most prevalent are 8x8 bits (byte as input and output) and 8x32 bits (byte as input and four byte word as output). They are essential in many cryptosystem designs (see [2, 3, 8, 14, 23, 24]) since they can introduce the required non-linearity characteristics, making cryptanalysis a more difficult endeavour (see [11]).

Regarding their design, there are two basic schools of thought:

- **Crafted s-boxes**. These are carefully designed to achieve certain desirable characteristics and often have an underlying generator function. The advantage is they can be chosen so they are optimum in terms of non-linearity, avalanche, bit independence, etc. On the other hand, the fact that they are fixed or based on some kind of underlying structure could make cryptanalysis easier (see [8, 9, 12, 13]).

* Partially supported by grant TIN2011-25452 (TUERI)

– **Random or key-derived s-boxes**. Unlike crafted s-boxes, random s-boxes cannot be guaranteed to have certain values of non-linearity or other metrics but many cryptographers think that the fact they do not have any exploitable underlying structure is a distinct advantage against cryptanalysis. Furthermore, pseudo-random s-boxes that are generated as a function of the key (key-derived s-boxes) allow the cryptosystem to use a different s-box per each key, possibly making cryptanalysis even more difficult.
  In general, random s-boxes offer acceptable security characteristics although not as high as the best crafted s-boxes (see [4]).

How random are key-derived s-boxes is, therefore, an interesting question to ask. Ideally, they should approximate purely random s-boxes as much as possible so that they have the advantage of being free of underlying structure while at the same time permitting the cryptosystem to use a different s-box for each key.

We have developed a testing framework capable of generating and analysing 8x8 s-boxes and we have tested 340,000 different s-boxes corresponding to random s-boxes, those generated by the RC4 stream cipher (see [14]), the Blowfish block cipher (see [23]) and its improved sibling, Twofish (see [24]). We have computed useful metrics for each s-box and summarized the resulting data to extract meaningful conclusions.

Since key-derived s-boxes have a great number of applications as a non-linear element in cryptographic primitives, it is also interesting to consider some aspects regarding their generation process in order to achieve satisfactory results.

The rest of the paper is organised as follows: a description of the tests performed is given in section 2, the results are analysed in section 3, some concepts and algorithms for key-derived s-box generation are described in section 4 and, finally, the conclusions are in section 5.

2 Description

In order to test the randomness of key-derived s-boxes we have compared several algorithms against random s-boxes. During our study, we have generated 10,000 random s-boxes, 10,000 RC4 s-boxes, 40,000 8x32 Blowfish s-boxes (separated into 160,000 8x8 s-boxes) and 40,000 Twofish s-boxes (also separated into 160,000 8x8 s-boxes); totalling 340,000 tested s-boxes.

We have developed a custom testing framework capable of generating and analysing s-boxes, computing multiple metrics for each s-box. It is comprised of two separate components: a completely new element written in Go (see [10]) that generates s-boxes in a parallel, concurrent, fashion using hooks into the cryptosystems to obtain the required s-boxes and other custom generators; and an element written in C (evolved from previous work, see [4]) that batch tests groups of s-boxes and is simultaneously executed in as many cores as possible.

Then, the results have been statistically analysed to find meaningful distinguishing characteristics in each set.

2.1 Random S-Boxes

Random s-boxes have been generated as bijective 8x8 s-boxes using random permutations of the values $0, 1, \ldots, 255$. This guarantees perfect balance but can introduce fixed points. If fixed points are not desired, then some kind of filtering must be introduced to eliminate the offending s-boxes or transform them into valid ones.

These random s-boxes form a standard of randomness that other key-derived s-boxes can be compared to.

2.2 RC4

The RC4 stream cipher algorithm (see [14, 19]) is one of the most widely used software stream ciphers since it is part of the Secure Sockets Layer (SSL, see [7]), Transport Layer Security (TLS, see [5]) and Wired Equivalent Privacy (WEP, see [6]) protocols, among others. It does not employ a s-box per-se, but its internal state consists of an 8x8 bijective table that dynamically changes as data is encrypted. The key used with RC4 solely determines the initial state of this internal s-box, so we consider that initial state as a key-derived s-box.

2.3 Blowfish

Blowfish (see [23]) is a popular block cipher that, due to its slower key schedule algorithm, is commonly used as a password hashing algorithm in operating systems and applications (see [18]). It uses four 8x32 key-derived s-boxes that are generated during the key schedule phase.

Since some of the metrics require extensive calculations with a computational complexity on the order of $O(n) = 2^b$, being b the output bit size, they are too big to be analysed natively. For this reason we consider each 8x32 s-box as 4 adjacent 8x8 s-boxes, obtaining 16 8x8 s-boxes per each different key.

2.4 Twofish

Twofish (see [24]) is the successor to Blowfish and was one of the five finalists of the Advanced Encryption Standard (AES) contest. It also generates four 8x32 s-boxes during the key schedule algorithm, which have been considered as sixteen 8x8 s-boxes so they can be tested for all metrics.

3 Results

3.1 Balance

Balance can be defined in terms of just the component functions (columns) of the s-box, or of all the linear combinations of the component functions. The latter definition is probably more correct, since unbalanced linear combinations

Table 1. Balance test results

	Random	RC4	Blowfish	Twofish
Min.	255	255	1	255
Max.	255	255	39	255
Med.	255	255	13	255
Inv.	0%	0%	100%	0%

imply also some type of statistical bias towards 0 or 1 in the output. See [4, 15, 25] for more information.

When bijective s-boxes are used, then measuring balance can act as a way of checking that they are really bijective. In those cases where unbalanced s-boxes may be used, the ratio of balanced linear combinations to total number of linear combinations is an useful metric.

As shown in table 1, we can see that, depending on the algorithm, they are either all valid (all s-boxes have all linear combinations balanced) or all invalid (no s-box has all linear combinations balanced). This table includes the minimum (Min.), maximum (Max.) and median (Med.) number of balanced linear combinations found in an s-box of that set. It also includes the percentage of unbalanced s-boxes (Inv.) in each set.

The random s-boxes, and the ones generated by RC4, are all balanced since they are permutations of the values 0 to 255 (bijective).

Blowfish, on the other hand, fails the balance test since, once you subdivide each 8x32 s-box into four 8x8 adjacent s-boxes, they do not contain all possible values achieving terrible results in this test with a median of 13 and a minimum of 1 or 2 balanced linear combinations (so, in most cases, not even the component functions are balanced). This is a problem, since you can always consider an 8x32 s-box as four adjacent 8x8 s-boxes and attack those accordingly.

Twofish is, certainly, an improvement in this regard, generating 8x32 s-boxes that are balanced when separated as 4 distinct 8x8 s-boxes.

3.2 Fixed Points

Direct ($S(i) = i$) or reverse ($S(i) = 255 - i$) fixed points are generally not desirable in s-boxes since they imply that the output is equivalent to the input and not modified in some cases (see [4]).

Fixed points are not an absolute requirement since there is no known attack exploiting this characteristic, although it is generally desirable that all input bytes are transformed into something different by the s-box.

As shown in table 2, none of the studied algorithms prevent fixed points from happening. The occurrence rate is of 1 direct and 1 reverse fixed point per s-box on average, which is on par with random s-boxes. This table includes the minimum (Min.), maximum (Max.) and median (Med.) number of direct and reverse fixed points found in an s-box of that set. It also includes the percentage of invalid s-boxes (Inv. – having fixed points) in each set.

Table 2. Fixed points test results

Direct

	Random	RC4	Blowfish	Twofish
Min.	0	0	0	0
Max.	7	8	8	7
Med.	1	1	1	1
Inv.	63%	57%	64%	63%

Reverse

	Random	RC4	Blowfish	Twofish
Min.	0	0	0	0
Max.	7	6	8	7
Med.	1	1	1	1
Inv.	64%	62%	64%	63%

3.3 Non-linearity

Table 3. Non-linearity test results

	Random	RC4	Blowfish	Twofish[1]	Twofish[2]
Min.	78	78	77	78	80
Max.	98	98	98	98	98
Med.	92	92	93	92	94
Inv.	19%	19%	41%	19%	50%

Non-linearity (see [4, 15]) is a measure of resistance to linear cryptanalysis and is defined for Boolean functions. In the case of s-boxes we take the minimum non-linearity of all linear combinations of the component functions of each s-box. Although the theoretical minimum is 0, we consider that a better minimum threshold is 2^{n-2} because, generally, random s-boxes are above that value. The maximum non-linearity considered is $2^{n-1} - 2^{\frac{n}{2}}$.

As can be seen in table 3, all algorithms present similar values to random s-boxes in minimum (Min.) and maximum (Max.) non-linearity values ranging from 78 to 98 approximately.

The median value (Med.) shows some interesting characteristics with random and RC4 s-boxes having a median of 92, Blowfish a median of 93 and Twofish 92 for the first half of s-boxes (denoted by Twofish[1]) and 94 for the second half (Twofish[2]). These could be used, to a certain degree, as a distinguishing factor for Blowfish or second half Twofish s-boxes. The table also includes the percentage of not-as-good, or *invalid*, s-boxes (Inv.) that have non-linearity values below the median value.

Please note that the maximum non-linearity considered would be $2^{n-1} - 2^{n/2}$, or 112 for $n = 8$. The AES s-box achieves a non-linearity of 112 (see table 7).

3.4 XOR Table

Table 4. XOR Table test results

	Random	RC4	Blowfish	Twofish
Min.	8	8	8	8
Max.	16	18	18	18
Med.	12	12	12	12
Valid	91%	91%	91%	91%

The XOR table is a metric related to differential cryptanalysis (see [1, 16]). There are two possible metrics for the XOR table: the first corresponds to the number of valid entries (entries with values 0 or 2) of the XOR table, with the second being the maximum entry in the table. See [4, 15, 25] for more information.

As shown in table 4, there are no differentiating factors among the algorithms tested; they all present a minimum value (Min.) of the maximum entry of 8, a maximum value (Max.) of the maximum entry of 18 (except 16 for the random s-boxes) and a median value (Med.) of the maximum entry of 12. The number of valid entries in the table is also the same, stable at 91%.

3.5 Avalanche

Table 5. Avalanche (*DSAC(1)*) test results

	Random	RC4	Blowfish	Twofish
Min.	12	10	10	10
Max.	28	28	29	30
Med.	16	16	17	16

Defined for Boolean functions, we consider the distance to the strict avalanche criterion of order 1 or DSAC(1). The DSAC(1) for the complete s-box is the maximum of the distances of the component functions (columns) (see [4, 15]). The lower boundary is determined as 2^{n-2}.

As shown in table 5, all algorithms present similar minimum (Min.) and maximum (Max.) values, while Blowfish presents a slightly higher median (Med.) value.

Table 6. Bit Independence (*DBIC(2,1)*) test results

	Random	RC4	Blowfish	Twofish
Min.	16	16	16	16
Max.	30	32	36	30
Med.	20	20	20	20

3.6 Bit Independence

Defined for the whole S-box, this corresponds to bit independence DBIC(2,1) (see [4, 15]).

This metric presents no differentiating factors among the tested algorithms, as shown in table 6 that includes minimum (Min.), maximum (Max.) and median (Med.) values.

3.7 Comparison with AES

Table 7. Comparison between AES and random s-boxes

	AES	Random (best)	Random (average)	Random (worst)
Balance	100%	100%	100%	100%
Fixed points	(0,0)	(0,0)	(1,1)	(7,7)
Nonlinearity	112	98	92	78
XOR table	4	8	12	16
DSAC(1)	8	12	16	28
DBIC(1)	8	16	20	30

If we compare random s-boxes to the AES s-box, we can see in table 7 that the AES s-box achieves better values in all non-trivial metrics. It is for this reason that cryptosystems based on key-derived s-boxes trade off ideal metric values for no underlying structure or generator function (see [4]).

4 Key-Derived S-Box Generation

It is important to consider some aspects when generating 8x8 key-derived s-boxes in order to achieve proper results. This process involves transforming the key into a pseudorandom sequence of the adequate length (key expansion) that can be used as the input for the permutation process that generates the final s-box. Other aspects are fixed point filtering mechanisms and accounting for different sizes that are also common in cryptography.

4.1 Key Expansion

In order to generate a key-derived s-box, the key usually requires stretching (expansion) into a pseudorandom sequence of the proper length. In the simplest case, this expansion process can consist on the repetition of the key up to the desired length; in other instances, the expansion can involve more complex procedures such as using a password based key derivation function (PBKDF) or a lightweight cryptographic primitive in pseudorandom mode: a key stream generator, a block cipher in a stream cipher mode or a hash function in repetitive iterations, for example.

Regarding length, it would seem that the longer the sequence the bigger the amount of swaps occurring during the permutation phase and, subsequently, the more random the resulting s-box is. On the contrary, there is a limit to the useful length because any sequence longer than the number of values of the s-box (256 for an 8x8 s-box) generates the same s-box as another sequence of 256 bytes or less.

Another important concept is avalanche, since it is desirable that similar keys (with very few differences) generate very different, unrelated, s-boxes. Using a complex expansion scheme or true pseudorandom generation rather than repetition can optimize the avalanche; unfortunately, this usually implies worse performance, also.

4.2 Permutation

The permutation process commences from a predetermined state (generally values 0 to 255 in ascending order) and takes each successive input byte from the sequence (*seq*) to modify the s-box swapping two values.

```
func GeneratePermutation(seq []byte) (s [256]byte) {
    for i := 0; i < 256; i++ {
        s[i] = i
    }
    j := 0
    for si := 0; si < len(seq); si++ {
        i := si % 256
        j = (j + s[i] + seq[si]) % 256
        s[i], s[j] = s[j], s[i]
    }
    return s
}
```

The RC4 algorithm employs this mechanism for initialization [14], using an expanded version of the key (through repetition, 256 bytes) as the pseudorandom sequence. Some bias problems have been found recently in RC4 [22], although s-boxes are usually employed in a different manner in other cryptosystems.

This scheme can also be repeated to generate multiple s-boxes, even using the previous s-box as the initial state for the following one. The following example

code constructs the second s-box ($s2$) using the first s-box ($s1$) as the initial state. In this case, the same sequence is used to construct both s-boxes but with different initial states, although another possibility is having a different sequence or sequence section for each s-box.

```
func GenerateMultiple(seq []byte) (s1,s2 [256]byte) {
    for i := 0; i < 256; i++ {
        s1[i] = i
    }
    j := 0
    for si := 0; si < len(seq); i++ {
        i := si % 256
        j = (j + s1[i] + seq[si]) % 256
        s1[i], s1[j] = s1[j], s1[i]
    }

    for i := 0; i < 256; i++ {
        s2[i] = s1[i]
    }
    j = 0
    for si := 0; si < len(seq); si++ {
        i := si % 256
        j = (j + s2[i] + seq[si]) % 256
        s2[i], s2[j] = s2[j], s2[i]
    }
    return s1,s2
}
```

Recently, Spritz (see [20]), a sponge-based drop-in replacement for RC4, has been proposed. In the paper, the authors include a new permutation algorithm that was found to have the best statistical properties after significant computational testing. This is meant to provide security against the short term biases found in RC4 (see [14, 17, 21]). This scheme includes an additional state variable, k, among other constructions that relate to the sponge architecture of Spritz.

We can summarize it as follows (although Spritz has additional functions):

```
func GenerateSpritzlike(seq []byte) (s [256]byte) {
    for i := 0; i < 256; i++ {
        s[i] = i
    }
    j := 0
        k := 0
    for si := 0; si < len(seq); si++ {
        i := si % 256
        j = (k + s[j + s[i]] + seq[si]) % 256
        k = (k + i + s[j]) % 256
        s[i], s[j] = s[j], s[i]
    }
    return s
}
```

Another approach would be to avoid swapping values and simply arrange the values from a pseudorandom sequence onto the s-box without repetition. Unfortunately, it is not possible to predetermine what sequence length will generate a complete s-box and the computation time would be dependent on the key too, enabling timing attacks.

4.3 Fixed Point Filtering

As described in section 3.2, it is desirable that the generated s-boxes do not exhibit any direct or reverse fixed points. This can be performed by the application of a post generation filter that performs a permutation on the offending values.

The straightforward approach is to check for each direct or reverse fixed point and swap for an adjacent value in order to remove the problem.

```
func isFixed(i,v byte) bool {
    if v == i || v == 255-i {
        return true
    }
    return false
}

func FilterFixedPoints(s [256]byte) {
    for i = 0; i < 256; i++ {
        if isFixed(i,s[i]) {
            j = i+1 % 256
            if isFixed(i,s[j]) {
                j++
            }
            s[i],s[j] = s[j],s[i]
```

```
        }
    }
}
```

Although this is very simple, the number of swaps (and the time taken) depends on the amount of fixed points and the adjacent values, making timing attacks possible. This problem can be solved if the filtering algorithm is coded in a way where the number of swaps is constant regardless of the number of fixed points in the s-box.

4.4 Other common sizes

Although 8x8 s-boxes are the most common due to implementation reasons, since they only require 256 bytes of memory and have a single byte input and output that is compatible with most memory controller designs, other sizes are also employed in other situations. One of these sizes is the 8x32 s-box, in which a single byte of input generates a quadruple byte (32 bit) word. This can be arranged as a single s-box filtering a group of 4 input bytes:

```
func Filter8x32(in uint32) (out uint32) {
    inb = [4]byte(in)
    out = S[inb[1]] ^ S[inb[2]] ^ S[inb[3]] ^ S[inb[4]]
    return out
}
```

or even as four different 8x32 s-boxes that are combined to generate a single 32 bit word output, approximating a 32x32 s-box (that would take an enormous amount of memory to implement directly):

```
func Filter8x32(in uint32) (out uint32) {
    inb = [4]byte(in)
    out = S1[inb[1]] ^ S2[inb[2]] ^ S3[inb[3]] ^ S4[inb[4]]
    return out
}
```

The simplest way to generate 8x32 s-boxes is by concatenation of proper, bijective, 8x8 s-boxes generated by the methods described above. In this way, the resulting s-box will be balanced and indistinguishable from a purely random 8x32 s-box. The four 8x8 s-boxes ($S1$ to $S4$) are combined so that for each input value, each s-box constitutes a single column of the resulting output (Sf):

```
Sf[i] = (S1[i]<<24) ^ (S2[i]<<16) ^ (S3[i]<<8) ^ S4[i]
```

In the case of a single 8x32 s-box, four 8x8 s-box must be generated and concatenated. For the case of an approximated 32x32 s-box (four 8x32 s-boxes) a total of sixteen 8x8 s-boxes are required.

Sometimes, smaller s-boxes like 4x4 are required. These can be generated using the same mechanism as an 8x8 s-box but adapting for a permutation of the values from 0 to 15, for example.

5 Conclusions

We have analysed s-boxes generated by RC4, Blowfish and Twofish, comparing them to pseudo-random s-boxes. During this study, we have found detectable differences in some of the metrics that could allow, to a certain degree, distinguishing Blowfish s-boxes and some Twofish s-boxes from random ones. This does not necessarily imply a vulnerable design but shows some small biases that could be improved. Nevertheless, our testing shows that key derivation is a suitable way to obtain random-equivalent s-boxes.

Another point of concern is the lack of balance in Blowfish s-boxes that, although they are employed as 8x32 s-boxes in the algorithm, when they are analysed as four adjacent 8x8 s-boxes they present statistical biases that could be exploited. This problem is corrected in its successor, Twofish, acknowledging the importance of maintaining proper balance in the 8x8 sub-s-boxes. It must be noted that despite being succeeded by Twofish, Blowfish is vastly more popular and widely used, like in the *bcrypt* password derivation algorithm (see [18]) and others.

We have established through randomness testing that key-derived s-boxes are essentially equivalent to random s-boxes, and can be equally employed as non-linear elements in many types of cryptosystems (stream and block ciphers, hash functions, etc.). They can function as standard substitution tables and as part of a key schedule algorithm. For this reason, we have analysed in detail some concepts related to the generation of proper, bijective, 8x8 key-derived s-boxes, in addition to fixed point filtering algorithms and generation of different sizes. In order to conduct our analysis, we have developed a custom framework for generating and testing 8x8 s-boxes.

Some open problems arise from the research performed for this paper, including the extension and improvement of the custom s-box generation/testing framework to support other generation algorithms and metrics, further cryptanalysis research involving the detected biases and the design of new cryptographic primitives based on the described techniques regarding key-derived s-boxes generation.

References

1. Adams, C. M., Tavares, S. E.: Designing S-Boxes For Ciphers Resistant To Differential Cryptanalysis. Proc. 3rd Symposium on State and Progress of Research in Cryptography, 181–190 (1993)
2. Álvarez, R., McGuire, G., Zamora, A.: The Tangle Hash Function. Submission to the NIST SHA-3 Competition (2008)
3. Álvarez, R., Vicent, J. F., Zamora, A.: Improving the Message Expansion of the Tangle Hash Function. Computational Intelligence in Security for Information Systems, LNCS 6694, 183–189. Springer (2011)
4. Álvarez, R., McGuire, G.: S-Boxes, APN Functions and Related Codes. Enhancing Cryptographic Primitives with Techniques from Error Correcting Codes, vol. 23, 49–62. IOS Press (2009)

5. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2. Internet Engineering Task Force (IETF), RFC 5246 (2008) http://tools.ietf.org/html/rfc5246
6. Fluhrer, S., Mantin, I., Shamir, A.: Weaknesses in the Key Scheduling Algorithm of RC4. Selected Areas in Cryptography, LNCS 2259, 1–24. Springer (2001)
7. Freier, A., Karlton, P., Kocher, P.: The Secure Sockets Layer (SSL) Protocol Version 3.0. Internet Engineering Task Force (IETF), RFC 6101 (2011) https://tools.ietf.org/html/rfc6101
8. Fuller, J., Millan, W.: On linear redundancy in the AES S-Box. Cryptology ePrint Archive, Report 2002/111
9. Fuller, J., Millan, W., Dawson, E.: Multi-objective Optimisation of Bijective S-boxes. Congress on Evolutionary Computation, vol. 2, 1525–1532 (2004)
10. The Go Programming Language. http://www.golang.org
11. Hussain, I., Shah, T., Gondal, M. A., Khan, W. A.: Construction of Cryptographically Strong 8x8 S-boxes. World Applied Sciences Journal, vol. 13–11, 2389–2395 (2011)
12. Jing-mei, L., Bao-dian, W., Xiang-guo, C., Xin-mei, W.: Cryptanalysis of Rijndael S-box and improvement. Applied Mathematics and Computation, vol. 170, 958–975. Elsevier (2005)
13. Kavut, S.: Results on rotation-symmetric S-boxes. Information Sciences, vol. 201, 93–113. Elsevier (2012)
14. Klein, A.: Attacks on the RC4 stream cipher. Designs, Codes and Cryptography, vol. 48–3, 269–286. Springer (2008)
15. Mister, S., Adams, C.: Practical S-Box Design. Selected Areas in Cryptography (1996)
16. Murphy, S., Robshaw, M. J. B.: Key-Dependent S-Boxes and Differential Cryptanalysis. Designs, Codes and Cryptography, vol. 27–3, 229–255. Springer (2002)
17. Paul, G., Maitra, S.: RC4 Stream Cipher and Its Variants. CRC Press (2012)
18. Provos, N., Mazeries, D.: Bcrypt Algorithm. USENIX (1999)
19. Rivest, R. L.: The RC4 Encryption Algorithm. RSA Data Security Inc. (1992)
20. Rivest, R, L., Schuldt, J.: Spritz – a spongy RC4-like stream cipher and hash function. http://people.csail.mit.edu/rivest/pubs/RS14.pdf. Presented at CRYPTO 2014 Rump Session (2014)
21. Sengupta, S., Maitra, S., Paul, G., Sarkar, S.: RC4: (Non-) random words from (non-) random permutations. IACR Cryptology ePrint Archive, 2011:448. (2011)
22. Sepehrdad, P., Vaudenay, S., Vuagnoux, M.: Discovery and Exploitation of New Biases in RC4. Selected Areas in Cryptography, Lecture Notes in Computer Science, vol. 6544, 74–91. Springer (2011)
23. Schneier, B.: Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish). Fast Software Encryption, Proc. Cambridge Security Workshop, 191–204. Springer-Verlag (1994)
24. Schneier, B., Kelsey, J., Whiting, D., Wagner, D., Hall, C., Ferguson, N.: The Twofish encryption algorithm: a 128-bit block cipher. John Wiley & Sons (1999)
25. Youssef, A. M., Tavares, S. E.: Resistance of Balanced S-boxes to Linear and Differential Cryptanalysis. Information Processing Letters, vol. 56–5, 249–252. Elsevier (1995)