Distributed Computation of SBoxes with Strong Security Properties

Panayotis E. Nastou
Dept of Mathematics
University of Aegean
Samos, Greece
pnastou@aegean.gr

Yannis C. Stamatiou
Dept of Business Administration
University of Patras
Patras, Greece
stamatiou@ceid.upatras.gr

Abstract

Substitution boxes are among the most critical components of a secure block cipher design. A substitution box or, for short, SBox, is a set of Boolean functions implementing a nonlinear mapping of inputs to outputs and it is employed for mixing the input of the cipher with the encryption key so that the output of the cipher reveals no information about the encryption key. Over the years, a number of good practices have evolved that, if employed, can lead to the construction of SBoxes with good security properties that lead to the cipher's resistance against known and envisaged cryptanalysis techniques. One such practice is the employment of particular classes of Boolean functions that possess a number of desirable properties. A drawback of such an approach is that it is frequently a computationally demanding task to check that the employed Boolean functions and the resulting SBox have the target properties. In this paper we describe a distributed algorithm that can accelerate significantly the construction of SBoxes with desirable cryptographic properties. The algorithm has been implemented on a computer cluster and it is fully parametric, with respect to the type of Boolean functions it can use as constituents of the SBox. A designer can use a class of Boolean functions with algorithmically definable properties in order to produce SBoxes of arbitrary sizes. We also present evidence for the algorithm's efficiency by comparing it with the best sequential approach available in a series of different experimental setups.

Keywords: *Symmetric Block Ciphers, Distributed SBox Computation, Nonlinearity, SAC criterion, Bent Functions, Almost Perfect Nonlinear functions, Computing Cluster.*

1 Introduction

Many of the network security protocols employed today utilize symmetric block ciphers. *IPSec*, for example, which is a widely used *Virtual Private Network* protocol (VPN), employs in the encapsulation mode the triple *DES*, *AES* in various modes, *CAST* and many other symmetric block ciphers. Many of the employed symmetric block ciphers are based on the *Feistel* block cipher structure, which is actually a product cipher that alternates, in a number of rounds, the operations of bit substitutions and permutations on the input data and key bits.

For a block cipher design based on the Feistel structure the block and key sizes, the number of rounds, the key schedule and the round function determine the security properties

of the design. As for the design of the round function f_i of Feistel based block ciphers, which may be different in every round i, it is fundamental to provide a high degree of diffusion and confusion. Considering, the input x of a round function f_i and the function value y, i.e. $y = f_i(x)$, as binary numbers, the maximum degree of confusion is achieved if every bit of y has the same probability to change when any bit of x changes. Moreover, a high degree of diffusion is provided if any change on a part of x should change a large number of bits of y. On a Feistel based block ciphers confusion is provided by the *substitution operation*, which is based on a look up table called *Substitution Box* or SBox for short.

If V_n is the vector space of dimension n over the field F_2, a $n \times m$ SBox S, is a mapping of $n - bit$ strings to $m - bit$ strings, i.e. $S : V_n \to V_m$. Alternatively, an $n \times m$ SBox can be considered as a set $S(x) = \{C_{m-1}(x), \ldots, C_0(x)\}$ of m Boolean functions $C_i(x)$ on V_n, that is $C_i : V_n \to F_2$, where $0 \leq i < m$. Various SBox sizes considered in a number of well known block ciphers are 6×4 for DES, 8×8 for AES, and 8×32 for CAST. In order to construct block ciphers which are resistant to currently known cryptanalytic attacks, such as linear and differential cryptanalysis, the employed SBoxes should be highly nonlinear and have a certain bit dispersal, i.e. avalanche, properties since these properties are highly related to confusion and diffusion described earlier.

In [7], a formal method that generates SBoxes according to certain mathematical properties of Boolean functions and their linear combinations is presented. Although this method provides cryptographically strong SBoxes, its main disadvantage is that it is computationally intensive. In general, for large values of n and m the computation of cryptographically strong SBoxes may be an intractable problem. In the literature (see, e.g., [2, 3, 4]) for $n = 8$, the space of all Boolean functions is considered large and consequently its exhaustive exploration for discovering functions with good cryptographic properties is considered a computationally intensive task.

In this paper, we present a distributed algorithm and its implementation on a computing cluster that accelerates the computation of secure substitution boxes with good security properties. Using this implementation, a block cipher designer can use any class of algorithmically definable Boolean functions in order to compute, within a reasonable time span, an SBox that satisfies a set of desirable properties. It is tunable and parametric since it can accept as input the algorithmic description (i.e. generation algorithm) of any class of Boolean functions, the target SBox dimensions n and m, as well as the target properties that should be satisfied by the SBox component functions and the SBox itself.

With respect to the structure of the paper, Sect. 2 presents an introduction of the basic theory of cryptographically strong SBoxes while Sects 3 and 4 present a distributed algorithm and its implementation on a computer cluster for the (computationally demanding) task of the computation of secure SBoxes. Experimental results that demonstrate the efficiency of the distributed implementation over the sequential counterpart are presented in Sect. 5. Finally, in Sect. 6 we discuss the obtained results and provide directions for further research.

2 Background

By $x \in V_n$ we denote the binary vector $x = (x_{n-1} \ldots x_0)$ on n Boolean variables where V_n is the vector space of dimension n over the field F_2. The sequence of values of a Boolean function f of n variables, i.e. the sequence $(f(0), f(1), \ldots, f(2^n - 1))$, is called the *truth*

table of function f. As it was mentioned above, an $n \times m$ *SBox* S can be considered as a set $S(x) = \{f_{m-1}(x), \ldots, f_0(x)\}$ of m Boolean functions $f_i(x)$ on V_n, that is $f_i : V_n \to F_2$, where $0 \leq i < m$. A Boolean function f_i of an SBox is called *SBox column*.

The *linear combination* of two Boolean functions f *and* g on V_n is defined by

$$(f \oplus g)(x) = f(x) \oplus g(x) \tag{1}$$

where $\oplus$ is the XOR operation. A Boolean function f on V_n is called *affine* if for every $x = (x_{n-1} \ldots x_0)$ it holds

$$f(x) = (w \cdot x) \oplus y \tag{2}$$

where $w = (w_{n-1} \ldots w_0) \in V_n$, $y \in F_2$ and $w \cdot x = w_{n-1} \cdot x_{n-1} \oplus \ldots \oplus w_0 \cdot x_0$. If $y = 1$, the inner product $w \cdot x$ on a specific value of x is simply inverted, while for $y = 0$ there is no effect on the inner product and f is called *linear*.

The *Hamming weight* $wt(f)$ of a Boolean function f is defined as the number of $1s$ in the truth table of f. A Boolean function f on V_n is *balanced* if $wt(f) = 2^{n-1}$, i.e. half of the values in the truth table of f are equal to 1. The *sign function* $\hat{f} : V_n \to R^*$ of a Boolean function f is defined as

$$\hat{f}(x) = (-1)^{f(x)}. \tag{3}$$

The *autocorrelation function* r_f of a Boolean function f on V_n and the autocorrelation function of its sign function $\hat{f}$ are defined as

$$r_f(a) = \sum_{x \in V_n} f(x) \oplus f(x \oplus a). \tag{4}$$

$$\hat{r}_{\hat{f}}(a) = \sum_{x \in V_n} \hat{f}(x) \cdot \hat{f}(x \oplus a) \tag{5}$$

for all $a \in V_n$, while the *correlation value* between two Boolean functions f and g is defined by

$$c(f, g) = 1 - \frac{wt(f \oplus g)}{2^{n-1}}. \tag{6}$$

Based on Equation (4), the value $\frac{1}{2} r_f(a)$ is the number of pairs of function values that are different. The number of pairs of function values that are equal is $2^{n-1} - \frac{1}{2} r_f(a)$. Thus, the positive part of the summation in Equation (5) is equal to $2^n - r_f(a)$ while the negative part is equal to $r_f(a)$. From the above analysis, it follows that $r_f(a)$ can be calculated from $\hat{r}_{\hat{f}}(a)$ according to the following equation:

$$r_f(a) = 2^{n-1} - \frac{1}{2} \hat{r}_{\hat{f}}(a). \tag{7}$$

Let $(a_{n-1}, \ldots, a_0)$ and $(x_{n-1}, \ldots, x_0)$ be the binary representations of a and x correspondingly. Then $x \leq a$ means that $x_i \leq a_i$ for all $1 \leq i \leq n$. The *Algebraic Normal Form (ANF)* of a boolean function f on V_n is a polynomial of the form

$$f(x) = \sum_{a \in V_n} c_a x_{n-1}^{a_{n-1}} \ldots x_0^{a_0} \tag{8}$$

where $c_a \in F_2$ and is the exclusive-or of $f(x)$ for all $x \leq a$, i.e. $c_a = \sum_{x \leq a} f(x)$.

The algebraic degree of f denoted as $deg(f)$ is the maximum number of x_i's in the ANF's monomials with nonzero coefficients. In [8] there is a method that calculates the algebraic degree of a boolean function f from its truth table. In the next sections, we shall see that certain function properties are related to its algebraic degree.

2.1 Nonlinearity of a Boolean function

The *non-linearity* of a function f on V_n is defined by

$$nl(f) = \min_{g \in A_n} wt(f \oplus g) \tag{9}$$

where A_n is the class of all affine Boolean functions on V_n. Intuitively, nonlinearity measures the *minimum distance* of a function f from all affine functions. The computation of the nonlinearity of a function f can be performed efficiently by the use of the *Walsh-Hadamard* transform [2]. The Walsh-Hadamard transform of a function f on V_n is a function $W_f : V_n \rightarrow R^*$ and is defined by:

$$W_f(w) = \sum_{x \in V_n} f(x) \cdot (-1)^{(w \cdot x)}. \tag{10}$$

Alternatively, a function f can be obtained by the inverse Walsh transform which is defined by:

$$f(x) = \frac{1}{2^n} \sum_{w \in V_n} W_f(w) \cdot (-1)^{(w \cdot x)}. \tag{11}$$

The list of the Walsh coefficients in summation (11) defines the Walsh spectrum of f. Moreover, the Walsh-Hadamard transform of the sign function $\hat{f}$ of a Boolean function f on V_n is a function $W_{\hat{f}} : V_n \rightarrow R^*$ and is defined as

$$W_{\hat{f}}(w) = \sum_{x \in V_n} \hat{f}(x) \cdot (-1)^{w \cdot x}. \tag{12}$$

It is easy to see that the factor $(-1)^{w \cdot x}$ in (10) and (12) is the value of the entry (w, x) of the Hadamard matrix H_n of order n, which is defined recursively as follows:

$$\begin{aligned} H_0 &= 1. \\ H_n &= \begin{bmatrix} H_{n-1} & H_{n-1} \\ H_{n-1} & -H_{n-1} \end{bmatrix}. \end{aligned}$$

Based on the above observation, the Walsh Transform of the sign function $\hat{f}$ can be computed by multiplying the truth table of $\hat{f}$, represented by $[\hat{f}(0), \ldots \hat{f}(2^n - 1)]$, with the Hadamard matrix H_n of order n ([2]):

$$W_{\hat{f}} = \begin{bmatrix} \hat{f}(0), \ldots, \hat{f}(2^n - 1) \end{bmatrix} \times \begin{bmatrix} H_{n-1} & H_{n-1} \\ H_{n-1} & -H_{n-1} \end{bmatrix}. \tag{13}$$

Similarly, a boolean function f on V_n can be obtained by multiplying its Walsh coefficients, represented by $[W_f(0), \ldots W_f(2^n - 1)]$, by Hadamard matrix H_n of order n ([2]):

$$f = \frac{1}{2^n} \begin{bmatrix} W_f(0), \ldots W_f(2^n - 1) \end{bmatrix} \times \begin{bmatrix} H_{n-1} & H_{n-1} \\ H_{n-1} & -H_{n-1} \end{bmatrix}. \tag{14}$$

The nonlinearity of a function f can be computed efficiently using the following theorem ([2]):

Theorem 2.1 *The nonlinearity of a Boolean function f on V_n is determined by the Walsh transform of its sign function $\hat{f}$ by the following equation:*

$$nl(f) = 2^{n-1} - \frac{1}{2} \max_{w \in V_n} |W_{\hat{f}}(w)|. \tag{15}$$

It is easy to see that the nonlinearity of a boolean function is a positive integer. Moreover, the Walsh coefficients of the sign function $\hat{f}$ are even integers as the following lemma states.

Lemma 2.1 *The Walsh coefficients of the sign function $\hat{f}$ are even.*

Proof 1 *The number of terms in the summation of Equation (12) is even. If the number p of x where $f(x) \neq w \cdot x$ is odd, then the number q of x where $f(x) = w \cdot x$ is also odd since the number of terms must be even. Consequently, the sum in Equation (12) is equal to $-p + q$, which is an even integer. This integer is negative if $|p| > q$ and positive if $|p| < q$.*

On the other hand, if p is even then q is also even for the same reason as above and the sum in Equation (12) is equal to $-p + q$ which is an even integer either negative if $|p| > q$ or positive if $|p| < q$.

A crucial result for the nonlinearity of a Boolean function is presented in [2] and [4]:

Theorem 2.2 *The nonlinearity of any Boolean function f on V_n satisfies*

$$nl(f) \leq 2^{n-1} - 2^{\frac{n}{2}-1}. \tag{16}$$

Since an SBox is a set of Boolean functions, i.e. its columns, if C is the set of all nontrivial linear combinations of the SBox columns, then the nonlinearity of an SBox S is defined as

$$nl_S = \min_{f \in C} nl(f). \tag{17}$$

2.2 Strict avalanche criterion

Another important property of a Boolean function f on V_n is the *Strict Avalanche Criterion* or SAC for short. A Boolean function f on V_n is said to satisfy SAC if changing one of the n bits in the binary representation of its input x, the function value $f(x)$ changes for exactly half of 2^n possible values of x ([2]). A typical definition of SAC is provided by the following lemma ([2]):

Lemma 2.2 *A boolean function f on V_n is SAC if and only if the function $f(x) \oplus f(x \oplus a)$ is balanced for every $a \in V_n$ with $wt(a) = 1$.*

The SAC property is particularly useful in cryptography since it is easy to see that a small change in the input of a function causes a massive change in the output behavior of the function (avalanche effect). Based on the definition of the autocorrelation function of a Boolean function f and Lemma 2.2 the following result follows ([2]):

Lemma 2.3 *A Boolean function f on V_n is SAC if and only if for the autocorrelation function it holds that $r_f(a) = 2^{n-1}$ $\forall a \in V_n$ with $wt(a) = 1$.*

Based on (7), the criterion that $r_f(a) = 2^{n-1}$ $\forall a \in V_n$ with $wt(a) = 1$ in Lemma 2.3 can be stated differently, as $\hat{r}_{\hat{f}}(a) = 0$ $\forall a \in V_n$ with $wt(a) = 1$. Since the autocorrelation function $\hat{r}_{\hat{f}}$ is fundamental in the verification of the SAC property, an efficient way for its calculation is demanding. In [2] the theorem of Wiener and Khintchine is discussed that correlates the spectrum of the sign function of a Boolean function f with its autocorrelation function:

Theorem 2.3 *A Boolean function on V_n satisfies $W_{\hat{r}_{\hat{f}}}(w) = W^2_{\hat{f}}(w)$ $\forall w \in V_n$.*

From this theorem, a value of the autocorrelation function $\hat{r}_{\hat{f}}$ can be obtained by computing the inverse Walsh transform of $W_{\hat{r}_{\hat{f}}}(w)$ which is, in turn, computed from the Walsh spectrum of the sign function $\hat{f}$. Thus, the calculation of the Walsh transform of the sign function is fundamental for checking both if a function is highly nonlinear and a SAC function.

In [2] a relationship between the degree of a boolean function f and its autocorrelation function is given:

Lemma 2.4 *If boolean function f in $n > 2$ variables has $deg(f) = n$, then $r_f(a) \neq 2^{n-1}$ $\forall a \in V_n$.*

Based on Lemmas 2.3 and 2.4, we can deduce that if a boolean function f on V_n has $deg(f) = n$ then f is not SAC.

A generalization of SAC, the SAC of high order, appears in [2]. A boolean function on V_n is said to satisfy the SAC of order k, $SAC(k)$ for short, if keeping any k of the n bits of the binary representation of input x fixed, the boolean function in the remaining $n-k$ variables is SAC. It is easy to see that a $SAC(0)$ function is a function that satisfies SAC.

An interesting result presented in [2] relates the algebraic degree of a boolean function to the $SAC(k)$ criterion:

Theorem 2.4 *Let f a boolean function on V_n where $n > 2$. If f satisfies $SAC(k)$, $0 \leq k \leq n-3$, then $2 \leq deg(f) \leq n-k-1$. If f satisfies $SAC(n-2)$ then $deg(f) = 2$.*

Based on this theorem, if the algebraic degree of a function f does not satisfy the above requirements then the function can be safely discarded because it is not $SAC(k)$. If by $|SAC(n-3)|$ we denote the number of $SAC(n-3)$ boolean functions on V_n and by $|BSAC(n-3)|$ we denote the number of balanced $SAC(n-3)$ boolean functions on V_n, then the following holds ([2]):

$$\lim_{n\to\infty} \frac{|BSAC(n-3)|}{|SAC(n-3)|} = 1 \tag{18}$$

which means that, as the number of boolean variables n increases, all functions that are $SAC(n-3)$ are balanced.

2.3 Propagation criteria and correlation immunity

The last two crucial properties of a Boolean function which are related to information leakage from the knowledge of certain of its values, are the Propagation Criteria and the Correlation Immunity.

A Boolean function f on V_n satisfies the propagation criterion of degree k, or f is a $PC(k)$ function, if changing any i of the n bits of the binary representation of input x, where $1 \leq i \leq k$, changes the function values for exactly half of the 2^n possible values of x. The propagation criterion of degree k can be defined by the autocorrelation function as follows ([2]):

Lemma 2.5 *A boolean function f on V_n is $PC(k)$ if and only if the autocorrelation function $r_f(a)$ is equal to 2^{n-1} $\forall a \in V_n$ with $1 \leq wt(a) \leq k$.*

It is easy to see that for a Boolean function f that is $PC(k)$ the autocorrelation function of $\hat{f}$ is equal to 0 for all a such that $1 \leq wt(a) \leq k$. In information theoretical terms, the

information that leaks for $f(x)$, knowing $f(x \oplus a)$ for any a with $1 \leq wt(a) \leq k$, is zero. An interesting result which appears in [2] states that a $PC(n)$ function $f(x)$ with $n > 2$ must have an even degree and $deg(f) \leq \frac{n}{2}$.

Another property related to information leakage is the correlation immunity. A boolean function f is correlation immune of order k ($1 \leq k \leq n$) if knowing the value of f for a certain x, the probability of having a certain value at a fixed set of k bits of the binary representation of x is equal to 2^{-k}, regardless of which k bits have been chosen. The following Lemma presents a characterization of the correlation immunity of a boolean function ([2]).

Lemma 2.6 *A function $f(x)$ on V_n is correlation immune of order k, $1 \leq k \leq n$, if and only if all of the Walsh transforms*

$$W_{\hat{f}}(w) = \sum_{x \in V_n} (-1)^{f(x) \oplus x \cdot w}, \ 1 \leq wt(w) \leq k \tag{19}$$

are equal to zero.

The correlation value between a Boolean function $f(x)$ and a linear function $l_w(x) = w \cdot x$, for a function w as defined by (6), can be expressed by ([2]):

$$c(f(x), l_w(x)) = 2^{-n} W_{\hat{f}}(w). \tag{20}$$

From Equation (20) and Lemma 2.6, we observe that a correlation immune function should have correlation value equal to 0, for any w. However, Meir and Staffelbach in [2] showed that this is impossible:

Lemma 2.7 *For any Boolean function $f(x)$, the total square correlation of $f(x)$ with the set of all linear functions is equal to one, that is,*

$$\sum_{w \in V_n} c(f(x), w \cdot x)^2 = 1. \tag{21}$$

Therefore, for some linear functions the correlation value is equal to 0 while for others this is not the case. Consequently, during the construction process of an SBOX it is better to find Boolean functions where the largest possible value of $|W_{\hat{f}}(x)|$ is as small as possible.

2.4 Bent functions

Rothaus in [4] defined a class of Boolean functions f on V_n with n even, such that

$$W_{\hat{f}}(w) = \pm 2^{\frac{n}{2}}, \forall w \in V_n.$$

This is the class of *bent* Boolean functions and is denoted by B_n. It follows from the definition that the *energy* spectrum coefficients $W^2_{\hat{f}}(w)$ are all the same.

A Boolean function f on V_n has *linear* structures if the function $g(x) = f(x) \oplus f(x \oplus a)$ is constant for a nonzero $a \in V_n$. In [2], the definition of perfect nonlinear functions is presented, as given by Meier and Staffelbach:

Definition 2.1 *A Boolean function f on V_n is called perfect nonlinear if for every nonzero $a \in V_n$ the values $f(x \oplus a)$ and $f(x)$ are equal for exactly half of the values of arguments $x \in V_n$.*

Rothaus discovered in [4] that the concept of perfect nonlinearity is equivalent to the bent property of Bent functions. Thus, the nonlinearity of a Bent function is given by

$$nl(f) = 2^{n-1} - 2^{\frac{n}{2}-1}.$$

This is the maximum nonlinearity according to theorem 2.2. For example, the nonlinearity of a bent function f on V_8 is equal to to $nl(f) = 120$.

Moreover, based on the definition of a perfect nonlinear function, Lemma 2.5, and the fact that if f is a Bent function then $h(x) = f(x) \oplus f(x \oplus a)$ is a balanced function for all nonzero $a \in V_n$, we can deduce that a Bent Boolean function is also a $PC(n)$ function. In [2], the definition for the *dual* of a Bent boolean function is given:

Definition 2.2 *For a Bent Boolean function f, its dual function F is defined by*

$$\hat{F}(w) = (-1)^{F(w)} = \frac{W_{\hat{f}}(w)}{2^{\frac{n}{2}}}.$$

With respect to the algebraic degree of a Bent function, the following important theorem and its corollary are given in [2]:

Theorem 2.5 *The degree of a Bent function f on V_n for $n > 2$ is at most $\frac{n}{2}$.*

Corollary 2.1 *If a boolean function f is Bent of $deg(f) = \frac{n}{2}$, then its dual F is also a Bent Boolean function of degree $\frac{n}{2}$.*

The first methods for the construction of Bent functions were given by Rothaus ([4]), Maiorana ([3] and [2]), McFarland ([2]), Dillon ([2]) and Dobbertin ([2]) while new methods have been described by Adams and Tavares in [1] and by Carlet in [2].

In [2], the Maiorana-McFarland function (MM) is defined to be any function $f(x, y)$ on V_n of the form:

$$f(x, y) = \phi(x) \cdot y \oplus g(x) \tag{22}$$

where $x = (x_{s-1} \dots x_0)$ and $y = (y_{t-1} \dots y_0)$ such that $n = s+t$, $\phi(x)$ is any mapping from V_s to V_t and $g(x)$ is any function on V_s. A Maiorana-McFarland function is cryptographically strong if function $\phi(x)$ is a nonlinear function.

A function $f(x, y)$ defined by Equation (22) is a $PC(k)$ function if the conditions stated by the following theorem holds ([2]):

Theorem 2.6 *A function $f(x, y)$ satisfies $PC(k)$ if*

1. *for each j, $1 \le j \le k$. the linear combination of any j of the coordinate boolean functions $\phi_i(x)$, $1 \le i \le t$ and $\phi(x) = (\phi_1(x) \dots \phi_t(x))$, is balanced.*
2. *for every nonzero a in V_s with $wt(a) \le k$, and for any $x \in V_s$, we have $\phi(x \oplus a) \neq \phi(x)$.*

If $s = t = \frac{n}{2}$ and $\phi(x)$ is any permutation $\pi(x)$ on $V_{\frac{n}{2}}$, the Maiorana-McFarland function is a Bent function. It is obvious that a permutation in $V_{\frac{n}{2}}$ satisfies the conditions of the above theorem and, thus, the corresponding function $f(x, y)$ is a $PC(k)$ function. From a cryptographic point of view it is better for this permutation to be an Almost Perfect Nonlinear permutation (APN). A permutation $\phi(x)$ on $V_{\frac{n}{2}}$ is APN if and only if the function $g(x, a) = \phi(x) \oplus \phi(x \oplus a)$ takes exactly $2^{\frac{n}{2}-1}$ different nonzero values and each of them appears twice when x runs over $V_{\frac{n}{2}}$ for each nonzero a ([11]).

3 Distributed SBox computation

An SBox of m columns can be constructed, sequentially, as follows. Starting with an empty SBox, a function $f(x)$ from V_n that has certain properties, i.e. certain value range for nonlinearity, the SAC property and the desired algebraic degree, is computed as the first SBox column. Next, a new function with the same class of properties is computed. If the linear combination of these first two functions has certain desirable properties (e.g. the nonlinearity and algebraic degree of this linear combination is within certain bounds) then the function is accepted as the second SBox column. As more column functions are added, in this fashion, all possible linear combinations of each new function is computed with previous ones and check for preservation of the target properties. This procedure is repeated until m columns have been produced. It is obvious that the procedure is computationally intensive since for large m the number of linear combinations that should be checked, each time a new function is inserted, is very large (equal to 2^m). Moreover, the process is likely to be computationally intractable as n and m grow since every linear combination of the SBox columns appears that has to be checked with respect to the desirable properties. In addition, these properties may also themselves be hard to check.

For the generation of the SBox columns a number of methods has been proposed in [2]. We have adopted the generation method of Maiorana-McFarland Bent functions, as described in Sect. 2.4 as well as the generation method that is based on the linear combination of certain Maiorana-McFarland bent functions which are SAC and have sufficiently high nonlinearity. In order to check every linear combination of the Boolean functions within the SBox, a mechanism for the computation of every linear combination should be employed. In the sequential SBox construction algorithm, we employed a slightly modified version of the sequential *lazy counting* algorithm described in [3] for the linear combinations generation:

Function SBox_Construction
Input: n, m, $functionClass$ and $PropertiesSet$
Output: $SBox$
$nc = 1$;
While ($nc < m$) do
    begin
        $SBox[nc] = generateFunction(n, functionClass)$;
        PropertiesSatisfied=TRUE;
        $g = 0$;
        $i = 1$;
        While ($i < 2^{nc}$ AND PropertiesSatisfied) do
            begin
                If $b_{nc-1} \dots b_0$ is the binary reprsentation of i
                find the k-th bit position from right where $b_k = 1$
                $g = g \oplus C_k$;
                PropertiesSatisfied=validateLinearCombination(g, $PropertiesSet$);
                $i = i + 1$;
            end
        if PropertiesSatisfied
            begin
                $nc = nc + 1$;

```
        end
    end
```

The number of linear combinations that should be checked by the sequential SBox computation method is very large even for small values of the counting variable nc. For instance, when the tenth column is about to enter SBox, i.e. when $nc = 10$, there are 1024 linear combinations that should be checked. This check requires the validation of a number of desirable properties, a task which may be highly time consuming. Thus, performing all these checks sequentially on a single processor based on the sequential lazy counting algorithm alone, slows down considerably the whole SBox computation process.

The SBox computation process can be accelerated by parallelizing the lazy counting algorithm. This can be achieved by partitioning the space of all 2^{nc} possible linear combinations into B non-overlapping subsets of linear combinations, each of them assigned for checking to a different processor from a pool of B available processors.

The space of linear combinations can be efficiently partitioned into B non-overlapping sets based on Lemma 3.1. This lemma provides a relationship between the columns of the SBox that are involved in the $i-th$ linear combination and the binary representation of the value of the index variable i of the SBox computation algorithm described above (the variable i enumerates the linear combinations of the nc SBox columns).

Lemma 3.1 *If $b_{nc-1}b_{nc-2}\dots b_0$ is the binary representation of the value of the variable i that enumerates the number of all possible linear combinations of the nc columns of an SBox then the following holds:*

(i) If l is the leftmost bit position of i that $b_l = 1$ $(0 \le l \le nc-1)$, then the i-th linear combination of the nc SBox columns contains the column C_l while it does not contain the columns C_k for $k > l$.

(ii) For any bit position l, $0 < l \le nc-1$, if $b_l \ne b_{l-1}$ then the i-th linear combination of the nc SBox columns contains the column C_{l-1} otherwise it does not contain it.

Proof 2 *We will proceed using induction on nc. We will, also, denote by S_i^{nc} the set of columns involved in the i-th linear combination of nc SBox columns generated by the sequential lazy counting method in the above SBox computation method.*

For $nc = 1$, (i) and (ii) hold as i and l assume only the value 1 and $S_i^{nc} = \{C_0\}$. Let us assume that they also hold for $nc \le nc_0$. We will prove that they hold for $nc = nc_0 + 1$. Note that if the leftmost bit position of i equal to 1 is position $l \le nc_0 - 1$, (i) and (ii) hold from the induction hypothesis. Thus, we only have to prove them in the case where $l = nc_0$.

The column C_{nc_0} belongs to the current linear combination since when the algorithm examines the first value of i such that the least significant bit position equal to 1 in the $l = nc_0$-th position, it includes C_{nc_0} to the current linear combination. Thus (i) holds.

In order to prove (ii), it suffices to consider the value $l = nc_0$ and only when this position is equal to 1 since in the other cases (ii) holds by induction hypothesis. If $b_{l-1} = 0$, while $b_l = 1$, then this means that column $nc_0 - 1$ has been considered for inclusion in the current linear combination in a previous step i. Thus, column C_{l-1} is involved to the linear combination.

On the other hand, if $b_{l-1} = 1$ then C_{l-1} has been considered twice in the current linear combination: once when the l-th bit was equal to 0 and $(l-1)$-th bit was equal to 1 and secondly when later the l-th bit became 1 and after some steps the $l-1$ bit became 1. But

this means that C_{l-1} appears (implicitly) twice in the current linear combination and as the operation is XOR, C_{l-1} vanishes and, thus, it effectively disappears from the current linear combination. This completes the proof of the inductive step and the proof of the lemma. □

Based on Lemma 3.1, the set of the 2^{nc} linear combinations of SBox columns can be partitioned into B equally sized subsets of $\frac{2^{nc}}{B}$ linear combinations each, where each subset is determined by the index x_j $(1 \leq j \leq B)$ of its first (i.e. lowest index) linear combination. The linear combination that corresponds to x_j is equal to the linear combination of the columns C_k for $k \leq l$, due to case (i) of the lemma, and b_k satisfies the condition given by case (ii), where l is the bit position of the most significant bit of the binary representation of x_j in which $b_l = 1$.

Thus, our distributed lazy counting algorithm is based on partitioning the space of linear combinations of SBox columns to several processors each time a new column is generated. More specifically, each time a new SBox column is computed the space of linear combinations is partitioned into B subsets, where B is the number of available processors. The subset of linear combinations S_j and its corresponding index x_j are assigned to the j-th processor, which calculates the first linear combination LC_{x_j} of its assigned subset. The LC_{x_j} can be computed by the following function:

```
Function Initial_LinearCombination
Input:  x_j
Output: LC_{x_j}
begin
    x = x_j;
    /*find the MSB of the binary representation of x that is equal to 1*/
    l = MSB_EQ_1(x);
    y = SBox[l];
    for (k = l − 1 downTo 0) do
       begin
          /*Check for case (ii) of Lemma*/
          if ((x & (1 << k)) != (x & (1 << l))) then y⊕ = SBox[k];
          l = l − 1;
       end
    LC_{x_j} = y;
end
```

Starting from the initial linear combination LC_{x_j}, every linear combination in the subset can be computed by setting $i = x_j$ and $g = LC_{x_j}$ in the sequential SBox computation algorithm presented above. After setting these variables, each processor enumerates linear combinations and checks whether they satisfy the target properties. If every linear combination satisfies the desirable properties, then the generated Boolean function is accepted otherwise it is discarded. The calculation of the initial linear combination LC_{x_j} should be repeated each time a new column is generated, since the number of total linear combinations is doubled. This process is repeated until all m SBox columns, whose linear combinations satisfy the target properties, have been generated.

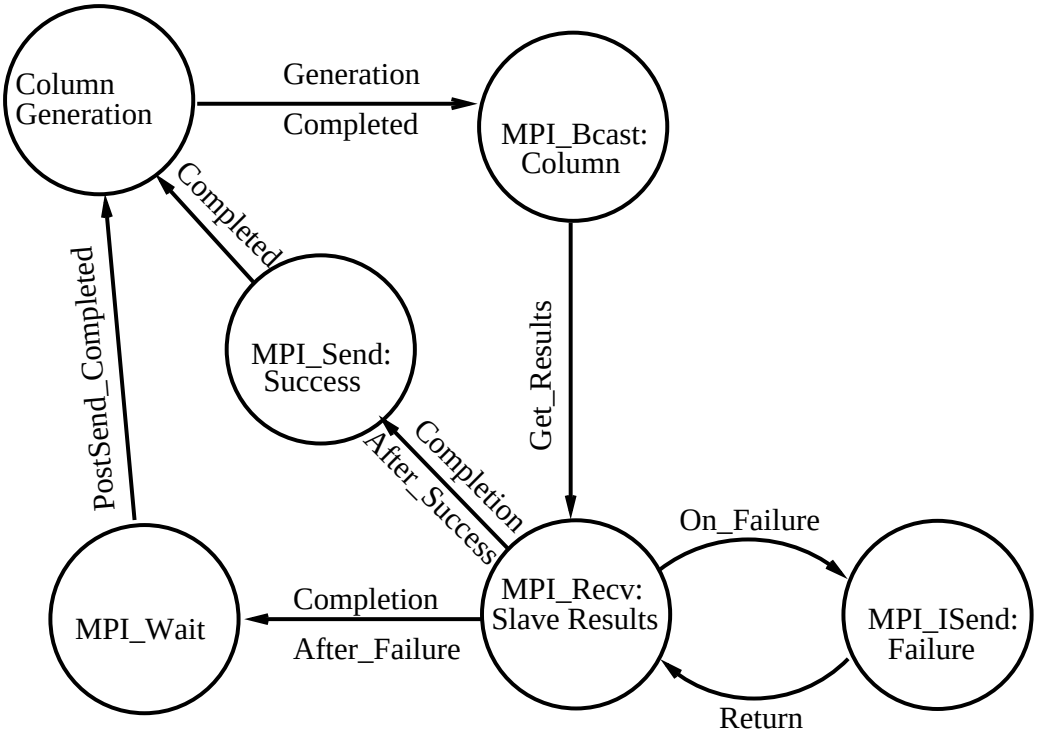

Figure 1. Master state diagram.

4 Implementing the distributed algorithm on a computing cluster

The domain of the distributed algorithm described in the previous section is the set of all possible linear combinations of the SBox Boolean functions. Based on Lemma 3.1, the domain decomposition programming model and the message passing programming paradigm are suitable for the implementation of the distributed algorithm on a computer cluster. We have implemented the distributed algorithm described in Section 3 using the *Message Passing Interface (MPI)* (see [6] for details on MPI). The algorithm was ran on the high performance computing cluster named *Pythagoras* of the Department of Mathematics of the Aegean University. The operating system environment of Pythagoras is the *Rocks Cluster Distribution* (version Chimichanga) on the CentOS Linux distribution. There are 32 processor cores available for computations distributed in 5 computing nodes and 4 cores for managing the computing cluster at the front-end node. Each node has a secondary storage while both the computing nodes and the front-end node share a filesystem through *Network File System* (NFS). The employed job scheduler is the Sun Grid Engine.

Assuming that $nProcs$ processor cores have been assigned to the SBox distributed computation, one of them is designated as the master of the computation while the remaining $nProcs - 1$ processor cores are the slaves. Initially, the master computes $IN = \lceil log_2(nProcs - 1) \rceil$ cryptographically strong Boolean functions, all linear combinations of whose satisfy a set of target properties (e.g. nonlinearity greater than a given value, SAC property, etc). This initial subset of SBox columns is saved in a file and the master broadcasts a message to the $nProcs - 1$ slaves in order to trigger them to read the file through NFS. Each slave, while waiting to receive a new function from the master, increments IN

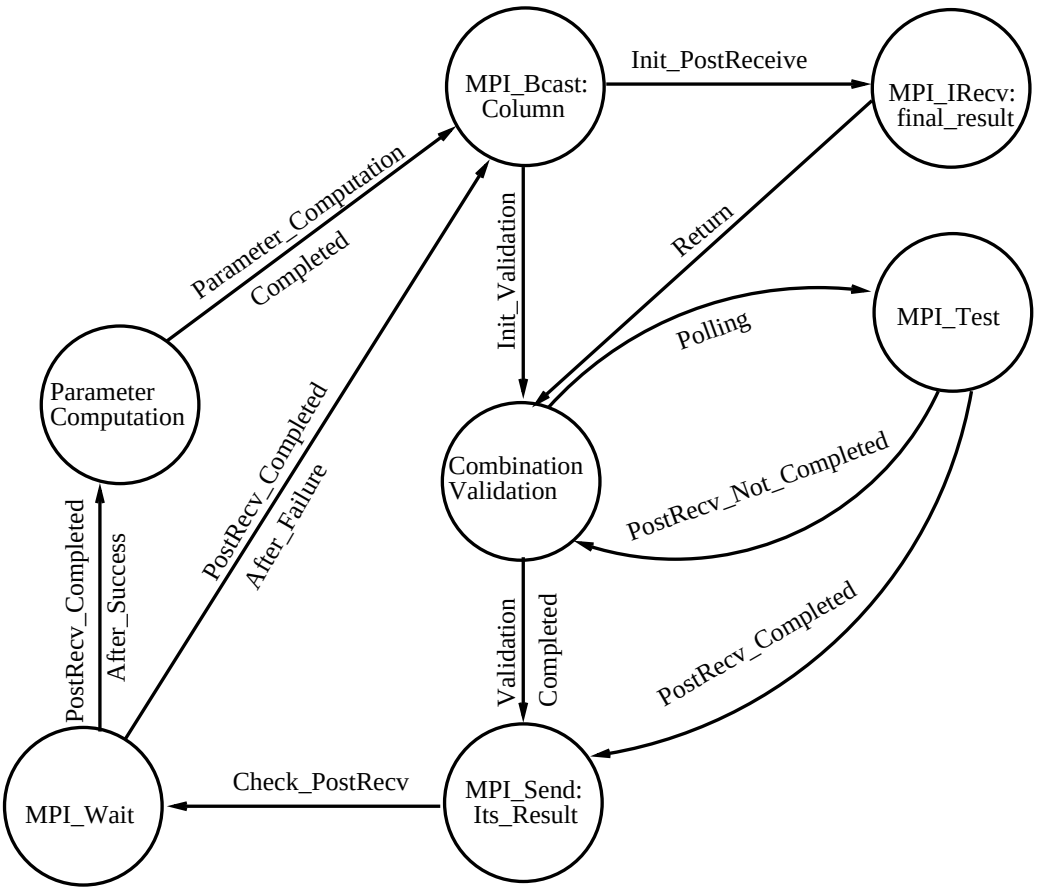

Figure 2. Slave state diagram.

i.e. it sets $nc = IN + 1$, divides nc by $nProcs - 1$ in order to determine the size of its subset of linear combinations and then it multiplies this number by its identification number Id in order to find the offset of its first linear combination. Then it waits for receiving a new Boolean function for evaluation.

Figures 1 and 2 show the communication and computation steps of both the master and the slave processor cores during the SBox computation. After the completion of the initial phase described above, the master generates a new cryptographically strong Boolean function and broadcasts it to slaves (Column Generation and $MPI\_Bcast : Column$ in Figure 1). After receiving the column, each slave initiates a nonblocking receive process in order to get a message from the master that designates the final result of the validation procedure and starts the validation of its linear combinations (states labeled Combination Validation and $MPI\_IRecv : final\_result$ in Figure 2). The master is in the state labeled $MPI\_Recv : SlaveResults$ where it gets the results of the validation procedure from every slave processor. Each slave polls the status of the nonblocking receive process during certain intervals depending on the number of linear combinations. For large numbers of linear combinations the polling takes place after the validation of 5 percent of the total number of linear combinations while for small numbers of linear combinations the polling takes place after the validation of 50 percent of the total number of linear combinations.

If a slave processor, while enumerating its linear combinations, discovers a linear combi-

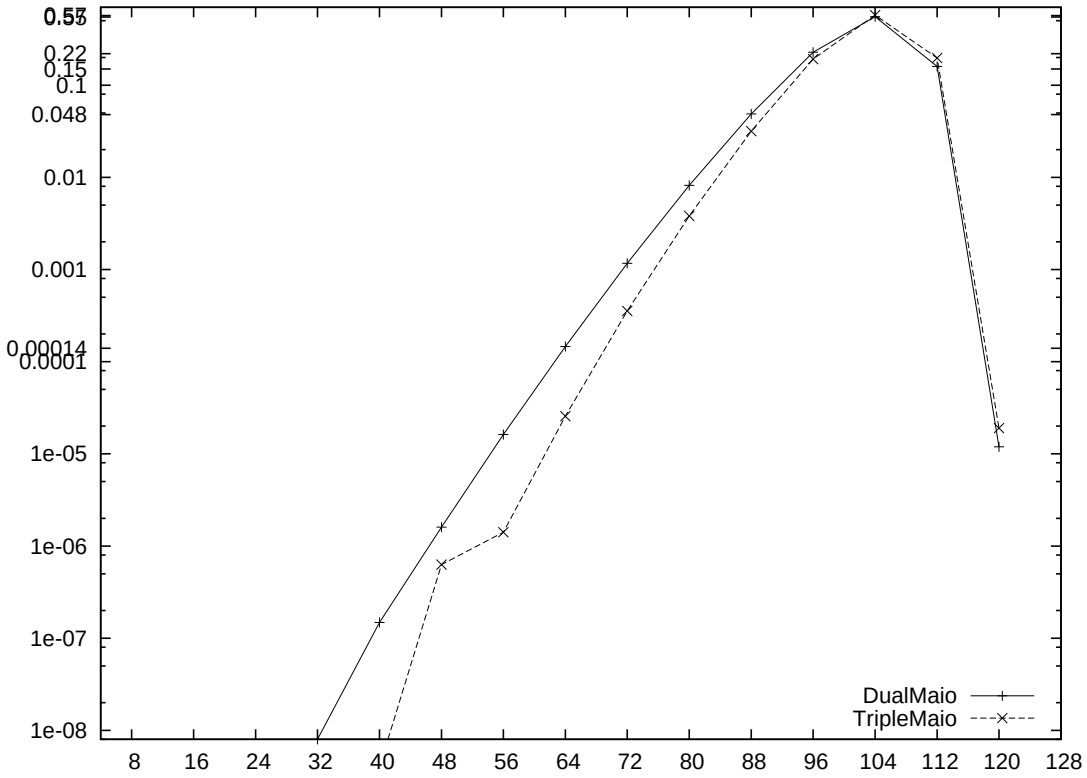

Figure 3. Probability distribution of random variables NL_1 and NL_2 which are the nonlinearities of the linear combinations of two and three randomly selected MM bent functions respectively

nation that does not satisfy the target properties, it completes the validation procedure prematurely and sends a failure message to the master (state labeled $MPI\_Send : its\_Result$ in Figure 2). On the other hand, if every linear combination of a slave processor satisfies the properties it sends a success message to the master. After sending its result, a slave waits for the final decision of the master (state labeled MPI_Wait in Figure 2). If it receives a success message, it recalculates the number of its linear combinations and the index of the first linear combination, since now $nc = nc + 1$ (state labeled $Parameter\ Calculation$ in Figure 2), and waits for the next column, while in case of a failure it simply waits for the next column without any recalculation.

The master, after receiving a failure message from a slave, sends a failure message to all slaves through a nonblocking send process (state labeled $MPI\_ISend : Failure$ in Figure 1) since there is no reason to enumerate and check any more linear combinations. After receiving the failure message through the polling process described above, the slaves terminate their validation procedure and send a failure message to the master, waiting for synchronization. The master, after receiving a message from every slave, waits for the nonblocking send to be completed and discards the function caused the failure. On the other hand, if the master receives a success message from every slave, then it sends a success message to every slave through a blocking send (state labeled $MPI\_Send : Success$ in Figure 1). After its completion, the master starts the generation of the next SBox

Table 1. Performance of the computation of SBoxes of various sizes for certain nonlinearity using MM class of functions

SBox Size	NL	cores=1	cores=4		cores=8		cores=16	
		time	time	speedup	time	speedup	time	speedup
8×4	112	711.84	509.89	1.4	-	-	-	-
8×6	104	83.85	26.45	3.17	23.85	3.52	-	-
8×9	96	265.58	130.21	2.04	98.67	2.69	85.16	3.12
8×12	88	544.19	130.85	4.16	66.22	8.22	37.27	14.6

column. The described distributed computation is repeated until all the m SBox columns have been computed.

5 Experimental results

Since the class of MM bent functions presented in Sect. 2.4 have maximum nonlinearity and satisfy the SAC property, it was selected for the evaluation of our distributed SBox computation algorithm. Functions on V_8 were selected for the evaluation of the algorithm, since the space of more general cryptographically secure Boolean functions for $n > 8$ can not be explored exhaustively. For the computation of an MM bent function, a permutation $\phi(x)$ on V_4 and a Boolean function $g(x)$ on V_4 are chosen at random.

Before using this class of functions, the properties of linear combinations of two and three MM bent functions were examined with respect to nonlinearity and the satisfaction of the SAC property. The probability distributions of the random variables NL_1 and NL_2, which represent the nonlinearities of the linear combinations of two randomly selected MM bent functions and three randomly selected MM bent functions respectively are graphically presented in Figure 3. These distributions were produced after the examination of 10^9 pairs and triples of MM bent functions, a process which required 27 days. The most probable values for the nonlinearity of a linear combination of two and three MM bent functions are $96, 104, 112$ with 104 having the highest probability. Moreover, these probabilities are higher for the linear combinations of three MM functions and about 0.5% of 10^9 linear combinations also satisfy SAC. Thus, we constructed a subclass of Boolean functions on V_8 named TMM where each function is the linear combination of three MM bent functions with nonlinearity greater than 96 while it also satisfies SAC and its algebraic degree is 4 (i.e. $\frac{n}{2}$).

From the above description, it follows that the computation of an MM and a TMM function is a probabilistic procedure, since the components of each function are randomly selected. In particular for the computation of a TMM function we choose randomly three MM functions and, then, their linear combination is checked for the satisfaction of the the target nonlinearity value and the SAC property. This implies that the execution time of the function computation procedure can be highly irregular.

For example, running our computation procedure on one processor core using functions from the TMM class, the computation of an SBox of size 8×3 that has nonlinearity 104 (close to the maximum nonlinearity which is 120) and every linear combination of its columns satisfies SAC, took almost 8 days while running it on two processor cores required one and a half day only, which is a considerable improvement.

Table 2. Performance of the computation of SBoxes of various sizes for certain nonlinearity using TMM class of functions

SBox Size	NL	cores=1	cores=4		cores=8		cores=16	
		time	time	speedup	time	speedup	time	speedup
8×6	104	7013.85	2697.72	2.6	3223.42	2.18	–	–
8×8	96	134.27	98.14	1.37	51.15	2.63	–	–
8×9	96	22077.39	17007.22	1.3	6969.54	3.17	5814.64	3.8

Using functions from the MM class, the computation of an SBox with the same size and satisfying the same set of properties took 2450.53 secs using one processor core and 97.66 secs using two processor cores. Due to this variability in execution times, every experiment was run 10 times and the final execution time of the experiment was the average of the individual execution times.

Tables 1 and 2 present the execution times, in seconds, of the computation of SBoxes of various sizes and the corresponding nonlinearity obtained. They also show the achieved speedup over the sequential execution, i.e. running the same computation procedure on one processor core without partitioning the linear combination space to more processor cores.

Table 1 presents the results obtained using only MM functions while Table 2 presents the results obtained by the use of only TMM functions. In the last Table 3, the maximum percentage of linear combinations that satisfy SAC is presented for certain SBox sizes and for both TMM and MM function classes. From a cryptographic point of view, both the MM and TMM classes lead to SBoxes with similar properties.

As for the use of TMM functions, due to the fact that the execution time of the function computation procedure is irregular, we have the unexpected result that the speedup obtained with 8 processor cores for an SBox of size 8×6 with nonlinearity 104, is less than the speedup obtained with 4 processor cores. This occurred because the function computation procedure is a highly irregular and computationally intensive task while each subset of linear combinations, per processor core is, relatively small (each of the 6 processor cores enumerates 9 combinations while the remaining one enumerates 10 combinations). Consequently, the function computation procedure overhead, the cardinality of each linear combination set, and the computational intensiveness of checking each linear combination against the target properties determine the number of processor cores that could provide the maximum efficiency and speedup. This means that the optimal number of processor cores for the computation of a cryptographically strong SBox depends on the above parameters of the proposed SBox distributed computation algorithm.

6 Conclusions

In this paper we have presented a distributed algorithm for the computation of SBoxes with certain security properties and its implementation on a computing cluster. It was demonstrated that it can accelerate considerably the SBox computation process, which is a highly computationally intensive task. Based on this distributed algorithm, a fully parametric platform for the computation of a secure cipher SBox has been implemented. It uses as an input any class of Boolean functions, the SBox size parameters n and m and the properties that the SBox should satisfy. It can be employed as a tool by block cipher designers

Table 3. The maximum percentage of linear combinations that satisfy SAC for certain SBox sizes and for both TMM and MM class of functions

SBox Size	MM	TMM
8×4	40	-
8×6	12.69	12.69
8×8	-	3.9
8×9	2.9	3.1
8×12	0.98	-

and it can be used for both the construction of secure SBoxes in a reasonable amount of time as well as the exploration of SBox properties containing, as columns, functions from any Boolean function class.

As an interesting future research goal we plan to compute SBoxes using more general Boolean function classes and explore the properties of the resulting SBoxes. This task will be speeded up considerably by the fast construction processing times offered by the described distributed algorithm. Moreover, for this paper, each Boolean function was computed at random. We intend to investigate possible improvement over the random selection by selecting the optimal function from within a set of Boolean functions satisfying certain formal and heuristic criteria.

References

[1] C.M.Adams and Tavares, S., Generating and counting binary bent sequences. IEEE Transactions on Information Theory IT-36, pp. 1170-1173, **(1990)**.

[2] T.W. Cusick and P. Stănică, Cryptographic Boolean Functions and Applications. Academic Press Elsevier **(2009)**.

[3] S. Mister and C. Adams, Practical sbox design. In Third Annual Workshop on Selected areas in Cryptography **(1996)**, Kingston Ontario.

[4] O. Rothaus, On bent functions. Journal of Combinatorial Theory 20(A), pp. 300-305, **(1976)**.

[5] J. Seberry, X. M. Zhang, and Y. Zheng, Systematic generation of cryptographically robust s-boxes. In Proceedings of the First ACM Conference on Computer and Communications Security. **(1993)**, pp. 172-182.

[6] M. Snir, S. Oto, S. Huss-Lederman, D. Walker and J. Dongarra, MPI: The Complete Reference. The MIT Press, Cambridge, Massachusetts **(1996)**.

[7] W. Stallings, Cryptography and Security: Principles and Practice. Prentice Hall **(1999)**.

[8] FJ Mac Williams and NJA Sloane, The Theory of error-correcting codes. Amsterdam:North-Holland Publishing Company, **(1978)**.

[9] P. Nastou and Y. Stamatiou, Embedded Cryptographic Hardware:Methodologies and Architectures, chapter VIII. Nova Publisher, **(2004)**.

[10] Hans Dobbertin and Gregor Leander, A survey of some recent results on bent functions. In SETA 2004, LNCS 3486, Spinger-Verlag Berlin Heidelberg, **(2005)**, pp. 1-29.

[11] T. Bent and C. Ding, An almost perfect nonlinear permutations. In Advances in Cryptology-EUROCRYPT '93, LNCS 765, Spinger-Verlag Berlin Heidelberg, **(1994)**, pp. 65-76.

[12] J. Lee and H.M. Heys and S.E. Tavares, Resistance of a CAST-Like Encryption Algorithm to Linear and Differential Cryptanalysis, Designs, Codes and Cryptography, Vol. 12, pp. 300-305, **(1997)**.

Authors

Dr Panayotis Nastou is Lecturer of the Applied Mathematics and Engineering within the Department of Mathematics of Aegean University, Samos, Greece. His research interests lie in Wireless Sensor Networks, Embedded Systems Design, Cryptography, Distributed Computing, Design and analysis of algorithms and Combinatorial Optimization.

Dr Yannis Stamatiou is Associate Professor at the Department of Business Administration at the University of Patras, Greece. His main research interests lie in theoretical and applied cryptography, ICT security, sensor network security, and eGovernment applications with a focus in eVoting.