Generating S-Boxes for Block Ciphers

Tommy Thorsen

17th January 2017

In this paper, we will explore methods for generating cryptographically strong s-boxes, using almost perfect nonlinear functions. We will also look at methods for determining the quality of any given s-box. In addition to looking at the theory, we implement the algorithms we find, using the python 3 programming language.

1 Introduction

An *s-box* (short for "substitution-box") is an essential component in many cryptographic algorithms. Its purpose is to provide the necessary nonlinearity the algorithm needs to withstand cryptanalysis. The requirement for nonlinearity stems back all the way to [Shannon, 1949], and his properties of *confusion* and *diffusion*. Nonlinear functions provide confusion, which means that as many bits of the key as possible goes into creating each bit of the ciphertext. Along with p-boxes ("permutation boxes") which provide diffusion, they provide the bulk of the logic in a cryptographic cipher algorithm.

Not all nonlinear functions make good s-boxes. In addition to being nonlinear, there are a number of criteria that must be fulfilled for the function to be a good s-box. These criteria ensure resistance to techniques such as differential and linear cryptanalysis. [Brickell et al., 1986] gives a list of five criteria that were used in the design of the s-boxes for the DES algorithm. [Coppersmith, 1994] elaborates on these criteria, and extends the list to eight criteria.

A simpler, more general, set of four criteria can be found in [Adams and Tavares, 1990]. These four criteria are *nonlinearity*, *bijection*, *strict avalanche* and *independence of output bits*. The paper claims that these four general criteria encompass most of the criteria for DES. In fact, by requiring the strict avalanche property, most of the DES criteria will be fulfilled. In addition to the criteria themselves, the paper provides an algorithm that can be used to check the quality of a generated s-box.

In addition to looking at existing theory on this subject, we will get our hands dirty and try to implement a set of computer programs that can find new s-boxes, and automatically evaluate their strengths. It will be of interest to compare these results with measurements on existing s-boxes, such as the ones used in DES, KASUMI or other common cryptographic algorithms. In the next section, we will go into more detail on both the theory and the implementation.

2 Methods

2.1 Generating s-boxes

Now that we have stated our goal — the generation of cryptographically strong s-boxes — where do we start? One possible approach would be to first implement the above-mentioned algorithm for measuring the strength of an s-box. We could then generate s-boxes completely at random, rank them using our algorithm, and throw away those that were of insufficient strength. While this approach might yield some viable s-boxes, we would not be guaranteed any good results, as the search space is very large. For a 7-bit s-box, there are more than 10^{215} potential s-boxes. The author would much prefer a smarter and more elegant approach to the task at hand.

A good solution seems to be so-called *almost perfect nonlinear functions* (APN functions), or *bent functions* as they are sometimes called. There is quite a lot of literature on this subject, for instance [Rothaus, 1976] which may be the paper that introduced the term "bent" functions. Another good paper on the subject is [Nyberg, 1991], which has a lot of information on these functions from a cryptographic standpoint.

Power functions such as $f(x) = x^d$ over $GF(2^n)$ can be shown to be APN, if d is carefully chosen. There are a few different formulas that can be used to find good values for d. The two most popular ones are:

Gold's function $d = 2^k + 1;\ (n, k) = 1;\ 1 \leq k \leq m$

Kasami's function $d = 2^{2k} - 2^k + 1;\ (n, k) = 1;\ 2 \leq k \leq m$

Additionally, we will be applying a linear transformation $F(x) = a \cdot f(x) + b$ on the APN function.

As we can see, there are quite a lot of moving parts in this formula. This gives us a lot of options when it comes to generating new s-boxes. The first, most obvious thing to try, is different values for d. We can get different values by first trying all possible values of k, but this does not give us a very large number of different values for d. The number of values for d can be increased somewhat by making use of the following rule from Hans Dobbertin:

> If for an exponent d the power function x^d is maximally nonlinear (resp. APN), then the same is true for $2^i d \ mod\ 2^n - 1\ (i < n)$
>
> —[Dobbertin, 1999]

We will be operating within a finite field $GF(2^7)$ *mod* $f(x)$ where $f(x)$ is our generating polynomial. This polynomial can also be varied somewhat. Finally, we can start modifying a and b in our linear transformation. All these different variables combined, should give us plenty of room to discover new viable s-boxes, especially for larger values of n.

To keep the implementation simple, and the running time within manageable bounds, we will keep the generating polynomial constant, while trying all possible values of the other variables. This will give us something like algorithm 1 below.

Algorithm 1 Algorithm for generating s-boxes

```
function SBOXSEARCH($n$)
    $D \leftarrow$ the set of all possible values of $d$ given $n$
    $P \leftarrow$ the set of all polynomials of degree less than $n$
    $S \leftarrow \emptyset$
    $f \leftarrow$ an arbitrary polynomial of degree $n$
    for all $d \in D$ do
        for all $a \in P, a \neq 0$ do
            for all $b \in P$ do
                $S \leftarrow S \cup createSbox(f, a, b, n, d)$
            end for
        end for
    end for
    return $S$
end function
```

2.2 Testing s-boxes

Now that we have generated some candidate s-boxes, it's time to see if they're any good. Let's step back to [Adams and Tavares, 1990]. This paper specifies four general criteria that a good s-box should fulfill. We will go into more detail for each of these criteria, below. The paper also presents an algorithm for checking whether or not an s-box fulfills the four criteria. Since we will be using a slightly different approach to building our s-box, the algorithm is not usable verbatim, but we will keep this algorithm in mind when implementing our own algorithm.

2.2.1 Nonlinearity

This first criteria is the most essential requirement of an s-box. It provides protection against cryptanalysis with linear and near-linear methods, but what does nonlinearity really mean? [Meier and Staffelbach, 1989] provides the following definition:

> *With respect to linear structures, a function f has optimum nonlinearity if for every nonzero vector $a \in GF(2)^n$ the values $f(x+a)$ and $f(x)$ are equal*

for exactly half of the arguments $x \in GF(2)^n$. If a function f satisfies this property we will call it perfect nonlinear with respect to linear structures, or briefly perfect nonlinear.

For measuring nonlinearity we will go with the following formula from [Matsui, 1993]. This formula is made specifically for measuring the linearity of the DES s-boxes, hence the hard-coded sizes and bit-widths. It is, however quite simple to generalize this formula for any size s-box.

$$NS_a(\alpha,\beta) \overset{def}{=} \#\{x|0 \le x < 64,\ (\bigoplus_{s=0}^{5}(x[s]\cdot\alpha[s])) = (\bigoplus_{t=0}^{3}(S_a(x)[t]\cdot\beta[t]))\}$$

This formula gives us the number of coincidences for a single input-output pair. If we run this formula for all possible inputs and outputs, the maximal absolute value returned by the formula gives us our measure of nonlinearity. The lower the number, the more nonlinear the s-box. We have incorporated the formula into algorithm 2.

Algorithm 2 Algorithm for measuring the nonlinearity of an s-box

```
function MEASURENONLINEARITY($S = [s_0, s_1, s_2, ..., s_{|S|-1}]$)
                                        ▷ S (the s-box) is an array of size $2^n$
    $m \leftarrow 0$
    for all $0 \le \alpha, \beta < |S|$ do
        $c \leftarrow -|S|/2$
        for all $0 \le x < |S|$ do
            if $(wt(x \cdot \alpha)\ mod\ 2) = (wt(s_x \cdot \beta)\ mod\ 2)$ then
                                        ▷ wt is the hamming weight
                $c \leftarrow c + 1$
            end if
        end for
        if $\alpha \ne 0 \wedge \beta \ne 0 \wedge |c| > m$ then
            $m \leftarrow |c|$
        end if
    end for
    return $m$
end function
```

2.2.2 Bijection

The requirement that the s-box is invertible. This means that the s-box must be a permutation. Even though invertibility may not be a requirement for all cipher structures, we will still keep it as a requirement. Algorithm 3 is a simple algorithm for checking invertibility.

Algorithm 3 Algorithm for checking that an s-box is invertible

```
function CHECKINVERTIBLE($S = [s_0, s_1, s_2, ..., s_{|S|-1}]$)
                                        ▷ S (the s-box) is an array of size $2^n$
    $i \leftarrow |S|$
    $A \leftarrow [a_0, a_1, a_2, ..., a_{|S|-1}], \forall a \in A \; a \leftarrow 0$
    for all $s \in S$ do
        if $s < 0 \vee s \geq |S|$ then
            return false                                  ▷ Illegal number
        end if
        if $a_s = 1$ then
            return false                                  ▷ Found a duplicate
        else
            $a_s \leftarrow 1$
            $i \leftarrow i - 1$
        end if
    end for
    return $i = 0$
end function
```

2.2.3 Strict avalanche

The strict avalanche criterion (SAC) was introduced in [Webster and Tavares, 1986]. In order to satisfy this criterion, changing one bit of the input should cause each bit of the output to be changed with a probability of one half. The paper suggests a method for determining if an s-box satisfies SAC which involves splitting the all the possible inputs into a list of input pairs such that these input pairs differ only in a single bit. For simplicity, we will be using a slightly different algorithm, as outlined in algorithm 4.

Algorithm 4 Algorithm for checking that an s-box satisfies SAC

```
function CHECKSAC($S = [s_0, s_1, s_2, ..., s_{|S|-1}]$)
                                        ▷ S (the s-box) is an array of size $2^n$
    $c \leftarrow 0$
    for all $0 \leq i < |S|$ do
        for all $0 \leq b < n$ do
            $c \leftarrow c + wt(s_i \oplus s_{i'})$          ▷ $i'$ is $i$ with bit $b$ flipped
                                                  ▷ wt is the hamming weight
        end for
    end for
    return $\frac{c}{|S|n^2}$
end function
```

2.2.4 Independence of output bits

Finally, the bit independence criterion (BIC) is the requirement that any two output bits must not be equal to each other significantly more than or less than half the time. Failure to fulfill this criteria may allow the cryptanalyst to reduce his search space. Again, the explanation of this criterion can be found in [Webster and Tavares, 1986]. We will use algorithm 5 to measure this criterion.

Algorithm 5 Algorithm for checking that an s-box satisfies BIC

```
function CHECKBIC($S = [s_0, s_1, s_2, ..., s_{|S|-1}]$)
                                        ▷ S (the s-box) is an array of size $2^n$
    $C \leftarrow [c_{0,0}, c_{0,1}, c_{0,2}, ..., c_{n-1,n-1}], \forall c \in C\ c \leftarrow 0$
    for all $0 \leq i < |S|$ do
        for all $0 \leq b < n$ do
            for all $0 \leq j, k < n; j \neq k$ do
                if $s_i[j] \oplus s_{i'}[j] = s_i[k] \oplus s_{i'}[k]$ then      ▷ $i'$ is $i$ with bit $b$ flipped
                    $c_{j,k} \leftarrow c_{j,k} + 1 - 2(s_{i'}[j] \oplus s_{i'}[k])$
                end if
            end for
        end for
    end for

    return the largest absolute value in the set $C$
end function
```

2.3 Implementation

The programs that accompany this paper, come in the form of a set of `Python3` scripts. The `Python3` language was chosen because of the author's familiarity with the language, but also because of the availability of useful libraries such as `NumPy`, which provides powerful scientific computing capabilities. `Numpy`'s `Polynomial` class in particular, is central in the implementation of the polynomial arithmetic used to generate permutations of our fields and to calculate the final s-box based on the APN function.

This polynomial class does unfortunately not have support for polynomials in GF(2), so I have written a wrapper class, `G2Polynomial` which extends `Numpy`'s `Polynomial` class with functionality for performing the kind of reductions we can do on a polynomial in GF(2) (i.e. $x^2 + x^2 = 0$). `G2Polynomial` also contains functions for converting from and to strings of binary digits.

All of the three scripts should be relatively easy to run on any system, provided the two dependencies (python + numpy) are in place. All of the scripts will give somewhat useful information when launched with `-h` or `-help`. Let's take a closer look at the three scripts.

2.3.1 `create-sbox`

This is the tool that takes all your parameters, and generates an actual s-box. If launched with all of its parameters, it's fully automatic. If any of the parameters are missing, it will query the user to enter the parameters on the command line. Figure 1 shows what this looks like.

2.3.2 `check-sbox`

After you have created an sbox with `create-sbox`, you can test its strength with this tool. Call `./check-sbox <filename>` to check an sbox stored in a file, or `./check-sbox -` to read the sbox from `stdin`. See figure 2.

2.3.3 `sbox-search`

Call `./sbox-search -n <n>` to have the tool traverse all possible values for the given n and create an s-box for each combination. The resulting s-boxes will be put in individual files in the `out` folder. The s-box files will contain the evaluation from `check-sbox`. S-boxes that fail the check are still written to `out`, and must be manually eliminated if unwanted.

It is also possible to use `sbox-search` to generate single, random s-boxes for a given n. Just call `./sbox-search -r -n <n>`. In this mode, the generated s-box will be written to `stdout`.

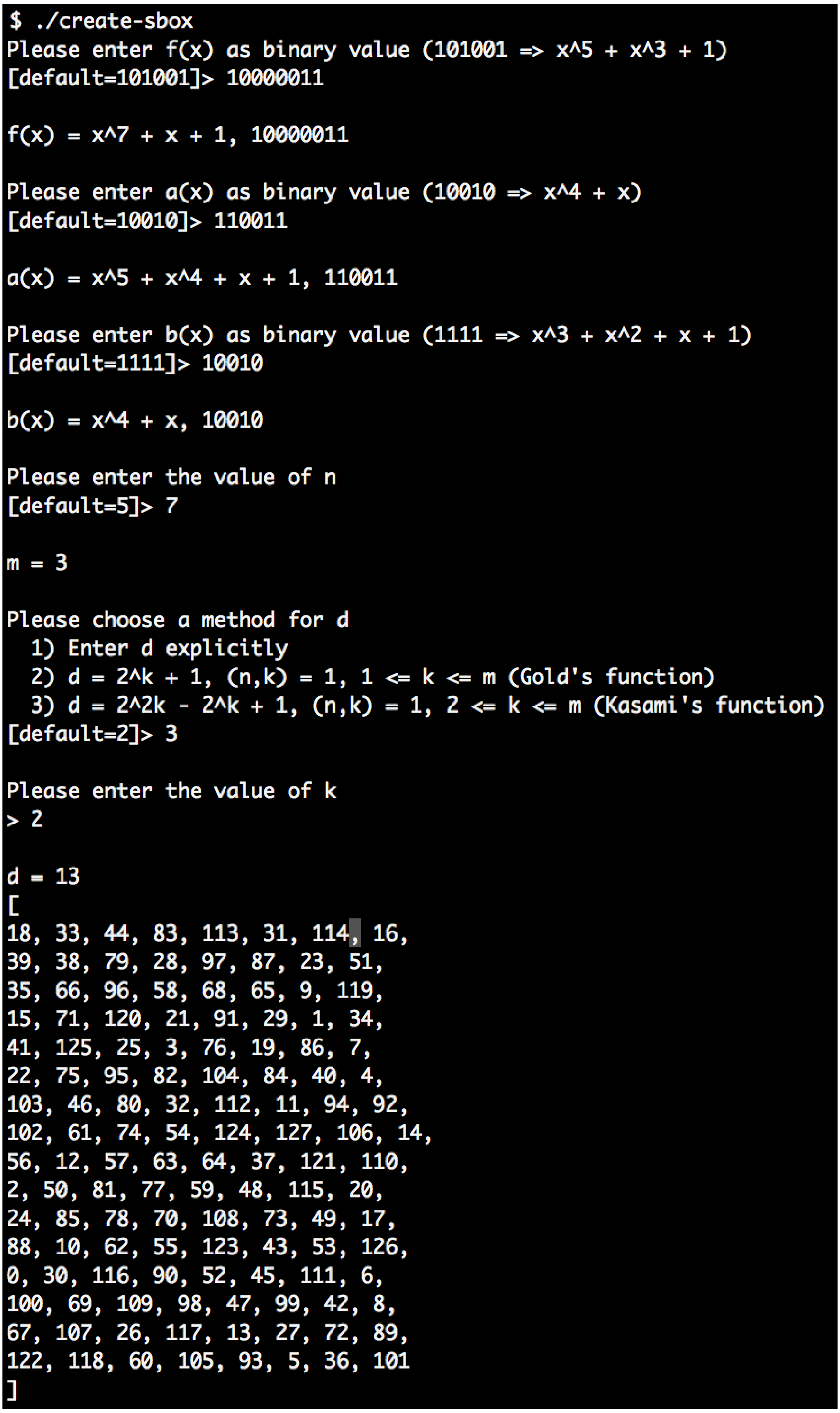
3 Results

Preliminary results showed that the algorithm used to generate the s-boxes from the APN functions over the finite fields produces very good results. Because of this, I decided to make the strength-checking tool **very** strict. `check-sbox` will only give a `PASS` if the nonlinearity of the s-box is the maximum possible nonlinearity, `BIC` is 0 and `SAC` is **exactly** 0.5. Even with these extreme requirements, the search tool is able to come up with multiple viable s-boxes for $n \in \{3, 5, 7\}$. These requirements are, however, strict enough that most other s-boxes fail. For instance KASUMI's S7 s-box fails because it has a `SAC` value of 0.5089.

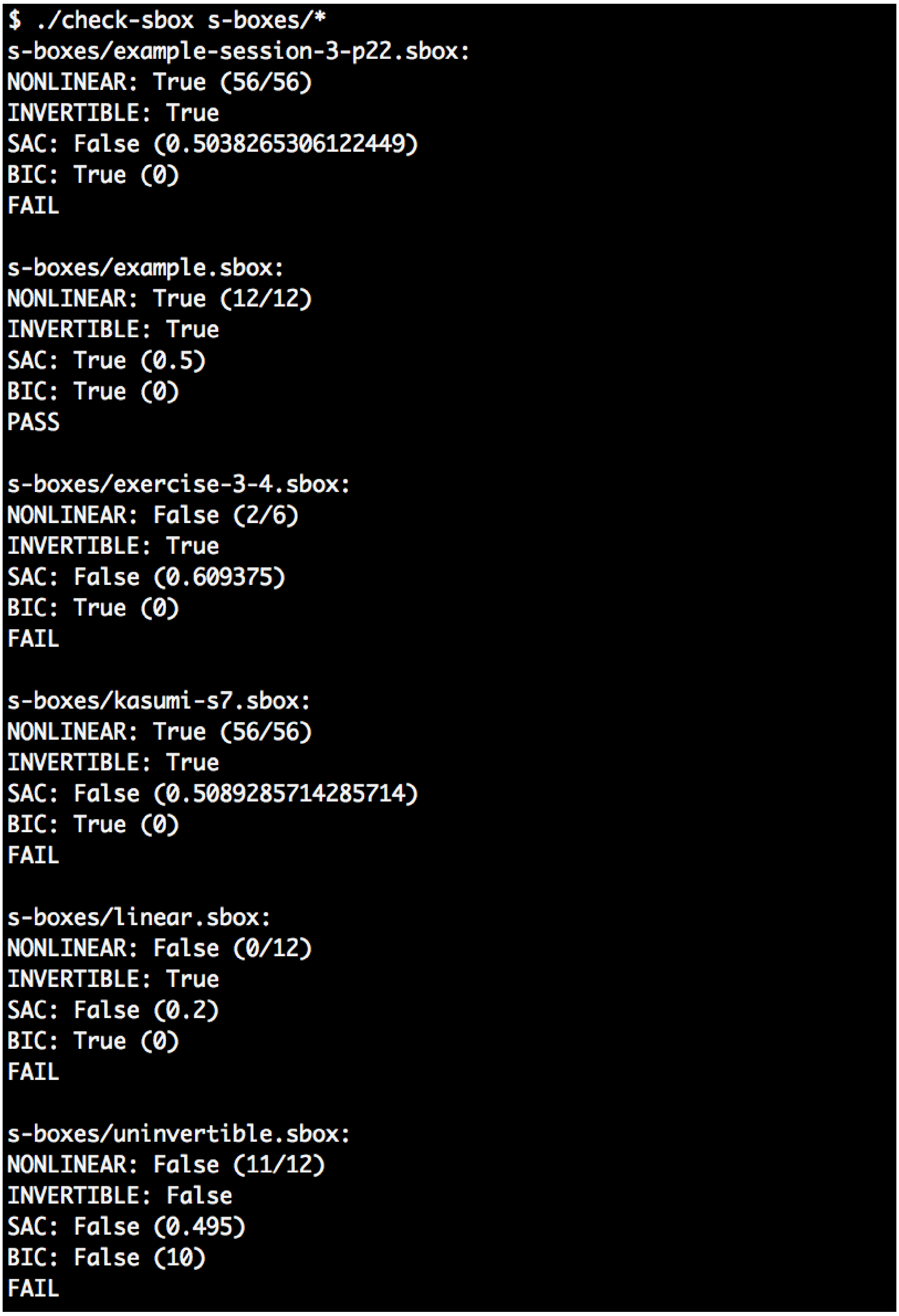
As expected, the tools are only able to produce good s-boxes for odd values of n. The s-boxes generated by even n all fail the checks, and most of them fail all of our four checks.

For lower values of n, the available combinations of variables are few, and `sbox-search` returns relatively few candidates. For $n = 3$ the number of generated s-boxes is 36. 32 of these fail the strength checks, which leaves us with 4 good looking s-boxes of size 3. Nonetheless, it is interesting that strong s-boxes of this size exists at all.

For $n = 5$, our selection is much better. `sbox-search` returns 3600 candidates in total. 736 of these pass our tests, which gives us plenty of strong s-boxes to choose from. For $n = 7$, the script generates 141120 candidates, 79781 of which pass the checks.



```
$ ./create-sbox
Please enter f(x) as binary value (101001 => x^5 + x^3 + 1)
[default=101001]> 10000011

f(x) = x^7 + x + 1, 10000011

Please enter a(x) as binary value (10010 => x^4 + x)
[default=10010]> 110011

a(x) = x^5 + x^4 + x + 1, 110011

Please enter b(x) as binary value (1111 => x^3 + x^2 + x + 1)
[default=1111]> 10010

b(x) = x^4 + x, 10010

Please enter the value of n
[default=5]> 7

m = 3

Please choose a method for d
  1) Enter d explicitly
  2) d = 2^k + 1, (n,k) = 1, 1 <= k <= m (Gold's function)
  3) d = 2^2k - 2^k + 1, (n,k) = 1, 2 <= k <= m (Kasami's function)
[default=2]> 3

Please enter the value of k
> 2

d = 13
[
18, 33, 44, 83, 113, 31, 114, 16,
39, 38, 79, 28, 97, 87, 23, 51,
35, 66, 96, 58, 68, 65, 9, 119,
15, 71, 120, 21, 91, 29, 1, 34,
41, 125, 25, 3, 76, 19, 86, 7,
22, 75, 95, 82, 104, 84, 40, 4,
103, 46, 80, 32, 112, 11, 94, 92,
102, 61, 74, 54, 124, 127, 106, 14,
56, 12, 57, 63, 64, 37, 121, 110,
2, 50, 81, 77, 59, 48, 115, 20,
24, 85, 78, 70, 108, 73, 49, 17,
88, 10, 62, 55, 123, 43, 53, 126,
0, 30, 116, 90, 52, 45, 111, 6,
100, 69, 109, 98, 47, 99, 42, 8,
67, 107, 26, 117, 13, 27, 72, 89,
122, 118, 60, 105, 93, 5, 36, 101
]
```



```
$ ./check-sbox s-boxes/*
s-boxes/example-session-3-p22.sbox:
NONLINEAR: True (56/56)
INVERTIBLE: True
SAC: False (0.5038265306122449)
BIC: True (0)
FAIL

s-boxes/example.sbox:
NONLINEAR: True (12/12)
INVERTIBLE: True
SAC: True (0.5)
BIC: True (0)
PASS

s-boxes/exercise-3-4.sbox:
NONLINEAR: False (2/6)
INVERTIBLE: True
SAC: False (0.609375)
BIC: True (0)
FAIL

s-boxes/kasumi-s7.sbox:
NONLINEAR: True (56/56)
INVERTIBLE: True
SAC: False (0.5089285714285714)
BIC: True (0)
FAIL

s-boxes/linear.sbox:
NONLINEAR: False (0/12)
INVERTIBLE: True
SAC: False (0.2)
BIC: True (0)
FAIL

s-boxes/uninvertible.sbox:
NONLINEAR: False (11/12)
INVERTIBLE: False
SAC: False (0.495)
BIC: False (10)
FAIL
```

Figure 2: Running `check-sbox` on multiple s-boxes.

Generating all these s-boxes can take quite some time. For the author, it took the better part of a weekend on a normal workstation. It may be better to use the random mode of `sbox-search` for generating single s-boxes of this size and larger.

For the reader's convenience, we include some of the good s-boxes below.

3.1 Strong s-boxes of size $n = 3$, $f(x) = x^3 + x + 1$

Note that all four of these were created with almost the same parameters. Only $b(x)$ differs.

$d = 6$
$a(x) = x + 1$
$b(x) = 0$

```
0, 3, 4, 1, 2, 6, 5, 7
```

—

$d = 6$
$a(x) = x + 1$
$b(x) = 1$

```
1, 2, 5, 0, 3, 7, 4, 6
```

—

$d = 6$
$a(x) = x + 1$
$b(x) = x$

```
2, 1, 6, 3, 0, 4, 7, 5
```

—

$d = 6$
$a(x) = x + 1$
$b(x) = x + 1$

```
3, 0, 7, 2, 1, 5, 6, 4
```

3.2 Strong s-boxes of size $n = 5$, $f(x) = x^5 + x^2 + 1$

$d = 5$
$a(x) = x^2 + x + 1$
$b(x) = x^2$

```
 4,  3, 31,  9, 28,  1, 24, 20, 19,  6, 21, 17,  7,  8, 30,  0,
 5, 22, 14, 12, 27, 18, 15, 23, 11, 10, 29, 13, 25,  2, 16, 26
```

—

$d = 12$
$a(x) = x^3 + x^2 + 1$
$b(x) = x^3 + x$

```
10,  7,  6, 30,  8,  0, 12, 17, 22, 21,  9, 31, 11, 13, 28, 15,
19, 26,  1, 29, 24, 20,  2, 27,  4,  3,  5, 23, 16, 18, 25, 14
```

—

$d = 20$
$a(x) = x^2 + x + 1$
$b(x) = x^3 + x^2 + x + 1$

```
15,  8, 14, 29,  3, 12,  9, 18, 21, 11, 27, 17,  2, 20,  7,  5,
 6, 22, 26, 30,  4, 28, 19, 31, 25, 16, 10, 23,  0,  1, 24, 13
```

—

$d = 11$
$a(x) = x^3 + x^2 + x + 1$
$b(x) = 0$

```
 0, 15,  8, 16, 29, 27, 31, 12, 25,  2, 11, 17, 23,  3,  1, 22,
 5, 26, 14, 30, 20, 19, 24, 18, 10,  6,  9, 28,  7, 21, 13,  4
```

3.3 Strong s-boxes of size $n = 7$, $f(x) = x^7 + x + 1$

$d = 5$
$a(x) = x^3 + x + 1$
$b(x) = x^4 + x^3$

```
24, 19, 126, 80, 115, 110, 46, 22, 86, 44, 127, 32, 79, 35, 93,
20, 109, 74, 15, 13, 83, 98, 10, 30, 71, 17, 106, 25, 11, 75,
29, 120, 31, 7, 100, 89, 119, 121, 55, 28, 78, 39, 122, 54, 84,
43, 91, 1, 65, 117, 62, 47, 124, 94, 56, 63, 116, 49, 68, 36,
59, 104, 48, 70, 123, 125, 113, 82, 57, 41, 8, 61, 85, 34, 16,
66, 101, 4, 27, 95, 103, 77, 105, 102, 112, 76, 69, 92, 45, 118,
108, 18, 72, 5, 50, 90, 2, 23, 21, 37, 67, 64, 111, 73, 51, 87,
107, 42, 0, 114, 99, 52, 53, 12, 38, 58, 33, 14, 9, 3, 96, 40,
60, 81, 6, 88, 97, 26
```

—

$d = 66$
$a(x) = x^2 + x + 1$
$b(x) = x^2$

```
51, 41, 119, 20, 121, 114, 120, 10, 0, 47, 23, 65, 72, 118, 26,
93, 69, 107, 66, 21, 35, 28, 97, 39, 101, 126, 49, 83, 1, 11,
16, 99, 71, 111, 8, 89, 58, 3, 48, 112, 40, 53, 52, 80, 87, 91,
14, 123, 113, 109, 125, 24, 32, 45, 105, 29, 13, 36, 82, 2, 94,
102, 68, 5, 84, 64, 46, 67, 9, 12, 54, 74, 55, 22, 30, 70, 104,
88, 4, 77, 63, 31, 6, 95, 78, 127, 50, 122, 79, 90, 37, 73, 60,
56, 19, 110, 76, 106, 61, 98, 38, 17, 18, 92, 115, 96, 81, 59,
27, 25, 124, 7, 103, 117, 85, 62, 33, 34, 86, 44, 75, 108, 42,
116, 15, 57, 43, 100
```

—

$d = 68$
$a(x) = x^4 + x$
$b(x) = x^5 + x^3 + x^2 + 1$

```
45, 63, 13, 59, 118, 44, 22, 104, 40, 105, 11, 110, 70, 79, 37,
8, 16, 36, 3, 19, 33, 93, 114, 42, 31, 120, 15, 76, 27, 52, 75,
64, 17, 124, 61, 116, 23, 50, 123, 122, 0, 62, 47, 53, 51, 69,
92, 14, 86, 29, 73, 38, 58, 57, 101, 66, 77, 85, 81, 109, 20, 68,
72, 60, 55, 84, 82, 21, 65, 106, 100, 107, 96, 80, 6, 18, 35, 91,
5, 89, 125, 56, 43, 74, 97, 108, 119, 94, 32, 54, 117, 71, 9, 87,
28, 102, 115, 111, 26, 34, 88, 12, 113, 1, 48, 127, 90, 49, 46,
41, 4, 39, 67, 121, 25, 7, 2, 112, 24, 78, 10, 99, 83, 30, 126,
95, 103, 98
```

4 Discussion

Based on the experiments that have been performed, it must be concluded that using APN functions to generate s-boxes works very well. Some may argue that there might exist very strong s-boxes that can not be generated using an APN function, and while this may be true, this method has proven to be able to generate more s-boxes than we need.

We have mentioned above, how we have been able to search for s-boxes for $n \in \{3, 5, 7\}$. Why not $n = 9$? Well, it turns out that `check-sbox` performs very poorly for large values of n. When using it to measure the strength of KASUMI's S9 s-box, it takes about 1 minute and 16 seconds on the author's laptop. While this may be acceptable for checking a single s-box, it is not usable for deciding whether to pass or fail a large number of generated s-boxes. The slowness of this tool is mainly caused by the nonlinearity measurement algorithm being inefficient. A possible future improvement might be to replace this with for instance the *Fast Walsh-Hadamard* algorithm, which promises to be a more efficient way to measure nonlinearity.

There is an interesting observation we can make from the s-box generation numbers presented above. Remember we said that for $n = 3$, 4 out of 36 s-boxes passed the checks? These corresponding numbers were 736 out of 3600 for $n = 5$ and 79781 out of 141120 for $n = 7$. As expected, we are able to generate more s-boxes for higher values of n — what was not expected, however, is that the percentage of s-boxes passing the checks is also higher for larger values of n. The percentages are 11%, 20% and 57% respectively. Does this indicate that larger size s-boxes are intrinsically more secure? Are there criteria even more strict than the four we've been using, that can only be passed by s-boxes of larger size?

Looking at the results from the strength-checking tool, it seems that some of the checks are more useful than the others. Due to the way we generate our s-boxes, the nonlinearity check is not actually necessary, as all s-boxes generated by this method are maximally nonlinear (for odd n, anyway). The same goes for the invertibility check. These two are still very useful checks for measuring s-boxes generated by other methods. The least useful check seems to be the BIC check. This almost never returns false, and in the cases where it does return false, the other checks also tend to return false.

I would like to conclude this paper by encouraging the reader to check out the accompanying python scripts. Much of the effort behind this paper has gone into the implementation of all these famous algorithms and formulas, and this will hopefully be reflected in the usefulness of these tools.

References

[Adams and Tavares, 1990] Adams, C. and Tavares, S. (1990). The structured design of cryptographically good s-boxes. *journal of Cryptology*, 3(1):27–41.

[Brickell et al., 1986] Brickell, E. F., Moore, J. H., and Purtill, M. (1986). Structure in the s-boxes of the des. In *Advances in Cryptology—CRYPTO'86*, pages 3–8. Springer.

[Coppersmith, 1994] Coppersmith, D. (1994). The data encryption standard (des) and its strength against attacks. *IBM journal of research and development*, 38(3):243–250.

[Dobbertin, 1999] Dobbertin, H. (1999). Almost perfect nonlinear power functions on gf (2 n): the niho case. *Information and Computation*, 151(1):57–72.

[Matsui, 1993] Matsui, M. (1993). Linear cryptanalysis method for des cipher. In *Advances in Cryptology—EUROCRYPT'93*, pages 386–397. Springer.

[Meier and Staffelbach, 1989] Meier, W. and Staffelbach, O. (1989). Nonlinearity criteria for cryptographic functions. In *Advances in Cryptology—EUROCRYPT'89*, pages 549–562. Springer.

[Nyberg, 1991] Nyberg, K. (1991). Perfect nonlinear s-boxes. In *Advances in Cryptology—EUROCRYPT'91*, pages 378–386. Springer.

[Rothaus, 1976] Rothaus, O. S. (1976). On “bent” functions. *Journal of Combinatorial Theory, Series A*, 20(3):300–305.

[Shannon, 1949] Shannon, C. E. (1949). Communication theory of secrecy systems*. *Bell system technical journal*, 28(4):656–715.

[Webster and Tavares, 1986] Webster, A. and Tavares, S. E. (1986). On the design of s-boxes. In *Advances in Cryptology—CRYPTO'85 Proceedings*, pages 523–534. Springer.