About BoolSPL: A library with parallel algorithms for Boolean functions and S-boxes for GPU

Current release: v0.2

What is BoolSPL?

BoolSPL (Boolean S-box parallel library for GPU) provides, reusable software components for every layer of the CUDA programming model [4]. BoolSPLG is a library consisting procedures for analysis and compute cryptographic properties of Boolean and Vector Boolean function (S-box). Our procedures have function for auto grid configuration. Most of the functions are designed to compute the data in registers because they offer the highest bandwidth.

Overview of BoolSPLG Basic Procedures

The proposed library implement algorithm as composition of basic function into one parameterized kernel, without care about details of implementation. The building function can be classified into computation (Butterfly (FWT, IFWT, FMT, bitwise FMT, min-max), DDT, AlgebraicDegree, ComponentFunction, PowerInt), reordering operations (Copy, MemoryPatern) and support operation reduction (min, max).

Figure 1 presents a scheme with the classification of the functions used to build procedures for computing the cryptographic properties of Boolean and Vector Boolean function. The solid line indicates a dependency while the dashed line represents an optional component.

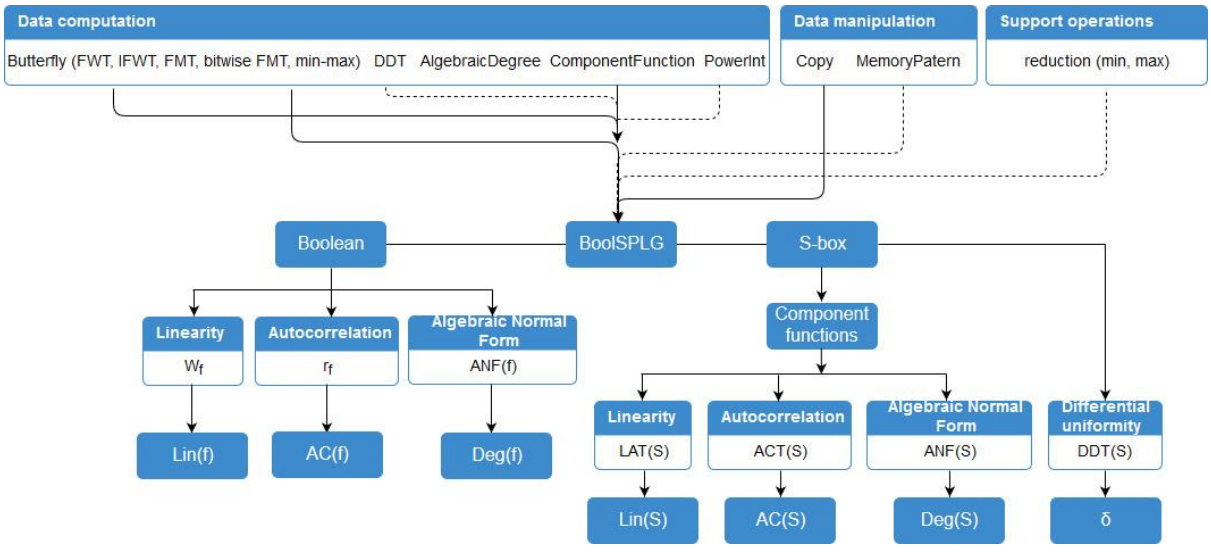

Figure 1. Classification and module dependencies of the building blocks involved in the library

Our library contains the following butterfly algorithms: binary Fast Walsh Transforms (FWT), binary Inverse Fast Walsh Transforms (IFWT), binary Fast Mobius Transforms (FMT), Bitwise binary Fast Mobius Transform (bitwise FMT) and butterfly Min-Max. There are additional algorithms and function for computing DDT, algebraic normal form, component function and auxiliary function reduction for maintaining necessary operations figure 1.

File BoolSPLG_v02 contain all declaration of host and device functions and procedures. On the end of declaration there is comment for the library version of the function/procedure. Included procedure in BoolSPLG compute next cryptographic properties: $W_f(f)$ (Walsh spectra of Boolean function), Lin(f) (Linearity of Boolean function), LAT(S) (Linear Approximation Table of S-box), Lin(S) (Linearity of S-box), $r_f(f)$ (Autocorrelation Spectrum of Boolean function), AC(f) (Autocorrelation of Boolean function), ACT(S) (Autocorrelation spectrum of S-box), AC(S) (Autocorrelation of S-box), ANF(f) (Algebraic Normal Form of Boolean function), ANF(S) (Algebraic Normal Form of S-box), Deg(f) (Algebraic Degree of Boolean function), Deg(S) (Algebraic Degree of S-box), DDT(S) (Difference Distribution Table), δ (Differential uniformity) and S_b (Component function of S-box) [1].

Device functions

```
//Declaration for Boolean GPU device functions

//GPU Fast Walsh Transform
extern __global__ void fwt_kernel_shfl_xor_SM(int *VectorValue, int *VectorValueRez,
int step); //0.1
extern __global__ void fwt_kernel_shfl_xor_SM_MP(int *VectorValue, int fsize, int
fsize1); //0.1

//GPU Fast Mobius Transform
extern __global__ void fmt_kernel_shfl_xor_SM(int * VectorValue, int * VectorRez, int
sizefor); //0.1
extern __global__ void fmt_kernel_shfl_xor_SM_MP(int * VectorValue, int fsize, int
fsize1); //0.1

//GPU Bitwise Fast Mobius Transform
extern __global__ void fmt_bitwise_kernel_shfl_xor_SM(unsigned long long int *vect,
unsigned long long int *vect_out, int sizefor, int sizefor1); //v0.2
extern __global__ void fmt_bitwise_kernel_shfl_xor_SM_MP(unsigned long long int *
VectorValue, int fsize, int fsize1); //v0.2

//GPU compute Algebraic Degree
extern __global__ void kernel_AD(int *Vec); //0.1
extern __global__ void kernel_bitwise_AD(unsigned long long int *NumIntVec, int
*Vec_max_values, int NumOfBits); //v0.2

//GPU Inverse Fast Walsh Transform
extern __global__ void ifmt_kernel_shfl_xor_SM(int * VectorValue, int *
VectorValueRez, int step); //0.1
extern __global__ void ifmt_kernel_shfl_xor_SM_MP(int * VectorValue, int fsize, int
fsize1); //0.1

//GPU Min-Max Butterfly
extern __global__ void Butterfly_max_min_kernel_shfl_xor_SM(int *VectorValue, int
*VectorValueRez, int step); //v0.2
extern __global__ void Butterfly_max_min_kernel_shfl_xor_SM_MP(int * VectorValue, int
fsize, int fsize1); //v0.2

extern __global__ void ifmt_kernel_shfl_xor_SM_Sbox(int * VectorValue, int *
VectorValueRez, int step); //0.1

//Declaration for S-box GPU device functions

//GPU Bitwise Fast Mobius Transform
extern __global__ void fmt_bitwise_kernel_shfl_xor_SM_Sbox(unsigned long long int
*vect, unsigned long long int *vect_out, int sizefor, int sizefor1); //v0.2
```

```
//GPU compute Algebraic Degree
extern __global__ void kernel_AD_Sbox(int *Vec); //0.1
extern __global__ void kernel_bitwise_AD_Sbox(unsigned long long int *NumIntVecANF,
int *max_values, int NumOfBits); //v0.2

//GPU Difference Distribution Table
extern __global__ void DDTFnAll_kernel(int *Sbox_in, int *DDT_out, int n); //0.1
extern __global__ void DDTFnVec_kernel(int *Sbox_in, int *DDT_out, int row); //0.1

//GPU S-box Component functions
extern __global__ void ComponentFnAll_kernel(int *Sbox_in, int *CF_out, int n); //0.1
extern __global__ void ComponentFnVec_kernel(int *Sbox_in, int *CF_out, int row); //0.1
```

Blocks passing data

Hardware limitation resources (memory, number of thread per block) influence the design of algorithms. If dimension of input array bigger from 2^{10} entries, in some point it is need rearranges of data between memory from different blocks. The memory pattern (use in [2, 3]) rearranges the shared memory in such a way that the memory elements from different blocks (intermediate results) are set in order to perform butterfly algorithms from the beginning. Rearranges of shared memory data between different block is made by pointers, without worrying about the number of blocks cooperating.

S-box, component function

In order to study the cryptographic properties of a vectorial Boolean function we need to consider all non-zero linear combinations of the coordinates of the vectorial Boolean function [1]. We implement two similar algorithms for computing component functions. The first one (ComponentFnAll kernel) compute all component function at once ($n \leq 10$) and the second one (ComponentFnVec_kernel) compute component function one after another ($n > 10$). This separation is caused by the hardware resource limitation (memory, number of thread per block).

First device function (ComponentFnAll kernel), use two array and integer block size. Input array contain vector Boolean function. Every one of the element represent integer which binary representation is column from vector Boolean matrix representation. Output $2^n \times 2^n$ array contain all component function. Number of component function, number of blocks, size of component function and the block size are equal. Threads from one block computed one component function. Every block, have copy from the vector Boolean function, in fact values from the vector Boolean function is write in threads local register value. The output array CF_out contain sequence of component function and it is $2^n \times 2^n$ dimensional Figure 2.

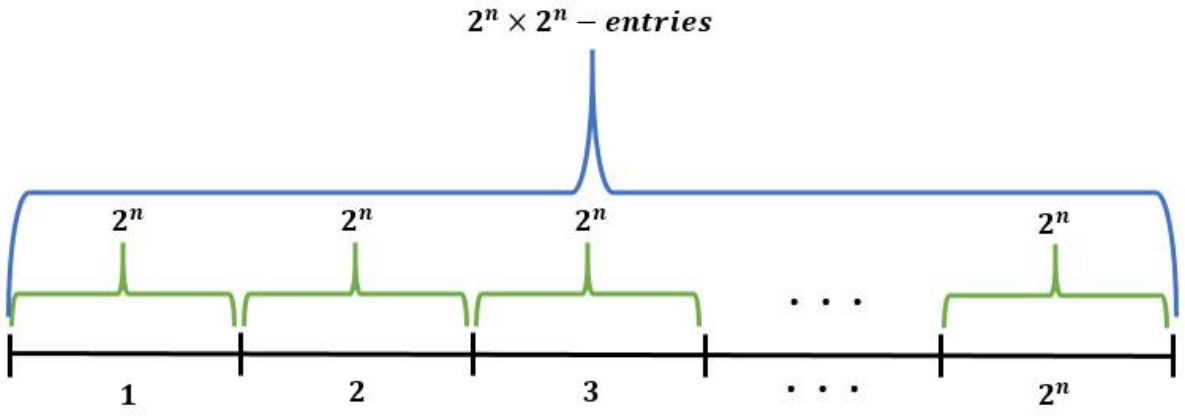

Figure 2. S-box, array contain all component function

Function support operations, reduction

In some of the algorithms there are need maximum or minimum value depending from the examined properties. General parallel algorithm suitable for problem of this type are known as reduction. Very important to notice that we need to find absolute maximum or minimum (minimum will be implemented in next library release) value. This is the reason for our modified reduction implementation. But we will not give detail of implementation because algorithm is well known.

```
//Declaration for Max - Min Reduction function
int runReductionMax(int size, int *d_idata); //0.1
int runReductionMin(int size, int *d_idata); //v0.2

//Declaration for Butterfly max function
int Butterfly_max_kernel(int sizeSbox, int *device_data); //v0.2
```

Grid configuration

One of the basic requirements for to obtain optimal performance is assignment of maximum parallelism that we control with configuration of the grid. Optimal setting and configuration of the parallel function's grid is limited by the available hardware resources. Our procedures automatically adjust the grid. Resource that impact performing are numbers of registers per thread, shared memory per block, number of running block per SM and number of threads per block.

```
//Function: Set GRID
inline void setgrid(int size); //0.1
inline void setgridBitwise(int size); //0.2
```

BoolSPLG procedures

The procedures are combination of functions that unite sequence of different algorithms, and at the same time they maintain the correct parallel distribution of work. Introduced device function give possibility for design different compact algorithm. This main parallel function beside combining function, perform grid configuration.

```
//Declaration for Boolean procedures

// Wf(f) Walsh spectra, return Lin(f) Linearity of Boolean function
int WalshSpecTranBoolGPU(int *device_Vect, int *device_Vect_rez, int size, bool
returnMaxReduction); //0.1 BoolFWT_compute
// ANF(f) Algebraic Normal Form of Boolean function
void MobiusTranBoolGPU(int *device_Vect, int *device_Vect_rez, int size);
       //0.1 BoolFMT_compute
//return - deg(f) Algebraic Degree of Boolean function
int AlgebraicDegreeBoolGPU(int *device_Vect, int *device_Vect_rez, int size);
       //0.1 BoolAD_compute
// rf(f) Autocorrelation Spectrum, return AC(f) Autocorrelation of Boolean function
int AutocorrelationTranBoolGPU(int *device_Vect, int *device_Vect_rez, int size, bool
returnMaxReduction); //0.1BoolAC_compute

// Wf(f) Walsh spectra, return Lin(f) Linearity of Boolean function (Butterfly max)
int WalshSpecTranBoolGPU_ButterflyMax(int *device_Vect, int *device_Vect_rez, int
size, bool returnMaxReduction); //v0.2 BoolFWT_compute
//return - deg(f) Algebraic Degree of Boolean function (Butterfly max)
int AlgebraicDegreeBoolGPU_ButterflyMax(int *device_Vect, int *device_Vect_rez, int
size);                                          //v0.2 BoolAD_compute
// rf(f) Autocorrelation Spectrum, return AC(f) Autocorrelation of Boolean function
(Butterfly max)
int AutocorrelationTranBoolGPU_ButterflyMax(int *device_Vect, int *device_Vect_rez,
int size, bool returnMaxReduction);  //v0.2 BoolAC_compute
```

```
//Declaration for Bitwise Boolean procedures

// ANF(f) Algebraic Normal Form of Boolean function
void BitwiseMobiusTranBoolGPU(unsigned long long int *device_Vect, unsigned long long
int *device_Vect_rez, int size); //v0.2 Bitwise BoolFMT_compute
//return - deg(f) Algebraic Degree of Boolean function (Butterfly max)
int BitwiseAlgebraicDegreeBoolGPU_ButterflyMax(unsigned long long int *device_Vect,
unsigned long long int *device_Vect_rez, int *device_Vec_max_values, int
*host_Vec_max_values, int size); //v0.2

//Declaration for S-box procedures

// LAT(S) Linear Approximation Table, return Lin(S) Linearity of S-box
int WalshSpecTranSboxGPU(int *device_Sbox, int *device_CF, int *device_LAT, int
sizeSbox); //0.1
// ANF(S) Algebraic Normal Form of S-box and return deg(S) Algebraic Degree of S-box
int MobiusTranSboxADGPU(int *device_Sbox, int *device_CF, int *device_ANF, int
sizeSbox); //0.1
// ACT(S) Autocorrelation Table, return AC(S) Autocorrelation of S-box
int AutocorrelationTranSboxGPU(int *device_Sbox, int *device_CF, int *device_ACT, int
sizeSbox); //0.1
// DDT(S) Difference Distribution Table, return δ Differential uniformity of S-box
int DDTSboxGPU(int *device_Sbox, int *device_DDT, int sizeSbox); //0.1

// LAT(S) Linear Approximation Table, return Lin(S) Linearity of S-box (Butterfly max)
int WalshSpecTranSboxGPU_ButterflyMax(int *device_Sbox, int *device_CF, int
*device_LAT, int sizeSbox, bool returnMax); //v0.2
// ANF(S) Algebraic Normal Form of S-box
void MobiusTranSboxGPU(int *device_Sbox, int *device_CF, int *device_ANF, int
sizeSbox); //v0.2
// return deg(S) Algebraic Degree of S-box (Butterfly max)
int AlgebraicDegreeSboxGPU_ButterflyMax(int *device_Sbox, int *device_CF, int
*device_ANF, int sizeSbox); //v0.2
// ACT(S) Autocorrelation Table, return AC(S) Autocorrelation of S-box (Butterfly max)
int AutocorrelationTranSboxGPU_ButterflyMax(int *device_Sbox, int *device_CF, int
*device_ACT, int sizeSbox, bool returnMax); //v0.2
// DDT(S) Difference Distribution Table, return δ Differential uniformity of S-box
(Butterfly max)
int DDTSboxGPU_ButterflyMax(int *device_Sbox, int *device_DDT, int sizeSbox, bool
returnMax); //v0.2

//Declaration for Bitwise S-box procedures

// ANF(f) Algebraic Normal Form of S-box
void BitwiseMobiusTranSboxGPU(int *host_Sbox, int *host_Vect_CF, unsigned long long
int *host_NumIntVecCF, unsigned long long int *device_NumIntVecCF, unsigned long long
int *device_NumIntVecANF, int sizeSbox); //v0.2
//return - deg(f) Algebraic Degree of S-box (Butterfly max)
int BitwiseAlgebraicDegreeSboxGPU_ButterflyMax(int *host_Sbox, int *host_Vect_CF, int
*host_max_values, unsigned long long int *host_NumIntVecCF, unsigned long long int
*device_NumIntVecCF, unsigned long long int *device_NumIntVecANF, int
*device_Vec_max_values, int sizeSbox); //v0.2
```

How do I get started using BoolSPL?

BoolSPL is implemented as a C++ header library. There is no need to "build" BoolSPL separately. To use BoolSPL primitives in your code, simply:

1. Download and unzip the latest BoolSPL distribution from the Downloads section and extract the contents of the zip file to a directory. You need to install (copy) only "BoolSPL" directory

from the main BoolSPL-vx.x directory. We suggest installing BoolSPL to the CUDA include directory, which is usually:

- /usr/local/cuda/include/ on a Linux and Mac OSX;
- C:\CUDA\include\ on a Windows system.

Example: C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v8.0\include\;

If you are unable to install BoolSPL to the CUDA include directory, then you can place BoolSPL somewhere in your home directory, for example: /home/nathan/libraries/.

2. #include the "umbrella" <BoolSPL/BoolSPLG_v0.cuh> header file in your CUDA C++ sources;
3. Compile your program with NVIDIA's nvcc CUDA compiler, specifying a -I<path-to-BoolSPL> include-path flag to reference the location of the BoolSPL header library.

Examples

BoolSPL distribution directory contain “examples” directory with examples (Boolean and S-box) programs. For the examples to work there is need to include (add) the additional header files from the directory “help and additional header files”. This additional header files contain CPU Boolean and S-box function used for comparison and checking the obtained results from GPU functions.

Reference and Publications related to the BoolSPL library

[1] Bikov D., and I. Bouyukliev, BoolSPLG: A library with parallel algorithms for Boolean functions and S-boxes for GPU, preprint.

[2] Bikov D., Bouyukliev I.: Parallel Fast Walsh Transform Algorithm and its implementation with CUDA on GPUs. Cybernetics and Information Technologies. Cybernetics and Information Technologies 18, 21–43 (2018).

[3] Bikov D., and I. Bouyukliev, Parallel Fast Mobius (Reed-Muller) Transform and its Implementation with CUDA on GPUs, Proceedings of PASCO 2017, Kaiserslautern, Germany, Germany — July 23 - 24, 2017, ISBN: 978-1-4503-5288-8 (**improvement presented in this publication are implemented in current v0.2 BoolSPL library**)

[4] CUDA homepage, Availaible on: http://www.nvidia.com/object/cuda_home_new.html

Additional - Reference and Publications related to the BoolSPL library

[1] I. Bouyukliev, D, Bikov, Applications of the binary representation of integers in algorithms for boolean functions, Proceedings of the Forty Fourth Spring Conference of the Union of Bulgarian Mathematicians SOK “Kamchia”, (2015), pp.161-166, ISSN: 1313-3330

[2] D. Bikov, I. Bouyukliev, Walsh Transform Algorithm and its Parallel Implementation with CUDA on GPUs, Proceedings of 25 YEARS FACULTY OF MATHEMATICS AND INFORMATICS, Veliko Tarnovo, Bulgaria, (2015), pp. 29-34, ISBN: 978-619-00-0419-6

[3] D. Bikov, I. Bouyukliev, A. Stojanova, Benefit of Using Shared Memory in Implementation of Parallel FWT Algorithm with CUDA C on GPUs, Proceedings of 7th International Conference Information Technologies and Education Development, Zrenjanin, Serbia, (2016) pp.250-256, ISBN 978-86-7672-285-3

[4] I. Bouyukliev, D. Bikov, S. Bouyuklieva, S-Boxes from Binary Quasi-Cyclic Codes, Electronic Notes in Discrete Mathematics Volume 57, (2017), pp. 67–72, SJR 0.320

[5] D. Bikov, I. Bouyukliev, S. Bouyuklieva, Bijective S-boxes of Different Sizes Obtained from Quasi-Cyclic Codes, submitted.

ACKNOWLEDGMENTS

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

The development of the library is supported by Bulgarian Science Fund under Contract DN-02-2/13.12.2016