



Article

# BoolSPLG: A Library with Parallel Algorithms for Boolean Functions and S-Boxes for GPU

Dushan Bikov <sup>1</sup>, Iliya Bouyukliev <sup>2,\*</sup> and Mariya Dzhumalieva-Stoeva <sup>3</sup>

<sup>1</sup> Faculty of Computer Science, Goce Delchev University, 2000 Stip, North Macedonia; dusan.bikov@ugd.edu.mk

<sup>2</sup> Institute of Mathematics and Informatics, Bulgarian Academy of Sciences, 1113 Sofia, Bulgaria

<sup>3</sup> Faculty of Mathematics and Informatics, University of Veliko Turnovo, 5003 Veliko Turnovo, Bulgaria; m.dzhumalieva@ts.uni-vt.bg

\* Correspondence: iliyab@math.bas.bg;

**Abstract:** In this paper, we present a library with sequential and parallel functions for computing some of the most important cryptographic characteristics of Boolean and vectorial Boolean functions. The library implements algorithms to calculate the nonlinearity, algebraic degree, autocorrelation, differential uniformity and related tables of vectorial Boolean functions. For the sake of completeness, we provide the mathematical basis of these algorithms. Furthermore, we compare the performance of the parallel functions from the developed software with the corresponding sequential functions and with analogous functions from the well-known SageMath and SET packages. Functions from BoolSPLG can be used to develop efficient algorithms for constructing Boolean and vectorial Boolean functions with good cryptographic properties. The parallel part of the library is implemented using a CUDA parallel programming model for recent NVIDIA GPU architectures. BoolSPLG is an open-source software library written in CUDA C/C++ with explicit documentation, test examples, and detailed input and output descriptions of all functions, both sequential and parallel, and it is available online.

**Keywords:** mathematical software; CUDA C; CUDA library; GPU; butterfly algorithms; cryptographic properties; vectorial Boolean function

**MSC:** 94D10; 90-04; 68W10



**Citation:** Bikov, D.; Bouyukliev, I.; Dzhumalieva-Stoeva, M. BoolSPLG: A Library with Parallel Algorithms for Boolean Functions and S-Boxes for GPU. *Mathematics* **2023**, *11*, 1864. <https://doi.org/10.3390/math11081864>

Academic Editor: Ximeng Liu

Received: 15 March 2023

Revised: 10 April 2023

Accepted: 12 April 2023

Published: 14 April 2023



**Copyright:** © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction

The main subjects which we consider in this paper are Boolean and vectorial Boolean functions (S-boxes) with good cryptographic properties. There is a substantial body of research on S-boxes with eight or fewer variables that are embedded in the most popular ciphers [1], but not much is known about larger S-boxes, despite the interest in them [2–4]. One of the reasons for this is the computationally difficult evaluation of their cryptographic properties. The construction of such types of objects is very important, but in most cases it is also a computationally expensive issue.

Some of the construction methods are based on checking for proper parameters between huge amounts of candidates [5]. The cryptographic parameters which we investigate in this paper are nonlinearity, algebraic degree, autocorrelation, and differential uniformity. The computation of these parameters is related to Fourier-type transforms such as Walsh–Hadamard and Möbius (Reed–Muller) transforms [6,7]. The algorithms, known as butterfly algorithms, that implement these fast, discrete transforms are very efficient [8]. Moreover, these algorithms are suitable for parallelization in SIMD (single instruction, multiple data) computer architectures.

For this type of parallelization, using modern graphics processing units (GPUs) together with CUDA (compute unified device architecture) [9] is natural and very effective.

GPUs are usually specialized for manipulating high-resolution computer graphics, but their structure makes them suitable for processing large amounts of data. This feature of GPUs is a great advantage for deep learning [10], neural systems [11], molecular modeling [12], etc.

For this paper, we developed a library BOOLSPLG with C/C++ functions. It can be used to study important cryptographic properties of Boolean functions with  $n$  variables and bijective  $n \times n$  S-boxes for  $n \leq 20$ . BoolSPLG computes the following cryptographic parameters of the Boolean and vectorial Boolean functions: the Walsh spectrum of a Boolean function, the linearity of a Boolean function, the Walsh Spectrum of an S-box, the linear approximation table of an S-box, the linearity of an S-box, the autocorrelation spectrum of a Boolean function, the autocorrelation of a Boolean function, the autocorrelation spectrum of an S-box, the autocorrelation of an S-box, the algebraic normal form of a Boolean function, the algebraic normal form of an S-box, the algebraic degree of a Boolean function, the algebraic degree of an S-box, the difference distribution table of an S-box, and the differential uniformity of an S-box and a component function of an S-box. All of the basic functions have two versions—sequential and parallel. All these features can be used for project development by anyone who knows the C/C++ programming language but is not so familiar with CUDA C.

The effectiveness of the developed library is based on the optimal use of the capabilities of the GPU architecture and on the properties of the CUDA platform as well. Building an optimized and portable parallel library requires various strategies so that it can be used not only for computing the properties of Boolean and vector Boolean functions, but also for implementation in another software. We will focus on some of the basic features for building the library.

First, developing a parallel library requires an efficient consecutive algorithm which is suitable for parallel implementation. Our research is focused on the cryptographic properties of vector Boolean functions. These are mainly calculated by butterfly algorithms, and such methods are appropriate for parallelization.

Second, GPUs and the CUDA platform have a great advantage—they possess huge computing power for large amounts of data. However, implementing optimized software is a complex task. One of the main problems is related to the transfer of data from the main memory to the GPU memory and vice versa. This is a time-consuming process, sometimes commensurate with the calculation time. Therefore, each of the functions is tailored to this fact. If the algorithm consists of several parallel steps, data transfer is conducted only at the beginning and the end of the function. The other technique that shortens the transfer time is compressing the data to bitwise representation.

Another peculiarity in the implementation of the library functions is using the fast memories of the GPU. The CUDA platform provides access to the global memory of the graphic card and also to the much faster shared and local memories. In order to perform calculations on variables located on the shared or local memory, additional steps are necessary to move the data from variables located on global memory. However, due to the throughput-oriented organization of the CUDA API, this turns out to be much more efficient than directly using global memory variables for calculations.

The functions of the library are adapted to the size of the processed data. In the case of S-boxes of fewer variables, simultaneous parallel computations are made on all component functions defined by the considered S-box. For larger parameters, component functions are processed separately.

Many of the library's features also use the cutting-edge techniques of the ever-updating CUDA interface. A good example of this are the shuffle operations, which allow for the direct passing of the values of local variables from one thread to another. The use of bitwise calculations, where possible, provides another degree of parallelism and further optimization of the library functions.

In addition, BOOLSPLG can be used for training research, to view the characteristics of the video card, and to compare parallel and sequential performance. For example, we

used it to study bijective S-boxes with  $n \leq 18$  variables and good cryptographic properties which were derived from linear codes with quasi-cyclic structures in Ref. [13].

There are many libraries and mathematical software for computing the cryptographic properties of Boolean and vectorial Boolean functions developed for sequential CPU computation. As examples, we point out SAGEMATH [14], MATLAB [15], VBF LIBRARY [16], PEIGEN [17], or SET (S-box Evaluation Tool) [18]. A detailed review of the software related to the cryptographic characteristics of the S-boxes is made in [16]. This software is good for training purposes and basic calculations but is not as fast as parallel implementation, especially for large ( $n \geq 8$ ) S-boxes.

In terms of linear algebra GPU tools, we can mention cuBLAS [19] and cuSPARSE [20]. There are GPU libraries for butterfly algorithms, such as BPLG [21], NVIDIA’s cuFFT [22], but most of them are for signal processing (fast Fourier transform, Hartley transform, etc.) and not for vector Boolean functions. Examples of parallel software related to cryptography include Eval16BitSbox and the algorithms in Refs. [23,24]. Ref. [24] discusses only the linear approximation tables (LAT) of S-boxes and offers a different approach to their calculation. In addition, the authors evaluate and compare their work with other known software in this area. The implementation of Ref. [24] is in CUDA, which allows us to compare it with our software library BoolSPLG. We would like to point out that the computation time for LAT of  $16 \times 16$  S-boxes in Ref. [24] is close (comparable) to the computation time with a function of our library and is much less than the other presented software.

The paper is organized as follows. The main definitions connected to Boolean and vectorial Boolean functions are given in Section 2. In Section 3, we present some advantages of the CUDA programming platform. Basic facts and information about the data organization and the algorithms used is provided in Section 4. Section 5 presents some experimental results as well as a comparison of the calculation times of the considered library to SageMath and SET packages. A short conclusion and directions for future improvements to the presented library are given in Section 6.

## 2. Main Definitions and Preliminaries

In this section, we present the terminology and definitions we follow (see Refs. [8,25,26]).

Let  $\alpha_0, \alpha_1 \dots \alpha_{2^n-1}$  be the vectors of the  $n$ -dimensional vector space  $\mathbb{F}_2^n$  over the field  $\mathbb{F}_2 = \{0, 1\}$  in lexicographic order. There is a one-to-one correspondence between the vectors of  $\mathbb{F}_2^n$  and integers in  $[0; 2^n - 1]$ , which allows us to switch from a vector to an integer and vice versa. The Hamming weight  $w_H(v)$  of a vector  $v$  is the number of its nonzero coordinates.

A Boolean function  $f$  of  $n$  variables is a mapping from  $\mathbb{F}_2^n$  into  $\mathbb{F}_2$ . The Hamming distance  $d_H(f, g)$  between two Boolean functions  $f$  and  $g$  is the number of function values in which they differ. Two natural representations of a Boolean function are its truth table  $TT(f)$  and its algebraic normal form  $ANF(f)$ . Any Boolean function  $f$  of  $n$  variables is uniquely determined by its truth table  $TT(f) = (f(\alpha_0), f(\alpha_1) \dots f(\alpha_{2^n-1}))$ . Another way of uniquely representing a Boolean function  $f$  is by means of a polynomial with  $n$  variables, called its algebraic normal form (ANF), whose monomials have the form  $x_{i_1}x_{i_2} \dots x_{i_k}$ ,  $1 \leq i_1 < i_2 < \dots < i_k \leq n, 0 \leq k \leq n$  [8].

Denote by  $x^u$  the monomial  $x_1^{u_1}x_2^{u_2} \dots x_n^{u_n}$ , where  $u \in \mathbb{Z}, 0 \leq u \leq 2^n - 1$ ,  $u = (u_1, u_2, \dots, u_n) \in \mathbb{F}_2^n$ . Then the algebraic normal form of  $f$  is a polynomial

$$f(x) = f(x_1, x_2, \dots, x_n) = \bigoplus_{u=0}^{2^n-1} a_u x^u. \tag{1}$$

The degree of  $ANF(f)$  is called the algebraic degree  $deg(f)$  of the Boolean function  $f$ , and it is equal to the maximum number of variables of the terms  $x^u$ , or

$$deg(f) = \max\{w_H(u) | a_u = 1\}, \text{ where } f(x) = \bigoplus_{u=0}^{2^n-1} a_u x^u.$$

The Boolean functions  $a_0 \oplus a_1x_1 \oplus a_2x_2 \oplus \dots \oplus a_nx_n = a_0 \oplus l_a(x)$  of algebraic degree at most 1 play a special role in our investigations, and they are called *affine*, while  $l_a(x)$  are called *linear*.

Obviously,  $ANF(f)$  can be associated with the binary ( $2^n$ -dimensional) vector  $f^{ANF} \in \mathbb{F}_2^{2^n}$  whose coordinates are the coefficients in (1) following the lexicographical order [27].

Associated with the Boolean function  $f$  is the function  $\hat{f} = (-1)^f = 1 - 2f$  whose function values belong to the set  $\{-1; 1\}$ . The corresponding vector that contains the function's values of  $\hat{f}$  is called the polarity truth table (*PTT*) of the function  $f$ .

**Definition 1.** Walsh (Hadamard, Walsh–Hadamard, Walsh–Fourier) transform  $f^W$  of the Boolean function  $f$  is the integer-valued function  $f^W : \mathbb{F}_2^n \rightarrow \mathbb{Z}$ , defined by

$$f^W(a) = \sum_{x \in \mathbb{F}_2^n} (-1)^{f(x) \oplus l_a(x)} = \sum_{x \in \mathbb{F}_2^n} \hat{f}(x) \hat{l}_a(x) = 2^n - 2d_H(f, l_a),$$

where  $a = (a_1, \dots, a_n) \in \mathbb{F}_2^n$ .

The function  $\hat{f}(x)$  can be recovered by the inverse Walsh transform:

$$\hat{f}(x) = 2^{-n} \sum_{a \in \mathbb{F}_2^n} (f^W)(a) (-1)^{a \cdot x}.$$

The values of  $f^W$  are called Walsh coefficients of the Boolean function  $f$ . For any Boolean function  $f$  and any vector  $a \in \mathbb{F}_2^n$ , we have  $-2^n \leq f^W(a) \leq 2^n$ . The functions  $l_a(x)$  and  $\bar{l}_a(x) = l_a(x) \oplus 1$  have the maximal and minimal Walsh coefficients, namely  $l_a^W(a) = 2^n$  and  $\bar{l}_a^W(a) = -2^n$ .

The vector  $W_f = (f^W(\alpha_0), f^W(\alpha_1) \dots f^W(\alpha_{2^n-1}))$  is called the Walsh spectrum of the Boolean function and is denoted by  $W_f$ . The Walsh spectrum measures the distance to the linear and affine functions.

The linearity of a Boolean function  $f$  is the maximum absolute value of a Walsh coefficient of  $f$ :  $Lin(f) = \max\{|f^W(a)| \mid a \in \mathbb{F}_2^n\}$ . The Parseval's Equality  $\sum_{a \in \mathbb{F}_2^n} (f^W(a))^2 = 2^{2n}$  gives that  $Lin(f) \geq 2^{n/2}$  [8]. Functions attaining this lower bound are called bent functions.

Another important parameter which is closely connected with linearity is nonlinearity.

*Nonlinearity*  $nl(f)$  of the Boolean function  $f$  is the minimum Hamming distance from  $f$  to the nearest affine function:

$$nl(f) = \min\{d_H(f, g) \mid g - \text{affine function}\}.$$

The relation between the linearity and nonlinearity of the Boolean function  $f$  is given by the equality  $Lin(f) = 2^n - 2nl(f)$  [8]. Obviously, minimum linearity corresponds to maximum nonlinearity.

**Definition 2.** *Autocorrelation function* of the Boolean function  $f$  (auto-correlation of  $f$  with a shift  $w$ ) is the function  $r_f : \mathbb{F}_2^n \rightarrow \mathbb{Z}$  defined by:

$$r_f(w) = \sum_{x \in \mathbb{F}_2^n} (-1)^{f(x) \oplus f(x \oplus w)},$$

where  $w \in \mathbb{F}_2^n$ .

The expression of the autocorrelation values  $r_f(w)$  for all  $w \in \mathbb{F}_2^n$  in terms of the Walsh coefficients [6] is equal to

$$r_f(w) = 2^{-n} \sum_{u \in \mathbb{F}_2^n} (f^W(u))^2 (-1)^{u \cdot w}.$$

For any Boolean function  $f$  and any vector  $w \in \mathbb{F}_2^n$  we have  $-2^n \leq r_f(w) \leq 2^n$  and  $r_f(0) = 2^n$ . The vector of the autocorrelation values  $r_f(w)$  is referred to as its *autocorrelation spectrum* of the function  $f$ .

The *absolute indicator* of a Boolean function  $f$  of  $n$  variable, denoted by  $AC(f)$ , is the maximum absolute value of an autocorrelation value and is defined by  $AC(f) = \max\{|r_f(w)| \mid w \in \mathbb{F}_2^n\}$ .

The Sylvester–Hadamard matrix (or Walsh–Hadamard matrix) of order  $2^n$ , denoted by  $H_n$ , is generated by the recursive relation:

$$H_0 = 1, H_1 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, H_n = \begin{pmatrix} H_{n-1} & H_{n-1} \\ H_{n-1} & -H_{n-1} \end{pmatrix} = H_1 \otimes H_{n-1} \text{ for } n > 1,$$

where  $\otimes$  denotes the Kronecker product. The  $i$ -th row (column) of  $H_n$  is a PTT of the linear function  $l_i$ . So  $W_f^t = H_n \cdot PTT(f)^t$  and  $PTT(f)^t = 2^{-n} H_n W_f^t$ .

Fast Walsh transform (FWT) is usually used to calculate the Walsh spectrum. It is based on matrix vector multiplication and can be given by a butterfly diagram. The theoretical base of the FWT is given by Good [28] and it follows from a suitable factorization of  $H_n$ .

A similar approach can be used to calculate the fast Möbius transform (FMT). This transform gives the coefficients of ANF( $f$ ) from the truth table of the Boolean function  $f$  and vice versa. It is based on the following matrices:

$$A_0 = 1, A_1 = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} A_n = \begin{pmatrix} A_{n-1} & 0 \\ A_{n-1} & A_{n-1} \end{pmatrix} = A_1 \otimes A_{n-1} \text{ for } n > 1.$$

Actually, the  $i$ -th column of  $A_n$  is the truth table of the monomial  $m_i = x^{\alpha_i}$ . Using these matrices, we have  $(f^{ANF})^t = A_n \cdot (TT(f))^t$  and  $TT(f)^t = A_n \cdot (f^{ANF})^t$ . The complexity of the algorithms for both fast Walsh and Möbius transforms is  $O(n2^n)$ , and they require  $O(2^n)$  memory units.

A vectorial Boolean function  $S : \mathbb{F}_2^n \rightarrow \mathbb{F}_2^m$  (also called  $(n, m)$  S-box or shortly S-box) can be represented by the vector  $(f_1, f_2, \dots, f_m)$ , where  $f_i$  are Boolean functions of  $n$  variables,  $i = 1, 2, \dots, m$ . The functions  $f_i$  are called the coordinate functions of the S-box. Then the  $m \times 2^n$  matrix

$$G_S = \begin{pmatrix} TT(f_1) \\ \vdots \\ TT(f_m) \end{pmatrix}$$

represents the considered S-box, where  $TT(f_i)$  is the truth table of the Boolean function  $f_i$ ,  $i = 1, \dots, m$ . An S-box is bijective if  $n = m$  and  $S$  is an invertible function.

In order to study the cryptographic properties of a vectorial Boolean function  $f$  related to linearity, algebraic degree, and autocorrelation, we need to consider all non-zero linear combinations of the coordinate functions of the S-box, denoted by

$$S_b = b \cdot G_S = b_1 f_1 \oplus \dots \oplus b_m f_m,$$

where  $b = (b_1, \dots, b_m) \in \mathbb{F}_2^m$ . These are the component functions of the S-box  $S$ .

The Walsh spectrum of  $S$  is defined as the collection of all of the Walsh spectra of the component functions of  $S$ . The linearity and nonlinearity of the vectorial Boolean function are defined as

$$Lin(S) = \max_{b \in \mathbb{F}_2^m \setminus \{0\}} Lin(S_b), \quad nl(S) = \min_{b \in \mathbb{F}_2^m \setminus \{0\}} nl(S_b).$$

In order to obtain the important parameters of an S-box, we use four tables, namely a linear approximation table (LAT), a difference distribution table (DDT), an autocorrelation table (ACT), and a table with the algebraic degrees (ADT) of the monomials in the component Boolean functions of the considered S-box. We define these tables below.

The  $2^n \times 2^m$  table whose entries are defined by

$$L_{a,b} = |\{x \in \mathbb{F}_2^n : S_b(x) = a \cdot x - 2^{n-1}\}|, \quad a \in \mathbb{F}_2^n, b \in \mathbb{F}_2^m,$$

is called its *linear approximation table* and is also denoted by  $LAT(S)$ . The elements of  $LAT(S)$  show the relationship between the inputs and outputs of the S-box. Since  $S_b^W(a) = 2L_{a,b}$ , the Walsh spectrum and the linear approximation table of an S-box are closely related, and by computing one of these parameters, we obtain the other. Therefore, we actually compute the Walsh spectrum instead of the  $LAT(S)$  in order to find the linearity and nonlinearity of  $S$ .

Another important parameter related to an S-box  $S$  is its algebraic degree. We define this as the maximum among all degrees of the component functions, or  $deg(S) = \max_{b \in \mathbb{F}_2^m \setminus \{0\}} deg(S_b)$ . The minimum degree is also important regarding algebraic attacks. Therefore, we define the maximal and the minimal algebraic degree of the vectorial Boolean function  $S$  as

$$\max deg(S) = \max_{b \in \mathbb{F}_2^m \setminus \{0\}} deg(S_b), \quad \min deg(S) = \min_{b \in \mathbb{F}_2^m \setminus \{0\}} deg(S_b).$$

Autocorrelation spectrum  $ACT$  of the vectorial Boolean function  $S$  is defined as the collection of all autocorrelation spectra of its component functions. In fact, we consider  $ACT(S)$  as a  $2^n \times 2^m$  autocorrelation matrix, whose columns represent the autocorrelation functions of all component Boolean functions of  $S$ . The autocorrelation (or the maximal absolute autocorrelation value)  $AC(S)$  is defined as:

$$AC(S) = \max_{b \in \mathbb{F}_2^m \setminus \{0\}} |r(S_b)|.$$

The difference distribution table (DDT) is a  $2^n \times 2^m$  table whose entries are defined as

$$DDT(S)_{\alpha,\beta} = |\{x \in \mathbb{F}_2^n, \alpha \in \mathbb{F}_2^n \setminus \{0\}, \beta \in \mathbb{F}_2^m | S(x) \oplus S(x \oplus \alpha) = \beta\}|. \quad (2)$$

The differential uniformity denoted by  $\delta(S)$  is defined as the largest value in its difference distribution table not counting the first entry in the first row, or

$$\delta(S) = \max_{\alpha \neq 0, \beta} DDT(S)_{\alpha,\beta}.$$

We are looking for S-boxes that have a differential uniformity as low as possible. It is well known that  $\delta(S)$  takes only even values in the interval  $[2^{n-m}, 2^n]$ . The smallest possible value of  $\delta$  in the case of bijective S-boxes ( $n = m$ ) is 2.

### 3. GPU and CUDA

One way to understand the difference between CPU and GPU is to compare the ways they process tasks. Usually, CPU consists of a few cores optimized for sequential serial processing. They have powerful ALU, large caches, and sophisticated control. Modern NVIDIA GPUs have their own memory, a massively parallel architecture consisting of thousands of smaller cores and designed for handling multiple tasks simultaneously. These cores have a throughput-oriented design with small caches, simple control, and energy-efficient ALUs, and they require a massive number of threads to tolerate latency. A GPU is very convenient when manipulating large data or using a high number of threads in single-instruction multiple-data (SIMD) programming model [29].

The CUDA programming platform allows programmers to interact directly with GPUs and run parallel parts of programs using the advantages of GPU architecture [9]. CUDA C is a programming language close to C by syntax, but conceptually and semantically it is quite different from C. The source code for CUDA applications consists of a mixture of conventional C/C++ host code and GPU device functions.



The processing of the data flow has several steps. At the highest level, we have a master process which runs on the CPU and performs the following steps: prepares data for manipulation, allocates memory on GPU, copies data from the host (CPU) to the GPU global memory, launches multiple instances of the execution “kernel” on GPU, copies data from the GPU memory to the host, deallocates all memory, and terminates. In a program, a parallel GPU part can be activated many times with different data and manipulations.

Functions for parallel execution on GPU are written in units called kernels. Syntactically, a kernel is a function of a programming language that is very similar to the C/C++ language functions. However, semantically, it is used by several directions. Its header initiates a grid of threads that practically performs the parallel execution of the calculations. The definition of a kernel header, which contains the type of grid, is given as follows:

$$\textit{kernel\_name} \lll \textit{grid blocks, threads per block} \ggg (\dots),$$

where “*kernel\_name*” is a usual name (identifier) and ‘grid blocks’ and ‘threads per block’ are positive integers. After the header, each kernel consists of a program code that refers to the single thread of the grid. Any thread has its own number in the grid of threads. According to this number, it is determined which part of the data will be calculated by the particular thread. Kernels are executed over the stream of data by many threads on a device in parallel. Thread is a process that performs series of programming instructions and it is a single instance of the kernel. Threads are organized into blocks, which are sets of threads that can communicate and synchronize their execution. Maximum 1024 threads per block can be launched. Multiple blocks can be executed simultaneously. First, a configuration of the kernel (number of blocks and number of threads per block inside) has to be made in order to launch it. Blocks and threads per block form a grid. All threads run the same code.

The threads are executed in groups of 32 threads called “wraps”. Usually, all of the threads in the wrap execute the same instruction at the same time. The difference is only input data, which depends on the unique number of any thread in block and any block in a grid.

The memory model has the following features. Each thread has access to the slowest global memory, but threads from different blocks can communicate with it. Each block has its own memory called shared which serves the communication between all of the threads in one block and is much faster than the global one. Each thread uses a small amount of fast local memory. The variables in the global memory, unlike the others, are preserved even after the terminating execution of each kernel.

We would like to mention some features of the CUDA model that are especially important for the efficiency of GPU calculations. Creating and destroying threads takes a negligible amount of time but only states which resources will be needed, so they do not affect performance. The time required to transfer data from the main memory to the global GPU memory and vice versa in many cases turns out to significantly lengthen the duration of calculations. Therefore, the master process has to manage the overall performance by running different kernels sequentially (if possible) without intermediate data transfer and only returning the final result.

#### 4. Strategies in Algorithms and Data Organization

In order to discuss the implemented strategies and how the data is organized in the memory, it is necessary to show the model of the library BoolSPLG. Its structure is presented in Figure 1.

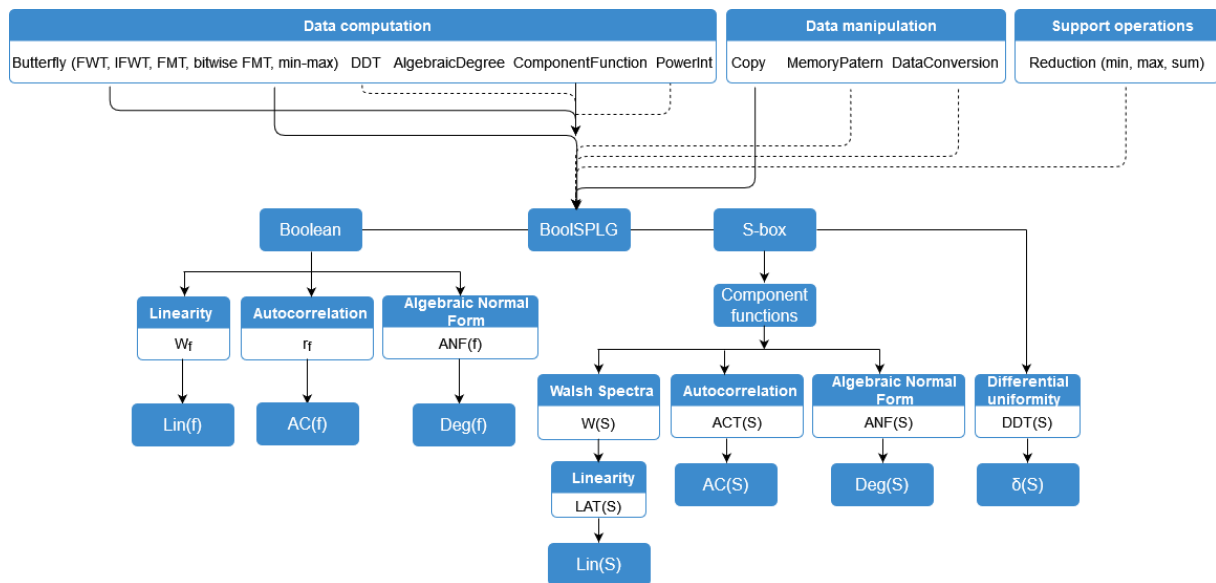


Figure 1. Classification and module dependencies of the building blocks involved in the library.

#### 4.1. Data Organization

One way to represent the  $n \times n$  vectorial Boolean function  $S$  that we use is through the truth tables of its coordinate functions. Therefore, an  $n \times 2^n$  matrix is needed. For convenience, we use the integers corresponding to the binary representation of the columns of this matrix, so  $S$  is defined by the integer vector  $(s_0, s_1, \dots, s_{2^n-1})$ . This representation has several advantages: data from the main memory is transferred to the GPU memory much faster, and the value of the function  $S$  for the input vector  $v \in \mathbb{F}_2^n$  corresponds to the  $v$ -th coordinate of the array.

Any vector  $a \in \mathbb{F}_2^n$  defines the component function  $S_a$  as a linear combination of rows of the matrix corresponding to  $S$ . The truth table of the component function  $S_a$  can be calculated in the following way:

$$TT(S_a) = (a \cdot s_0, a \cdot s_1, \dots, a \cdot s_{2^n-1}).$$

#### 4.2. Strategies in Algorithms

Effective sequential algorithms serve as a basis for parallel implementation. An in-depth examination of such algorithms, in some cases with different approaches, has been conducted in Ref. [30]. For the relationship between the different linear and differential characteristics, see Ref. [31]. The time complexity of the fast Walsh and Möbius transforms of a Boolean function is  $O(n2^n)$ , and the required memory is  $O(2^n)$ .

The fast Walsh Transform (FWT) is the main tool (part) of the functions in the library related to the linear characteristics of the the vectorial Boolean functions. In order to apply the fast Walsh transform to a Boolean function with  $n$  variables, we consider its truth tables as an integer array of length  $2^n$ . The algorithm requires  $n$  steps. In the  $i$ -th step, the sum and the difference of the integers from the  $j$ th and  $r$ th cells (depending on  $i$ ) of the current tuple must be written in the  $j$ th and  $r$ th cells of the new tuple. Therefore, the same array and variables can be used for the result of the calculations of all of its steps. Each thread can calculate the values of two elements without communicating with other threads. When using shared memory, it is convenient to calculate several steps of FWT on one thread on a given part of the array. A detailed description is presented in Ref. [32].

When calculating the linearity, nonlinearity, or LAT of a vectorial Boolean function  $S$ , we need all of its components' functions (not only the coordinate functions). For small  $n$ , we list all of the component functions in an array of size  $2^{2^n}$ . After the first  $n$  steps of the fast Walsh transform are applied to this array, we obtain the Walsh spectrum of  $S$ . This is enough to find the linearity (nonlinearity) of the vectorial Boolean function. The result is



obtained by finding the minimum element in the set of absolute values of the coordinates of the obtained vector (by reduction). For larger values of  $n$  (when the hardware resource of the GPU is insufficient), in order to determine the linearity of the vector Boolean function  $S$ , the Walsh transform of each component Boolean function has to be calculated separately (see Algorithm 1).

---

**Algorithm 1:** Linearity of an S-box.

---

**Input:** An S-box  $S\_box$   
**Output:** The linearity  $Lin(S\_box)$   
*sequential* copy  $S\_box$  (as it is represented) to GPU memory  
*sequential*  $Lin(S\_box) = 0$   
*sequential* for all component Boolean functions of  $S\_box$  do  
   {  
*parallel* get TT of current component function  $f$   
*parallel* calculate  $W^f$  (the Walsh spectrum of  $f$ )  
*parallel* compute the linearity of  $f$   
*parallel* update the value of  $Lin(S\_box)$  // using one thread  
   }  
*sequential* copy linearity of S-box in RAM memory

---

For the autocorrelation properties of S-boxes with  $n > 10$ , we use the following Algorithm 2.

---

**Algorithm 2:** The autocorrelation of an S-box.

---

**Input:** An S-box  $S\_box$   
**Output:**  $AC(S\_box)$   
*sequential* copy  $S\_box$  to GPU memory  
*sequential* current autocorrelation value  $AC(S\_box) = 0$   
*sequential* for all component Boolean functions of  $S\_box$  do  
   {  
*parallel* get TT of current component function  $f$   
*parallel* calculate  $W^f$  and  $g = (W^f)^2$   
*parallel* compute  $2^{-n}W^g$   
*parallel* update the value of  $AC(S\_box)$   
   }  
*sequential* copy  $AC(S\_box)$  in RAM memory

---

We would like to note the following points in the calculation of the algebraic degree of a Boolean function. Since  $ANF(f)$  can also be computed via a fast Möbius transform from the truth table of a Boolean function, this can be achieved using only bitwise operations. Compared to Walsh transform, this is a very significant advantage and allows us to use bitwise representation of the truth tables. Note that the  $i$ -th coordinate of the vector of  $ANF(f)$  corresponds to the monomial  $x^i$ , and the degree of this monomial is equal to the Hamming weight of  $i$ . This can easily be calculated from a single thread. A detailed description of the corresponding parallel butterfly algorithms is given in Ref. [33].

From the definition of DDT for a vectorial Boolean function, we can directly derive a basic algorithm. We enumerate all of the values of the input difference  $\Delta$ . For each possible difference, we initialize to zero an array  $DDT_{\Delta}(S)$  of  $2^n$  cells, one for each possible output difference. Then, for each pair  $(x, x \oplus \Delta)$  with the prescribed input difference, its output difference  $S(x) \oplus S(x \oplus \Delta)$  is computed and the corresponding counter in  $DDT_{\Delta}(S)$  is incremented. The runtime for sequential implementation is  $O(2^{2n})$ .

In our parallel implementation, each thread calculates only one output difference  $S(x) \oplus S(x \oplus \Delta)$ . The second part, increasing the value of the corresponding DDT cell, is more difficult. More than one thread can yield the same value for the output difference. Therefore, they have to write the results in the same time, and to avoid "race conditions", we use so-called *atomic* operators.

For S-boxes of less than 15 variables, all of the tables that we used are generated and accessible through the functions of the library.

In this case, the DDT for all of the input differences is generated in parallel. For S-boxes of more than 14 variables, the rows for each of the tables are generated sequentially (see Algorithm 3), but the elements of one row are generated in parallel.

**Algorithm 3:** The differential uniformity of an S-box

---

<b>Input:</b>	An S-box $S\_box$
<b>Output:</b>	$\delta(S\_box)$
<i>sequential</i>	copy $S\_box$ to GPU memory
<i>sequential</i>	$\delta(S\_box) = 2^n$
<i>sequential</i>	for all $\Delta \in \mathbb{F}_2^n$ do
	{
<i>parallel</i>	compute $DDT_{\Delta}(S\_box)$
<i>parallel</i>	find current $\delta(S\_box)$
	}
<i>sequential</i>	copy $\delta(S\_box)$ in RAM memory

---

### 5. Experimental Results

A server with two different GPU devices was used to evaluate the efficiency of the implemented library. Their parameters are listed in Table 1. One of the GPUs, presented as Device 0, is much more powerful than the other (Device 1).

The average times for calculating the considered cryptographic parameters per 100 randomly generated invertible S-boxes with  $n$  variables for any  $n = 8, 9, \dots, 20$  were obtained. The results for the different parameters were systematized in Tables 2–5. The first column in each table shows the size of the considered S-box. The second column contains the average time required to find the parameter with a sequential program using a single CPU core. The next column shows the time required to find the corresponding parameter using Device 0. Then, the speedup found between the sequential and parallel implementation methods is given by the formula

$$S_p = \frac{T_0(n)}{T_p(n)},$$

where  $n$  is the number of variables in the S-box,  $T_0(n)$  is the execution time of the fastest known sequential algorithm, and  $T_p(n)$  is the execution time of the parallel algorithm. The speedup of the parallel algorithm is given in the columns CPU vs. Dev0 and CPU vs. Dev1, respectively. For example, the linearity of an S-box of size  $2^8$  is calculated for 0.863 ms by the CPU, while Device 0 performs the calculation for about 0.223 ms. This means it gives a 3.86-times better executing time, written in the table as  $\times 3.86$ .

It should be noted that the time required for parallel implementation includes the time used for data transfer from RAM to device memory and vice versa. The next two columns provide the execution time of Device 2 and the corresponding acceleration. The test results show the following: Using parallel implementation is much more efficient for S-boxes with larger parameters. In parallel implementation, the algebraic degree is calculated the fastest and the DDT is calculated the slowest. Device 0 gives much better acceleration in most cases.

As can be seen, for S-boxes with particular sizes, the speedup drops. This occurs due to the CUDA memory hierarchy model. The functions in the library generate the following tables related to S-boxes: a linear approximation table LAT(S), an autocorrelation table ACT(S), a table with the algebraic degrees of the monomials in the component Boolean functions ADT(S), and a difference distribution table (DDT). In the case of S-boxes with less than 11 variables, all component functions are calculated simultaneously (LAT, ACT, and ADT). The necessary transformations of the functions to one vector saved in the global memory are also performed simultaneously. For S-boxes with 11 or more variables, the

component functions are generated one after the other. Further, in the case of S-boxes with more than 14 variables, the rows of the DDT table are generated sequentially, and this takes more time as well.

**Table 1.** Description of the used CPU and GPU devices.

Environment:		Platform	
CPU	Intel Xeon E5-2640, 2.50 GHz		
Memory	48 GB DDR3 1333 MHz		
OS	Windows 7, 64-bit		
IDE/Compiler	MSVC 2019		
CUDA SDK	10.2		
GPU Driver	V 471.96		
GPU	Nvidia TITAN X (Pascal)	GeForce GTX TITAN	
	Device 0	Device 1	
Architecture	Pascal	Kepler	
CUDA Cores	3584	2688	
Boost Clock	1531 MHz	876 MHz	
Memory Speed	10 Gbps	6 Gbps	
Global Memory	12 GB GDDR5X	6 GB GDDR5	
Memory Bandwidth	480 (GB/sec)	288.38 (GB/sec)	

One million randomly generated invertible 16-bit S-boxes have been studied. Table 6 provides information on the best S-boxes in terms of the considered cryptographic parameters. For comparison, the table also presents the parameters of one S-box generated in a different way from a quasi-cyclic code.

**Table 2.** Computing linearity

Size	CPU (ms)	Device 0 (ms)	CPU vs. Dev 0	Device 1 (ms)	CPU vs. Dev 1
2 <sup>8</sup> (256)	0.863	0.223	×3.86	0.14336	×6
2 <sup>9</sup> (512)	7.654	0.340	×22.51	0.3576	×21.4
2 <sup>10</sup> (1024)	15.855	0.318	×49.85	1.1442	×13.8
2 <sup>11</sup> (2048)	68.343	62.774	×1	72	×1
2 <sup>12</sup> (4096)	314.927	149.562	×2.1	186.106	×1.68
2 <sup>13</sup> (8192)	1257.97	287.688	×4.83	347.72	×3.6
2 <sup>14</sup> (16,384)	5686.84	622.29	×9.13	759.53	×7.5
2 <sup>15</sup> (32,768)	24,083.8	1239.54	×19.42	2141.55	×11.2
2 <sup>16</sup> (65,536)	99,800.3	2446.03	×40.8	6289.8	×15.86
2 <sup>17</sup> (131,072)	442,678	6165.59	×71.8	22,772	×19.43
2 <sup>18</sup> (262,144)	1,677,921	22,029.27	×76.1	90,089	×18.62
2 <sup>19</sup> (524,288)	7,562,247	90,786.66	×83.29	375,519	×20.13
2 <sup>20</sup> (1,048,576)	29,638,868	418,942	×70.7	1,618,402	×18.3

**Table 3.** Computing autocorrelation

Size	CPU (ms)	Device 0 (ms)	CPU vs. Dev 0	Device 1 (ms)	CPU vs. Dev 1
2 <sup>8</sup> (256)	2.020	0.206	×9.8	0.319	×6.3
2 <sup>9</sup> (512)	14.212	0.435	×32.67	0.393	×36.13
2 <sup>10</sup> (1024)	37.940	0.463	×81.94	1.462	×25.95
2 <sup>11</sup> (2048)	185.673	132.284	×1.4	144.06	×1.28
2 <sup>12</sup> (4096)	771.527	209.130	×3.68	253.18	×3
2 <sup>13</sup> (8192)	3142.98	503.77	×6.23	418.54	×7.5
2 <sup>14</sup> (16,384)	13,555.2	866.02	×15.65	1094.63	×12.3
2 <sup>15</sup> (32,768)	57,324.5	1931.21	×29.68	3189.85	×18
2 <sup>16</sup> (65,536)	238,216.4	3813.45	×62.46	9621.8	×24.76
2 <sup>17</sup> (131,072)	1,060,294	9396.38	×112.84	34,810.46	×30.45
2 <sup>18</sup> (262,144)	3,832,308	33,814.94	×113.33	133,629.15	×28.67
2 <sup>19</sup> (524,288)	16,860,299	138,108.15	×122.1	566,914	×29.7
2 <sup>20</sup> (1,048,576)	68,227,870	629,515	×108.38	2,411,828	×28.2

**Table 4.** Computing differential uniformity

Size	CPU (ms)	Device 0 (ms)	CPU vs. Dev 0	Device 1 (ms)	CPU vs. Dev 1
2 <sup>8</sup> (256)	0.282	0.208	×1	0.136	×2
2 <sup>9</sup> (512)	1.908	0.432	×4.416	0.351	×5.4
2 <sup>10</sup> (1024)	3.591	0.366	×9.811	1.053	×3.4
2 <sup>11</sup> (2048)	14.269	0.705	×20.23	1.313	×10.8
2 <sup>12</sup> (4096)	69.932	1.710	×40.89	2.914	×24
2 <sup>13</sup> (8192)	245.719	5.773	×42.56	11.27	×22
2 <sup>14</sup> (16,384)	998.174	21.022	×47.48	37.64	×27
2 <sup>15</sup> (32,768)	4059.71	990.88	×4.1	1497.6	×2.7
2 <sup>16</sup> (65,536)	18,307.9	1924.91	×9.5	4345	×4.2
2 <sup>17</sup> (131,072)	86,983.9	4206.78	×20.68	14,988.47	×5.8
2 <sup>18</sup> (262,144)	453,136	14,319.73	×31.64	55,269.61	×8.2
2 <sup>19</sup> (524,288)	1,925,444	61,572	×31.27	283,692	×6.7
2 <sup>20</sup> (1,048,576)	9,312,785	730,660	×12.7	1,531,534	×6

**Table 5.** Computing degree— $deg(S)$ — $deg\_bitwise(S)$

Size	CPU (ms)	Device 0 Base (ms)	Device 0 Bitwise	CPU vs. Dev0 Base	CPU vs. Dev0 Bitwise	Device 1 Base (ms)	Device 1 Bitwise	CPU vs. Dev1 Base	CPU vs. Dev1 Bitwise
2 <sup>8</sup>	1.127	0.122	0.115	×9.23	×9.8	0.3193	0.2692	×3.5	×14.1
2 <sup>9</sup>	10.099	0.201	0.202	×49.9	×49	0.7321	0.3087	×13.6	×32.46
2 <sup>10</sup>	20.396	0.318	0.142	×64.1	×143.6	1.1814	0.7277	×17.2	×28
2 <sup>11</sup>	102.390	76.042	12.447	×1.3	×8.2	78.66	23.71	×1.3	×4.3
2 <sup>12</sup>	369.589	162.987	28.963	×2.2	×12.7	220.58	42.9	×1.6	×8.6
2 <sup>13</sup>	1754.26	307.870	54.385	×5.69	×32.2	366	65.68	×4.8	×26.9
2 <sup>14</sup>	6822.06	720.525	279.5	×9.4	×24.45	822.66	767.35	×8.3	×8.9
2 <sup>15</sup>	28,395.3	1410.75	489.42	×20.1	×58	2221	2658.13	×12.78	×10.6
2 <sup>16</sup>	117,070.8	2961.87	1593.99	×39.5	×73.5	6400	9964	×18.29	×11.74
2 <sup>17</sup>	492,150.2	6421.09	6470	×76.6	×76	22,974	26,110	×21.42	×18.84
2 <sup>18</sup>	1,699,950	22,824.7	12,011.3	×74.48	×141.53	84,706	56,085	×20	×30.1
2 <sup>19</sup>	7,327,083	90,603.8	32,384.5	×80.87	×226.2	356,180	122,500	×20.5	×59.8
2 <sup>20</sup>	29,481,099	416,376	83,943	×70.8	×351.2	1,530,321	478,612	×19.2	×61.59

**Table 6.** Test evaluation between bijective 16-bit QCS-boxes and randomly generated 16-bit S-boxes

S-Boxes	Lin	nl	$\delta$	deg(S) (max)	AC(S)	Number
QCS-boxes, $n = 16$ C2, $M_1, m = 13,107, r = 5$	512	32,512	4	15	512	5
S-boxes, random $n = 16$	1532	32,002	22	15	2344, 2248	2
	1532	32,002	20	15	2568–2208	34
	1532	32,002	18	15	2432–2224	39
S-boxes, random $n = 16$	1528	32,004	20	15	2504–2232	23
	1528	32,004	18	15	2416–2264	17
S-boxes, random $n = 16$	1524	32,006	20	15	2512–2240	8
	1524	32,006	18	15	2392–2216	7
S-boxes, random $n = 16$	1520	32,008	20	15	2288, 2280, 2184	3
	1520	32,008	18	15	2352–2264	5
S-boxes, random $n = 16$	1516	32,010	20	15	2280	1
	1516	32,010	18	15	2312	1

In addition, we compared the parallel version of the presented library with the packages SageMath v9.8 and SET. In Table 7 are given the calculation times of the following cryptographic parameters of the S-boxes: linearity, differential uniformity, algebraic degree, and autocorrelation. The computing environment for BoolSPLG and SET is presented in Table 1, while for SageMath we used SageMathCell. Computing the linearity and the differential uniformity of S-boxes of sizes bigger than  $2^{12}$  is not possible in SageMath. The calculation of autocorrelation is also not included in this package.

**Table 7.** Calculation times of BoolSPLG, Sage, and SET

Size	Lin			$\delta$			deg			AC	
	Sage	SET	BoolSPLG	Sage	SET	BoolSPLG	Sage	SET	BoolSPLG	SET	BoolSPLG
$2^8$	0.66 s	2 ms	0.22 ms	0.096 s	0.4 ms	0.2 ms	0.12 s	3 ms	0.1 ms	2 ms	0.2 ms
$2^9$	1.94 s	5 ms	0.34 ms	0.35 s	2 ms	0.4 ms	0.14 s	18 ms	0.2 ms	7 ms	0.4 ms
$2^{10}$	7.2 s	39 ms	0.32 ms	1.44 s	8 ms	0.3 ms	0.18 s	52 ms	0.1 ms	48 ms	0.4 ms
$2^{11}$	26.3 s	246 ms	62 ms	6.2 s	83 ms	0.7 ms	0.22 s	374 ms	12 ms	0.4 s	132 ms
$2^{12}$	99 s	1.5 s	149 ms	29 s	0.5 s	1.7 ms	0.39 s	2.3 s	28 ms	2.7 s	209 ms
$2^{13}$	N/A	7.9 s	284 ms	N/A	2.1 s	5.7 ms	1 s	12.4 s	54 ms	14 s	503 ms
$2^{14}$	N/A	66 s	662 ms	N/A	9.9 s	21 ms	1.6 s	91 s	279 ms	102 s	866 ms

### 6. Conclusions and Future Work

In this article, a C++ library with sequential and parallel functions, implemented in CUDA C and designed to analyze large vectorial Boolean functions from cryptographic perspective, was presented. The parallel functions for many of the parameters are up to 60 times faster, which makes them convenient to use in ambitious research projects. The library has several opportunities for development. One is in the direction of universality. We plan to expand it so that it can be used for any type of vectorial Boolean function, not just bijective.

Another direction is to present more detailed information about the studied cryptographic parameters. For example, S-boxes that have the same differential uniformity but different differential spectra can perform differently in terms of resistance against differential attacks. Thus, some design criteria impose restrictions on the differential spectra of the S-box. The other example is related to algebraic degree. Note that, different from the notion of algebraic degree, the minimum among all degrees of the coordinate functions does not equal the minimum among all degrees of the component functions. Moreover, the number of component and coordinate functions with minimum (or maximal) degree is also important for some of the cases. In the current version, the library only calculates the value of the smallest (or largest) degrees of the component functions.

In the parallel functions of the current implementation, the calculations are conducted in two ways. One way (for smaller parameters) is to perform calculations for all of the component functions simultaneously. The other is to perform calculations for each of the component functions separately. In cases where the component functions are not of sufficient size, the second method of calculation does not provide good acceleration (as can be seen from the experimental results). In this case, it is more efficient to make calculations on appropriate groups of component functions.

**Author Contributions:** Conceptualization, D.B. and I.B.; methodology, D.B. and I.B.; software, D.B.; validation, D.B.; formal analysis, D.B. and I.B.; investigation, D.B., I.B. and M.D.-S.; resources, D.B., I.B. and M.D.-S.; data curation, D.B.; writing—original draft preparation, D.B., I.B. and M.D.-S.; writing—review and editing, M.D.-S.; visualization, M.D.-S.; supervision, D.B., I.B. and M.D.-S.; project administration, I.B.; funding acquisition, M.D.-S. All authors have read and agreed to the published version of the manuscript.

**Funding:** The research of Dushan Bikov and Iliya Bouyukliev is partially supported by the Bulgarian National Science Fund under contract no. KP-06-N62/2/13.12.2022. The research of Mariya Dzhumalieva-Stoeva was supported, in part, by a Bulgarian NSF contract KP-06-N32/2-2019.

**Data Availability Statement:** The library BoolSPLG, as well as the user manual and documentation, is available at <https://github.com/BoolSPLG/BoolSPLG-v0.3> and <https://doi.org/10.5281/zenodo.7825493>, accessed on 15 March 2023. There are also detailed descriptions of each function of the library and test examples. A Cmake file is provided for easy compilation.

**Acknowledgments:** We gratefully acknowledge the support of the NVIDIA Corporation with the donation of the Titan X Pascal GPU used for this research.

**Conflicts of Interest:** The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

## Abbreviations

The following abbreviations are used in this manuscript:

BoolSPLG	Boolean functions and S-box Parallel Library for GPU
SIMD	Single Instruction, Multiple Data
CUDA	Compute Unified Device Architecture
GPU	Graphic Processing Unit
CPU	Central Processing Unit
ALU	Arithmetic-Logic Unit
LAT	Linear Approximation Table
TT	Truth Table
PTT	Polarity Truth Table
ANF	Algebraic Normal Form
FWT	Fast Walsh Transform
FMT	Fast Möbius Transform
DDT	Difference Distribution Table
ACT	Autocorrelation Table
ADT	Algebraic Degree Table

## References

- Shetty, V.S.; Anusha, R.; Dileep Kumar, M.J.; Hegde, P. A survey on performance analysis of block cipher algorithms. In Proceedings of the 2020 International Conference on Inventive Computation Technologies (ICICT), Coimbatore, India, 26–28 February 2020; pp. 167–174.
- Kelly, M.; Kaminsky, A.; Kurdziel, M.; Łukowiak, M.; Radziszowski, S. Customizable sponge-based authenticated encryption using 16-bit s-boxes. In Proceedings of the MILCOM 2015-2015 IEEE Military Communications Conference, Tampa, FL, USA, 26–28 October 2015; pp. 43–48.



3. Canteaut, A.; Duval, S.; Leurent, G.; Naya-Plasencia, M.; Perrin, L.; Pornin, T.; Schrottenloher, A. Saturnin: A suite of lightweight symmetric algorithms for post-quantum security. *IACR Trans. Symmetric Cryptol.* **2020**, *2020*, 160–207. [[CrossRef](#)]
4. Matsui, M. New block encryption algorithm MISTY. In *Proceedings of the International Workshop on Fast Software Encryption*; Springer: Berlin/Heidelberg, Germany, 1997; pp. 54–68.
5. Georgi, I.; Nikolay, N.; Svetla, N. Reversed Genetic Algorithms for Generation of Bijective S-boxes with Good Cryptographic Properties. *IACR Cryptol. ePrint Arch.* **2014**, *2014*, 801.
6. Beauchamp, K. *Applications of Walsh and Related Functions. With an Introduction to Sequence Theory*; Microelectronics and Signal Processing Series; Academic Press, Inc.: London, UK; Orlando, FL, USA, 1985; p. xvi+308, ISBN 0-12-084180-0.
7. Bakoev, V. A method for fast computing the algebraic degree of boolean functions. In *Proceedings of the 21st International Conference on Computer Systems and Technologies*, Ruse, Bulgaria, 19–20 June 2020; pp. 141–147.
8. Carlet, C.; Crama, Y.; Hammer, P.L. Chapter Eight—Boolean Functions for Cryptography and Error-Correcting Codes. In *Boolean Models and Methods Mathematics, Computer Science, and Engineering*; Cambridge University Press: Cambridge, UK, 2010.
9. Sanders, J.; Kandrot, E. *CUDA by Example: An Introduction to General-Purpose GPU Programming*; Addison-Wesley Professional: Boston, MA, USA, 2010.
10. Jeon, W.; Ko, G.; Lee, J.; Lee, H.; Ha, D.; Ro, W.W. Chapter Six—Deep learning with GPUs. In *Hardware Accelerator Systems for Artificial Intelligence and Machine Learning*; Advances in Computers; Kim, S., DeKa, G.C., Eds.; Elsevier: Amsterdam, The Netherlands, 2021; Volume 122, pp. 167–215. .: 10.1016/bs.adcom.2020.11.003. [[CrossRef](#)]
11. Xie, Z.; Kwak, A.S.; George, E.; Dozal, L.W.; Van, H.; Jah, M.; Furfaro, R.; Jansen, P. Extracting Space Situational Awareness Events from News Text. *arXiv* **2022**, arXiv:2201.05721.
12. Stone, J.E.; Phillips, J.C.; Freddolino, P.L.; Hardy, D.J.; Trabuco, L.G.; Schulten, K. Accelerating molecular modeling applications with graphics processors. *J. Comput. Chem.* **2007**, *28*, 2618–2640. [[CrossRef](#)] [[PubMed](#)]
13. Bikov, D.; Bouyukliev, I.; Bouyuklieva, S. Bijective S-boxes of different sizes obtained from quasi-cyclic codes. *J. Algebra Comb. Discret. Struct. Appl.* **2019**, *6*, 123–134. [[CrossRef](#)]
14. Zimmermann, P.; Casamayou, A.; Cohen, N.; Connan, G.; Dumont, T.; Fousse, L.; Maltey, F.; Meulien, M.; Mezzarobba, M.; Pernet, C.; et al. *Computational mathematics with SageMath*; SIAM: Philadelphia, PA, USA, 2018.
15. Higham, D.J.; Higham, N.J. *MATLAB Guide*; SIAM: Philadelphia, PA, USA, 2016.
16. Álvarez-Cubero, J.A.; Zufiria, P.J. Algorithm 959: VBF: A library of C++ classes for vector Boolean functions in cryptography. *ACM Trans. Math. Softw. (TOMS)* **2016**, *42*, 1–22. [[CrossRef](#)]
17. Sasaki, Y.; Ling, S.; Guo, J.; Bao, Z.; Bao, Z.; Guo, J.; Ling, S.; Sasaki, Y.; Commons License, C. PEIGEN—A platform for evaluation, implementation, and generation of S-boxes. *IACR Trans. Symmetric Cryptol.* **2019**, *2019*, 330–394.
18. Picek, S.; Batina, L.; Jakobović, D.; Ege, B.; Golub, M. S-box, SET, match: A toolbox for S-box analysis. In *Proceedings of the IFIP International Workshop on Information Security Theory and Practice*; Springer: Berlin/Heidelberg, Germany, 2014; pp. 140–149.
19. Barrachina, S.; Castillo, M.; Igual, F.D.; Mayo, R.; Quintana-Orti, E.S. Evaluation and tuning of the level 3 CUBLAS for graphics processors. In *Proceedings of the 2008 IEEE International Symposium on Parallel and Distributed Processing*, Miami, FL, USA, 14–18 April 2008; pp. 1–8.
20. Naumov, M.; Chien, L.; Vandermersch, P.; Kapasi, U. Cuspars library. In *Proceedings of the GPU Technology Conference*, San Jose, CA, USA, 23 September 2010.
21. Lobeiras, J.; Amor, M.; Doallo, R. BPLG: A tuned butterfly processing library for GPU architectures. *Int. J. Parallel Program.* **2015**, *43*, 1078–1102. [[CrossRef](#)]
22. Vasilache, N.; Johnson, J.; Mathieu, M.; Chintala, S.; Piantino, S.; LeCun, Y. Fast convolutional nets with fbfft: A GPU performance evaluation. *arXiv* **2014**, arXiv:1412.7580.
23. Khadem, B.; Ghasemi, R. Improved algorithms in parallel evaluation of large cryptographic S-boxes. *Int. J. Parallel Emergent Distrib. Syst.* **2020**, *35*, 461–472. [[CrossRef](#)]
24. Kim, G.; Jeon, Y.; Kim, J. Speeding up LAT: Generating a Linear Approximation Table Using a Bitsliced Implementation. *IEEE Access* **2022**, *10*, 4919–4923. [[CrossRef](#)]
25. Preneel, B.; BRAEKEN, A. *Cryptographic Properties of Boolean Functions and S-Boxes*; Departement elektrotechniek (ESAT): Leuven, Belgium, 2006.
26. Chabaud, F.; Vaudenay, S. Links between differential and linear cryptanalysis. In *Proceedings of the Workshop on the Theory and Application of Cryptographic Techniques*; Springer: Berlin/Heidelberg, Germany, 1994; pp. 356–365.
27. Bakoev, V. Fast computing the algebraic degree of Boolean functions. In *Proceedings of the Algebraic Informatics: 8th International Conference, CAI 2019, Niš, Serbia, 30 June–4 July 2019*; Springer: Berlin/Heidelberg, Germany, 2019; pp. 50–63.
28. Good, I.J. The interaction algorithm and practical Fourier analysis. *J. R. Stat. Soc. Ser. B (Methodological)* **1958**, *20*, 361–372. [[CrossRef](#)]
29. Hughes, C.J. Single-instruction multiple-data execution. *Synth. Lect. Comput. Archit.* **2015**, *10*, 1–121.
30. Joux, A. *Algorithmic Cryptanalysis*; Chapman and Hall/CRC: Boca Raton, FL, USA, 2009.
31. Zhang, X.M.; Zheng, Y.; Imai, H. Relating differential distribution tables to other properties of substitution boxes. *Des. Codes Cryptogr.* **2000**, *19*, 45–63. [[CrossRef](#)]

32. Bikov, D.; Bouyukliev, I. Parallel fast Walsh transform algorithm and its implementation with CUDA on GPUs. *Cybern. Inf. Technol.* **2018**, *18*, 21–43. [[CrossRef](#)]
33. Bikov, D.; Bouyukliev, I. Parallel fast Möbius (Reed-Muller) transform and its implementation with CUDA on GPUs. In Proceedings of the International Workshop on Parallel Symbolic Computation, Kaiserslautern, Germany, 23–24 July 2017; pp. 1–6.

**Disclaimer/Publisher’s Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.