On Hardware Implementation of Tang-Maitra Boolean Functions

Mustafa Khairallah[1], Anupam Chattopadhyay[1], Bimal Mandal[2], and Subhamoy Maitra[2]

[1] Nanyang Technological University, Singapore.
mustafam001@e.ntu.edu.sg, anupam@ntu.edu.sg
[2] Indian Statistical Institute, Kolkata, India.
bimalmandal90@gmail.com, subho@isical.ac.in

Abstract. In this paper, we investigate the hardware circuit complexity of the class of Boolean functions recently introduced by Tang and Maitra (IEEE-TIT 64(1): 393 402, 2018). While this class of functions has very good cryptographic properties, the exact hardware requirement is an immediate concern as noted in the paper itself. In this direction, we consider different circuit architectures based on finite field arithmetic and Boolean optimization. An estimation of the circuit complexity is provided for such functions given any input size n. We study different candidate architectures for implementing these functions, all based on the finite field arithmetic. We also show different implementations for both ASIC and FPGA, providing further analysis on the practical aspects of the functions in question and the relation between these implementations and the theoretical bound. The practical results show that the Tang-Maitra functions are quite competitive in terms of area, while still maintaining an acceptable level of throughput performance for both ASIC and FPGA implementations.

Keywords: Boolean Functions, Bent Functions, Cryptology, Finite Fields, Hardware Implementation, Stream Cipher.

1 Introduction

Boolean functions are used in many domains such as sequences, cryptography, coding theory and combinatorics. In many cryptosystems, for example, linear/non-linear feedback shift register (LFSR and NFSR) based stream ciphers, a Boolean function is used to combine the outputs of several LFSRs/NFSRs. A special class of Boolean functions, having the maximum distance from the set of all affine functions, are known as bent functions. Introduced by Rothaus in 1976 [Rot76], these functions maximally resist any kind of affine approximations.

However, bent functions are not directly used as cryptographic primitives, since they are not balanced. Moreover, bent functions $f \in \mathcal{B}_n$ (we denote the set of n-variable Boolean functions as $\mathcal{B}_n$) exist only for even number of variables and its degree is at most $\frac{n}{2}$. Dillon [Dil74] constructed a class of bent functions, which is called the partial spread ($\mathcal{PS}$) class of bent functions, and the bent functions in $\mathcal{PS}_{ap}$ is a subclass of $\mathcal{PS}$. Another generic class of bent functions, called Maiorana–McFarland class (denoted by $\mathcal{M}$), was introduced in [McF73] and further investigated in [Dil74]. Further, Dobbertin [Dob94] and Carlet [Car93] constructed different classes of bent functions. For more details of bent Boolean functions, we refer to [MM16,CS09]. Recently, Tang and Maitra [TM18, Construction 1] constructed a class of cryptographically significant balanced Boolean functions by modifying a special type of bent functions in $\mathcal{PS}_{ap}$. Such functions have very good nonlinearity and autocorrelation at the same time. Further research in this direction has been reported in [KMT18].

Since the functions of [TM18] are derived from Dillon type bent functions (related to Maiorana-McFarland type functions), the actual hardware should follow from the finite field implementation ideas. This is the task that we take up here. Noting that such functions might be useful in lightweight stream ciphers, we try to see how efficiently one can actually implement such functions. In fact, we note that the 12-variable function requires area less than 500 GE (gate equivalent) which may be embedded in a lightweight stream cipher circuit of 1000 GE. Functions on 22-variables can be implemented with little more than 3000 GE. This underlines that such implementations might be of interest as primitives in stream cipher design.

Our Contributions In this paper, we study the circuit complexity of the Tang-Maitra class of Boolean functions. Since the construction is based on finite field arithmetic, we consider different representations for finite fields $\mathbb{F}_{2^n}$. The first two representations are the polynomial and normal bases representations. The third representation is the discrete-log representation of the finite field elements (though there are some additional complexities that we will discuss later). The results are summed up in the following three lemmas:

1. Lemma 1: The polynomial basis implementation of a Tang-Maitra function on n variables, where n is even, has the circuit complexity bounded by $\mathcal{O}(2^k + k^2)$ and depth of $\mathcal{O}(k)$, where $n = 2k$.
2. Lemma 2: The normal basis implementation of a Tang-Maitra function on n variables, where n is even, has the circuit complexity bounded by $\mathcal{O}(2^k + k^3)$, where $n = 2k$.
3. Lemma 4: The discrete-log implementation of a Tang-Maitra function on n variables, where n is even, has the circuit complexity bounded by $\mathcal{O}(2^k + k^2 + 2^k/k^2)$, where $n = 2k$.

We also propose a general circuit construction for Rotation Symmetric Boolean Functions (RSBF), bounded by $\mathcal{O}(k^2 + 2^k/k^2)$ (Lemma 3). Further, we briefly discuss the relationship between these different representations (Section 3.3) to clarify the well-known understanding of polynomial and normal bases representations related to certain cryptographic properties. Finally, we practically implement several instances of the Tang-Maitra functions for different values of n, where n is the number of variables. We show the implementation results for both ASIC and FPGA, providing a comparison with the hardware implementation GMM class of Boolean functions (Section 4). The results are based on the polynomial representation, which has the lowest asymptotic circuit complexity.

2 Preliminaries

2.1 Finite field $\mathbb{F}_{2^n}$ arithmetic

The finite field $\mathbb{F}_{2^n}$ is an extension field of $\mathbb{F}_2$ of degree n, where $\mathbb{F}_2$ is a prime field of two elements $\{0, 1\}$. In other words, $\mathbb{F}_{2^n}$ is the field of polynomials of degree at most $n-1$ over $\mathbb{F}_2$. It consists of 2^n elements and two basic operations are defined for it:

1. Addition ($\oplus$): polynomial addition modulo 2.
2. Multiplication ($\odot$): polynomial multiplication modulo $F[x]$, where $F[x]$ is an irreducible polynomial of degree n over $\mathbb{F}_2$.

There are three more operations of interest: Squaring, Inversion and the Trace Function, but they can be defined using the former two basic operations. Since there can be more than one irreducible polynomial of degree n, $\mathbb{F}_{2^n}$ according to this definition is not unique. However, all the fields generated by different irreducible polynomial choice are isomorphic, i.e., a certain field can be changed to another one by a permutation of the elements.

The trace function is a frequently referred to, in finite field theory. It is defined as $\mathrm{Tr}_1^n : \mathbb{F}_{2^n} \rightarrow \mathbb{F}_2$,

$$\mathrm{Tr}_1^n(\alpha) = \alpha \oplus \alpha^2 \oplus \alpha^{2^2} \oplus \ldots \oplus \alpha^{2^{n-1}}, \text{ for all } \alpha \in \mathbb{F}_{2^n} \tag{1}$$

We list here certain properties of the trace function Tr_1^n that are important for our work. An interested reader may refer to [LN94] for a complete discussion related to finite fields $\mathbb{F}_{2^n}$. These properties are:

1. $\mathrm{Tr}_1^n(\alpha \oplus \beta) = \mathrm{Tr}_1^n(\alpha) \oplus \mathrm{Tr}_1^n(\beta)$, for all $\alpha, \beta \in \mathbb{F}_{2^n}$.
2. $\mathrm{Tr}_1^n(\alpha^2) = \mathrm{Tr}_1^n(\alpha)$, for all $\alpha \in \mathbb{F}_{2^n}$.

2.2 Boolean functions

Let $\mathbb{F}_2^n$ be the vector space of dimension n over $\mathbb{F}_2$ and any element $x \in \mathbb{F}_2^n$ can be written as $x = (x_{n-1}, \ldots, x_1, x_0)$, where $x_i \in \mathbb{F}_2,\ 0 \leq i \leq n-1$. Any function f from $\mathbb{F}_2^n$ (or $\mathbb{F}_{2^n}$) to $\mathbb{F}_2$ is called Boolean function in n variables. The set of n-variable Boolean functions is denoted by $\mathcal{B}_n$. Any Boolean function $f \in \mathcal{B}_n$ can be represented in a unique way as

$$f(x) = \bigoplus_{\alpha \in \mathbb{F}_2^n} \mu_\alpha x_0^{\alpha_0} x_1^{\alpha_1} \ldots x_{n-1}^{\alpha_{n-1}},$$

for all $x \in \mathbb{F}_2^n$, where $\mu_\alpha \in \mathbb{F}_2$. This polynomial representation is called the algebraic normal form (ANF) of $f \in \mathcal{B}_n$. The algebraic degree of a Boolean function $f \in \mathcal{B}_n$ is defined as $deg(f) = \max_{\alpha \in \mathbb{F}_2^n}\{wt(\alpha) :\ \mu_\alpha \neq 0\}$, where $wt(\alpha)$ is the Hamming weight of $\alpha \in \mathbb{F}_2^n$, defined as $wt(\alpha) = \sum_{i=0}^{n-1} \alpha_i$ (the sum is over the ring of integers). Further more details, we refer to [MM16,CS09].

2.3 $\mathbb{F}_{2^n}$ practical representations

In order to perform operations on finite field elements in practice, we need to represent the elements and operations in terms of binary representations and circuits/algorithms, respectively. In this section, we describe three possible representations, with the advantages and disadvantages of each. An inquisitive reader can find more details in [DIS09].

Polynomial basis Let $\alpha \in \mathbb{F}_{2^n}$ be a root of the irreducible polynomial $F[x]$ used to define $\mathbb{F}_{2^n}$. Hence, $\mathbb{F}_{2^n}$ can be defined as $\{a(\alpha)|a(\alpha) = \bigoplus_{i=0}^{n-1} a_i\alpha^i\}$, where $a_i \in \mathbb{F}_2$. Therefore, $x(\alpha)$ can be represented by the binary string $(x_{n-1}, \ldots, x_1, x_0)$. Operations are performed as follows:

1. *Addition:* Addition is performed using coefficient-wise XOR.
2. *Multiplication:* Multiplication is more complicated. Since multiplication is defined as polynomial multiplication, the circuit complexity is $\mathcal{O}(n^{1+\epsilon})$, where $\epsilon > 0.5$ for the available circuits, as opposed to $\mathcal{O}(n)$ in case of addition. However, we explain the matrix-vector multiplication method, which has a complexity of $\mathcal{O}(n^2)$, as it is useful in discussing the complexity of squaring and the trace function. In the first step, we compute the polynomial $d(x) = a(x) \cdot b(x)$ of degree at most $2n - 2$. This is performed by the equation

$$\begin{bmatrix} d_0 \\ d_1 \\ \vdots \\ d_{n-1} \\ \vdots \\ d_{2n-2} \end{bmatrix} = \begin{bmatrix} a_0 & 0 & 0 & \ldots & 0 & 0 \\ a_1 & a_0 & 0 & \ldots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \\ a_{n-1} & a_{n-2} & a_{n-3} & \ldots & a_1 & a_0 \\ 0 & a_{n-1} & a_{n-2} & \ldots & a_2 & a_1 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \\ 0 & 0 & 0 & \ldots & 0 & a_{n-1} \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix} \tag{2}$$

This step requires n^2 AND gates and $(n-1)^2$ XOR gates. The second step is to compute $c(x) \equiv d(x)\ mod\ F[x]$, which is performed by the equation

$$\begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{I}\ \boldsymbol{R} \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ \vdots \\ d_{n-1} \\ \vdots \\ d_{2n-2} \end{bmatrix} \tag{3}$$

where $\boldsymbol{I}$ is an $n \times n$ identity matrix and $\boldsymbol{R}$ is a matrix whose elements are functions of $F[x]$. This step requires $wt(\boldsymbol{R})$ XOR gates and it can be lowered by choosing a suitable $F[x]$, e.g., a trinomial.

3. *Squaring:* It can be implemented using only reduction (and wiring). This is due to the fact that $a(x) \cdot a(x) = a_{n-1}x^{2n-2} \oplus a_{n-2}x^{2n-4} \oplus \ldots \oplus a_1x^2 \oplus a_0$. Hence, only the last step of the matrix-vector multiplication method is required, leading to a circuit with $wt(\boldsymbol{R})$ XOR gates.
4. *Inversion:* Inversion is the most complex operation in polynomial basis. It requires $\mathcal{O}(n^2)$ gates. For example, a straightforward implementation of the Extended Euclidean Algorithm for Inversion requires around $(24n^2 + 24n)$ MUXes, $(n^2 + n)$ AND gates and $(5n^2 + 5n)$ XOR gates. Complex Boolean optimization heuristics are usually used to design more efficient circuits.
5. *Trace Function* Tr_1^n*:* Requires $n-1$ squarings and $n-1$ XOR gates, as only the constant coefficients of α^{2^i} need to be added. The overall complexity is $(wt(\boldsymbol{R}) + 1)(n-1)$ XOR gates.

Normal basis The polynomial basis representation can be viewed as a vector space representation with the basis $\{1, \alpha, \alpha^2, \alpha^3, \ldots, \alpha^{n-1}\}$. Since it is known that the finite field representation is not unique, it is useful to look for other suitable bases that can be used. One special basis is the normal basis, which is defined as $\{\beta, \beta^2, \beta^{2^2}, \ldots, \beta^{2^{n-1}}\}$, where $\beta^{2^n} = \beta$ and $\beta^i \neq \beta$ for all $1 < i < 2^n$, with β is a primitive element of $\mathbb{F}_{2^n}$. $\mathbb{F}_{2^n}$ is defined as $\{a(\beta) | a(\beta) = \bigoplus_{i=0}^{n-1} a_i\beta^{2^i}\}$, where $a_i \in \mathbb{F}_2$. Since the vectors of the normal basis are linearly independent, all the elements of $\mathbb{F}_{2^n}$ can be generated as linear combinations of the basis vectors.

Again, $x(\beta)$ can be represented by the binary string $(x_{n-1}, \ldots, x_1, x_0)$ and addition is the same as in the case of polynomial basis. While multiplication is even more complex than in the case of polynomial basis ($\mathcal{O}(n^{1+\epsilon})$, where $\epsilon > 0.6$), two of the operations we are interested in are very simple using the normal basis representation; Squaring and Tr_1^n. Equation (4) shows that squaring in $\mathbb{F}_{2^n}$ is a linear operation, while Equation (5) is an application of Fermat's Little Theorem to $\mathbb{F}_{2^n}$. Using these two properties, it can be shown that in the normal basis representation x^2 is just a cyclic shift of the bit representation of x and $\mathrm{Tr}_1^n(x) = \bigoplus_{i=0}^{n-1} x_i$.

$$(\beta \oplus \gamma)^2 = \beta^2 \oplus \gamma^2, \text{ for all } \beta, \gamma \in \mathbb{F}_{2^n} \tag{4}$$

$$\beta^{2^n} = \beta, \text{ for all } \beta \in \mathbb{F}_{2^n} \tag{5}$$

Inversion, however, is not efficient in the normal basis. Using Fermat's little theorem, it requires circuit complexity of $\mathcal{O}(n^{2+\epsilon})$, where $\epsilon > 0.6$, since exponentiation requires $n-2$ multiplication operations. On the other hand, for special choices of n, more efficient circuits can be implemented. For example, for $n = 2^r + 1$ only $\log(n-1)$ multiplications are required. Similar results are available for the cases when $n = m * k$, using Tower Fields [IT88].

Discrete-log representation Another interesting representation of finite field elements is the discrete logarithmic representation. Let α be a primitive element of $\mathbb{F}_{2^n}$. Then $\alpha^{2^n} = \alpha$, $\alpha^i \neq \alpha$, for all $0 \leq i \leq 2^n - 2$ and any nonzero element $x \in \mathbb{F}_{2^n}$ is represented as $x = \alpha^i$ for a $0 \leq i \leq 2^n - 2$. A special representation is used for $x = 0$. Hence, only the exponent i needs to be stored. Since α has order of $2^n - 1$, each of the $2^n - 1$ non-zero field elements have unique representation $\in \mathbb{Z}_{2^n-1}$. Besides, $i = 2^n - 1$ is used to represent $x = 0$. This representation is very useful for applications that include a lot of multiplication, squaring and inversion operations.

However, the transformation to/from discrete-log is a non-linear operation, and the relation between this representation and the polynomial basis representation is non-linear. Hence, it cannot be used directly for the Tang-Maitra functions without implementing this non-linear transformation increasing the circuit complexity (details are given in Section 3.3). This representation is discussed in details in Appendices A and B as the cost of such non-linear transformations cannot be estimated immediately.

2.4 The Tang-Maitra Class of Functions

Tang and Maitra [TM18] constructed a class of Boolean functions with good cryptographic properties by modifying $\mathcal{PS}_{ap}$, a subclass of partial spread, as follow:

Construction 1 [TM18, Construction 1] *Let $n = 2k$ and $\lambda, \mu \in \mathbb{F}_{2^n}^*$, where $k \geq 9$ is an odd integer. An n-variable Boolean function over $\mathbb{F}_{2^n}$ is defined as follows*

$$f(x, y) = \begin{cases} h_0(y) & \textit{if } x = 0 \\ h_1(y) & \textit{if } x = \mu \\ \mathrm{Tr}_1^k(\frac{\lambda x}{y}) & \textit{otherwise} \end{cases} \tag{6}$$

Here we assume that $\mathrm{Tr}_1^k(\frac{\lambda x}{0}) = 0$, for all $x \in \mathbb{F}_{2^k}$. The functions h_0 and h_1 must satisfy some cryptographic properties, which can be found in [TM18,KMT18]. In Section 3, we provide circuits for Construction 1 using different finite field representations and derive an estimation of its circuit complexity.

3 Circuit Architectures of the Tang-Maitra Class of Functions

In order to study the hardware complexity of the Tang-Maitra class of functions, we need to divide it into its smaller components, which are:

1. Three equality comparators ($x = 0,\ x = \mu,\ y = 0$).
2. Finite field inverter y^{-1}.
3. Constant multiplier λx.
4. Finite field multiplier $(\lambda x) \cdot y^{-1}$.
5. The trace function Tr_1^k.
6. The circuits for h_0 and h_1.
7. A 4×1 multiplexer.
8. A 3×2 encoder to convert the output of the three comparator into 2 selection bits.

From the previous decomposition, we can already observe some of the properties of the hardware combinational circuit and also some simple optimizations. First, the cost of the multiplexer and encoder is small, constant and does not depend on k. Precisely, for any k, 6 AND gates and 3 OR gates are needed for the 4×1 multiplexer and the required encoder can be implemented using 3 AND gates and 2 OR gates. Second, an obvious optimization is to choose $\lambda \equiv e$ where e is the multiplicative identity element of $\mathbb{F}_{2^k}$. Hence, regardless of the finite field representation use, the cost for the constant multiplication λx is zero. Third, while the comparator cost is linear in k, precisely k XNOR gates and $k-1$ AND gates, we can choose μ in a way that reduces the overall cost of the three comparators. We consider, μ such that consider $wt(\mu \oplus 0) = 1$. The two k-bit comparators $x = 0$ and $x = \mu$ can be implemented using only one $(k-1)$-bit comparators, 2 single bit comparators and 2 extra AND gates. Overall, the three comparators will require $2k+1$ XNOR gates and $2k-1$ AND gates.

In the rest of this section, we discuss the overall circuit using different finite field representations, providing estimations for the cost using each of them. Since h_0 and h_1 are constructed as Boolean functions and not as finite field functions, we consider their cost to be roughly the same for all representations. In the next section, we discuss their cost more precisely. For all figures, thick arrows represent k-bit buses, blue blocks have $poly(k)$ cost, green blocks have constant cost and red boxes represent blocks of unknown cost.

3.1 Circuit based on polynomial basis

Lemma 1. *The polynomial basis implementation of the Tang-Maitra function of n variables, where n is even, has a circuit complexity bounded by $\mathcal{O}(2^k + k^2)$ and depth of $\mathcal{O}(k)$, where $n = 2k$.*

Figure 1 shows the polynomial basis circuit. The first branch for the multiplexer is the default option, which computes $\mathrm{Tr}_1^k(\frac{\lambda x}{y})$. Addition over $\mathbb{F}_{2^k}$ using polynomial basis is carry-free. Besides, 0 is represented as the binary string 0^k and 1 is represented the binary string $0^{k-1}1$. Hence, the addition part of Tr_1^k can be performed by adding only the least significant bits, requires $k-1$ XOR gates, instead of $k(k-1)$. Choosing $\lambda \equiv e$, the cost of this branch is the cost of 1 inversion, 1 multiplication, $k-1$ squaring and $k-1$ XOR gates. Multiplication costs $\mathcal{O}(k^{1+\epsilon})$, where $0.5 \leq \epsilon \leq 1$. The cost for every squaring is $\mathcal{O}(k)$, and inversion has cost of $\mathcal{O}(k^2)$. The overall complexity of this branch is $\mathcal{O}(k^2)$ and depth of $\mathcal{O}(k)$. However, as explained in Section 2.3, this complexity can be bad in practice due to large coefficients. Besides, the circuit complexity of the h_0 and h_1 is bounded by $\mathcal{O}(2^k)$.

3.2 Circuit based on normal basis

Lemma 2. *The normal basis implementation of the Tang-Maitra function of n variables, where n is even, has a circuit complexity bounded by $\mathcal{O}(2^k + k^3)$, where $n = 2k$.*

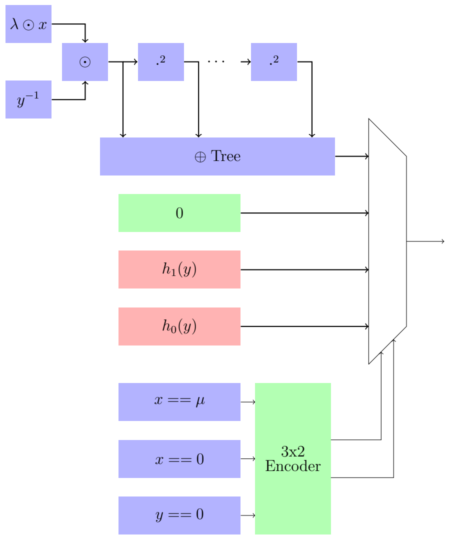

Fig. 1: Polynomial basis circuit for a Tang-Maitra function

Figure 2 shows the normal basis circuit, it is similar to the polynomial basis circuit, from a high-level point of view. However, since the trace function is implemented as the XOR of all input bits, squaring is not required. The complexity of the first branch is $\mathcal{O}(k^3)$, but it should be either smaller than or comparable to the polynomial basis circuit in practice. Besides, for certain choices of k (e.g. $k = 2^r+1$ or $k = m(2^r+1)$), the complexity can be in the order of $\mathcal{O}(k^2 \cdot \log(k))$. λ is chosen as e and $\mu = \beta$.

3.3 Comments on Different Representations

The Tang-Maitra functions are defined as finite field functions. However, the cryptographic properties of interest were evaluated for the Boolean circuit generated by implementing the polynomial representation of the underlying field. An important question is: are the cryptographic properties of the Tang-Maitra functions preserved under the change of basis/representation? The answer to that question is that if the change of representation operation is a linear (or affine) operation, they are preserved. Hence, both the polynomial and normal bases representations have the same cryptographic properties. However, since the transformation to/from discrete-log is a non-linear operation, it cannot be used directly without adjusting the input.

Let α be a primitive element of $\mathbb{F}_{2^k}$, and B_p^k and B_n^k be the polynomial and normal bases of $\mathbb{F}_{2^k}$ over $\mathbb{F}_2$, respectively. Then

$$B_p^k = \{1, \alpha, \alpha^2, \ldots, \alpha^{k-1}\}, \quad B_n^k = \{\alpha, \alpha^2, \ldots, \alpha^{2^{k-1}}\}.$$

Thus, any element $x \in \mathbb{F}_{2^k}$ can be written as $x = \bigoplus_{j=0}^{k-1} a_j \alpha^j$ and also $x = \bigoplus_{j=0}^{k-1} b_j \alpha^{2^j}$, where $a_j, b_j \in \mathbb{F}_2$, $0 \le j \le k-1$. The binary strings $(a_{k-1}, \ldots, a_1, a_0)$

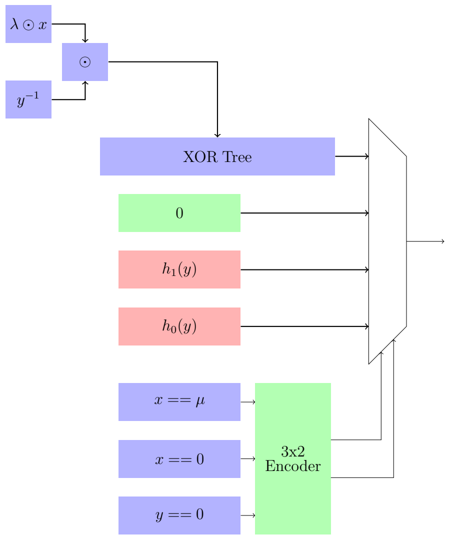

Fig. 2: Normal basis circuit for a Tang-Maitra function

and $(b_{k-1}, \ldots, b_1, b_0)$ are called the binary representation of x with respect to polynomial and normal bases, respectively. It is clear that two binary representations of $\mathbb{F}_{2^k}$ using two bases, normal and polynomial bases, are related by a linear nonsingular mapping.

For discrete-log representation of $\mathbb{F}_{2^k}$, let $0 = (1, 1, \ldots, 1)$, all one vector of $\mathbb{F}_2^k$, and $\alpha^i = (c_{k-1}, \ldots, c_1, c_0)$, where $i = \sum_{j=0}^{k-1} c_j 2^j$ and $c_j \in \mathbb{F}_2,\ 0 \leq j \leq k-1$, for all $0 \leq i \leq 2^k - 2$. In this representation, it is not possible to write all elements of $\mathbb{F}_{2^k}$ in the linear combination of k linearly independent elements and there is no linear (or affine) mapping between discrete-log and normal (or polynomial) basis representations.

We know that cryptographic properties of a Boolean function such as algebraic degree, balancedness, Walsh–Hadamard spectra, autocorrelation spectra, nonlinearity are invariant under the nonsingular affine transformations. But, if the transformations are not affine, then these properties may or may not be same.

For example let $k = 3$ and α be a primitive element of $\mathbb{F}_{2^3}$ such that $\alpha^3 \oplus \alpha^2 \oplus 1 = 0$. Then $B_p^3 = \{1, \alpha, \alpha^2\}$ and $B_n^3 = \{\alpha, \alpha^2, \alpha^4\}$ are the polynomial and normal bases of $\mathbb{F}_2^3$ over $\mathbb{F}_2$, respectively. For discrete-log representation of $\mathbb{F}_{2^3}$, let $0 = (1, 1, 1)$ and $\alpha^i = (c_2, c_1, c_0)$, where $i = \sum_{j=0}^{2} c_j 2^j$ and $c_j \in \mathbb{F}_2,\ 0 \leq j \leq 2$, for all $0 \leq i \leq 6$. The binary representations of $\mathbb{F}_{2^3}$ with respect to B_p^3, B_n^3 and discrete-log are given in Table 1.

One can check that using the nonsingular matrix

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \tag{7}$$

Table 1: Binary representations of $\mathbb{F}_{2^3}$ with respect different bases

$\mathbb{F}_{2^3}$	$\mathrm{Tr}_1^3(x)$	$\mathrm{Tr}_1^3(x^3)$	Polynomial basis	Normal basis	Discrete-log representation
0	0	0	000	000	111
1	1	1	001	111	000
α	1	0	010	001	001
α^2	1	0	100	010	010
α^3	0	1	101	101	011
α^4	1	0	111	100	100
α^5	0	1	011	110	101
α^6	0	1	110	011	110

the binary representation of $\mathbb{F}_{2^3}$ with respect to normal and polynomial bases are related. There is no such linear (or affine) transformation that maps between discrete-log and normal (or polynomial) basis. This we explain by an example here.

Suppose $f, g \in \mathcal{B}_3$ are defined as $f(x) = \mathrm{Tr}_1^3(x)$ and $g(x) = \mathrm{Tr}_1^3(x^3)$, for all $x \in \mathbb{F}_{2^3}$ (defined as in Table 1). The algebraic degrees of f and g are 1 and 2 respectively. Then the algebraic normal form (ANF) of $f(x_1, x_2, x_3)$ over $\mathbb{F}_2^3$ with respect to normal and polynomial bases is $x_1 \oplus x_2 \oplus x_3$, but with respect to discrete-log representation, it becomes $x_1x_2 \oplus x_2x_3 \oplus x_1x_3 \oplus 1$. Moreover, the ANF of $g(x_1, x_2, x_3)$ over $\mathbb{F}_2^3$ with respect to normal an polynomial bases are $x_1x_2 \oplus x_2x_3 \oplus x_1x_3$ and $x_1 \oplus x_2x_3$ respectively, but with respect to discrete-log representation, it is $x_1 \oplus x_2 \oplus x_3 \oplus 1$. Thus, while there are certain advantages in the discrete-log representation, unless the proper nonlinear transformation cannot be decided, the implementation is not complete. Still we explain the implementation in discrete-log domain in Appendices A, B.

3.4 Comparison

We assume the complexity of h_0 and h_1 is the same for the three representations. Comparing the asymptotic complexity of $\mathrm{Tr}_1^k(\frac{\lambda x}{y})$, polynomial basis has the smallest area complexity, while the normal basis has the smallest logical Depth. Moreover, in practice, sometimes the normal basis circuit can have a better area compared to the polynomial basis circuit, especially for good choices of k. The discrete-log circuit (Appendix B) has good parameters for small n, in practice, but due to its non-linearity with respect to the polynomial basis, it needs input adjustment, which can be costly.

4 Practical Implementations

In order to verify the theoretical bounds discussed in the paper, we have implemented the polynomial basis construction for the cases of $n \in \{12, 14, 16, 18, 20, 22\}$ for both FPGA and ASIC.

Table 2: Implementation Results on the Virtex-7 FPGA

n	LUTs	Critical Path (ns)	Logic Levels
12	16	2.25	5
12 [PCZ17]	115	3.47	9
14	42	2.98	6
16	89	3.53	8
16 [PCZ17]	828	4.73	11
18	196	5.13	11
20	640	18.86	33
22	813	23.6	40

For FPGA, the design have been synthesized on Virtex-7 using Xilinx ISE 14.7 design flow, given in Table 2. The results show quadratic and linear increase in the number of LUTs and the critical path, respectively. This conforms with the theoretical estimations in the paper. The technology mapping techniques from [KCP17] have been used to reduce the area, specially for h_0 and h_1 functions. In comparison with the implementations of the GMM functions in [PCZ17], the area and performance on FPGA are both better. Moreover, while the complexity h_0 and h_1 is theoretically exponential, the results show that the dominant factor of the circuit cost in practice is the cost of the trace function. The results also show that the circuit is relatively more costly for $n > 18$.

For ASIC, we have used Synopsys Design Compiler for both the Open Source Nangate 45nm and TSMC 65nm Standard Cell Libraries, given in Tables 3 and 4. The implementations from [PCZ17] have also been synthesized for the same technology. The ASIC results also follow the theoretical estimations. However, compared to [PCZ17], the cost of the Tang-Maitra functions is higher in terms of performance, due to the depth of linear order. With respect to area, the Tang-Maitra functions show a quadratic growth rate, compared to the sub-exponential rate in [PCZ17]. In addition, while the Tang-Maitra ASIC implementations are slower than the GMM ASIC implementations, the clock frequency for 22 variables is still more than 100 MHz, which is faster than the speed required by many practical applications. The hardware results in this paper combined with the cryptographic properties of the Tang-Maitra functions show that they can be a building block for promising cryptographic primitives. Besides, we have not studied the effect of advanced circuit optimization techniques on the ASIC implementations, which can further improve both the area and performance.

5 Conclusion

In this paper, we consider how a recently proposed construction of cryptographically significant Boolean functions in [TM18] can be efficiently implemented. Such functions are derived from Dillon type bent functions and can be interpreted as Maiorana-McFarland bent functions as well. Given that the functions have very good nonlinearity and autocorrelation properties, they may be useful as primitives in hardware stream cipher design. In particular, such functions can

Table 3: Implementation Results on the Nangate 45nm ASIC Technology Library

n	Area(GE)	Critical Path (ns)	Clock Frequency (MHz)
12	580	2.9	344
12 [PCZ17]	381	0.95	1052
14	898	3.49	286
16	1328	4.56	219
16 [PCZ17]	1778	1.47	680
18	1657	5.66	176
20	2602	6.73	148
22	3501	7.95	126

Table 4: Implementation Results on the TSMC 65nm ASIC Technology Library

n	Area(GE)	Critical Path (ns)	Clock Frequency (MHz)
12	477	3.46	289
12 [PCZ17]	378	1.51	1052
14	802	4.79	208
16	1250	5.62	177
16 [PCZ17]	1510	1.64	680
18	1503	6.82	147
20	2366	8.37	119
22	3273	9.84	102

be exploited for lightweight stream ciphers too. The implementation methods follow different ideas of finite field representation and the actual implementation results are explained. One may note that in truth table domain (considering the Boolean functions as a mapping from $\{0,1\}^n \rightarrow \{0,1\}$), the Maiorana-McFarland type bent functions can be seen as the concatenation of small affine functions. A theoretical view from this direction is being explored in an independent and parallel work [TKMM18] recently.

In a stream cipher, other than the Boolean function the main circuit component is LFSR or NFSR and the state size is decided by that. Thus, with a Boolean function with 500 GEs may be accommodated in a complete stream cipher circuit of 1000 GEs. Further, the Boolean function that we use have very good autocorrelation spectra, which may resist against proper signature generation in Differential Fault Attack. The complete design of stream cipher using such functions and the resistance against different attacks will be presented in the journal version of this paper.

References

[Car93] Claude Carlet. Two new classes of bent functions. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 77–101. Springer, 1993.

[CS09] Thomas W. Cusick and Pantelimon Stănică. *Cryptographic Boolean Functions and Applications.* Academic Press, 2009.

[Dil74] John F Dillon. *Elementary Hadamard difference sets.* PhD thesis, 1974.

[DIS09] Jean-Pierre Deschamps, Jose Luis Imana, and Gustavo D Sutter. *Hardware implementation of finite-field arithmetic.* McGraw-Hill New York, 2009.

[Dob94] Hans Dobbertin. Construction of bent functions and balanced Boolean functions with high nonlinearity. In *International Workshop on Fast Software Encryption*, pages 61–74. Springer, 1994.

[FF98] Eric Filiol and Caroline Fontaine. Highly nonlinear balanced Boolean functions with a good correlation-immunity. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 475–488. Springer, 1998.

[Fon99] Caroline Fontaine. On some cosets of the first-order reed-muller code with high minimum weight. *IEEE Transactions on Information Theory*, 45(4):1237–1243, 1999.

[IT88] Toshiya Itoh and Shigeo Tsujii. A fast algorithm for computing multiplicative inverses in gf (2m) using normal bases. *Information and computation*, 78(3):171–177, 1988.

[KCP17] Mustafa Khairallah, Anupam Chattopadhyay, and Thomas Peyrin. Looting the LUTs: FPGA Optimization of AES and AES-like Ciphers for Authenticated Encryption. In *International Conference in Cryptology in India*, pages 282–301. Springer, 2017.

[KMT18] Selçuk Kavut, Subhamoy Maitra, and Deng Tang. Searching balanced Boolean functions on even number of variables with excellent autocorrelation profile. In *Tenth International Workshop on Coding and Cryptography*, Saint-Petersburg, Russia, September 18-22, 2017.

[LN94] Rudolf Lidl and Harald Niederreiter. *Introduction to finite fields and their applications*. Cambridge university press, 1994.

[McF73] Robert L McFarland. A family of difference sets in non-cyclic groups. *Journal of Combinatorial Theory, Series A*, 15(1):1–10, 1973.

[MM16] Sihem Mesnager. *Bent functions*. Springer, 2016.

[PCZ17] Enes Pasalic, Anupam Chattopadhyay, and WeiGuo Zhang. Efficient implementation of generalized Maiorana–McFarland class of cryptographic functions. *Journal of Cryptographic Engineering*, 7(4):287–295, 2017.

[Rot76] Oscar S Rothaus. On "bent" functions. *Journal of Combinatorial Theory, Series A*, 20(3):300–305, 1976.

[SM08] Pantelimon Stănică and Subhamoy Maitra. Rotation symmetric Boolean functionscount and cryptographic properties. *Discrete Applied Mathematics*, 156(10):1567–1580, 2008.

[Spi80] R J Spillman. The effect of DON'T CARES on the complexity of combinational circuits. *Proceedings of the IEEE*, 68(8):1021–1022, 1980.

[TM18] Deng Tang and Subhamoy Maitra. Construction of n-variable (n$\equiv$ 2 mod 4) balanced Boolean functions with maximum absolute value in autocorrelation spectra $< 2^{n/2}$. *IEEE Transactions on Information Theory*, 64(1):393–402, 2018.

[TKMM18] Dang Tang, Selçuk Kavut, Bimal Mandal, Subhamoy Maitra. Modifying Maiorana-McFarland type bent functions for good cryptographic properties. Preprint, April 2018.

A Discrete-Log Representation of $\mathbb{F}_{2^n}$ Arithmetic

The Discrete-Log representation is described in Section 2.3. Multiplication can be defined as

$$x_1 \odot x_2 = \begin{cases} 2^n - 1 & \text{if } x_1 = 2^n - 1 \text{ or } x_2 = 2^n - 1 \\ x_1 + x_2 \pmod{2^n - 1} & \text{otherwise} \end{cases} \tag{8}$$

While inversion can be defined as

$$x^{-1} = \begin{cases} 2^n - 1 & \text{if } x = 2^n - 1 \\ -x \pmod{2^n - 1} & \text{otherwise} \end{cases} \tag{9}$$

Both operations require circuit complexity of $\mathcal{O}(n)$, which is smaller than the corresponding circuits for both normal and polynomial bases. While the same can be said about squaring, we show now that it can be implemented as a cyclic shift operation (similar to the case of normal basis). Squaring can be written in terms of multiplication as follows, where $\times$ is used for integer multiplications as opposed to finite field multiplication $\odot$,

$$x^2 = \begin{cases} 2^n - 1 & \text{if } x = 2^n - 1 \\ 2 \times x \pmod{2^n - 1} & \text{otherwise} \end{cases} \tag{10}$$

and

$$2 \times x \pmod{2^n - 1} = \begin{cases} x \ll 1 & \text{if } 2 \times x < 2^n - 1 \\ (x \ll 1) - (2^n - 1) & \text{otherwise} \end{cases} \tag{11}$$

Using the two's complement representation of integer arithmetic, Equation (11) can be written as

$$2 \times x \pmod{2^n - 1} = \begin{cases} x \ll 1 & \text{if } 2 \times x < 2^n - 1 \\ (x \ll 1) + 2^n + 1 \pmod{2^n} & \text{otherwise} \end{cases} \tag{12}$$

Equation (12) means that the squaring operation in the discrete-log representation is a left shift operation with the most significant bit of x becoming the least significant bit, i.e., a cyclic shift of x.

In addition, however, in the discrete-log representation is complicated. It can be implemented by using look-up tables or by conversion to another representation. Hence, studying the complexity of trace function is this representation without using addition is an interesting problem. Using property 2 of trace function in Section 2.1 and and Equation (12), we can conclude, as in the case of normal basis, that trace function is a Rotation Symmetric Boolean Function (RSBF). Now we define the rotation symmetric Boolean functions. Let $x_i \in \mathbb{F}_2$ for $0 \le i \le n-1$. We define

$$\rho_n^r(x_i) = x_{(i+r) \bmod n} = \begin{cases} x_{i+r}, & \text{if } i + r \le n - 1; \\ x_{i+r-n}, & \text{if } i + r \ge n. \end{cases}$$

Let $P_n = \{\rho_n^0, \rho_n^1, \ldots, \rho_n^{n-1}\}$ be the permutation group which contains the rotations of n symbols, defined as

$$\rho_n^i(x) = \rho_n^i(x_{n-1}, x_{n-2}, \ldots, x_0) = (x_{(n-1+i) \bmod n}, x_{(n-2+i) \bmod n}, \ldots, x_{(i) \bmod n}).$$

Definition 1. *A Boolean function f in n variables is said to be rotation symmetric if and only if for any $x \in \mathbb{F}_2^n$, $f(\rho_n^i(x)) = f(x)$, for all $0 \le i \le n-1$.*

The problem of defining an RSBF is related to the problem of necklace equivalence in combinatorics. This helps to derive an upper bound on the circuit complexity of a trace function in the discrete-log representation.

Definition 2. *A binary necklace of length n is an equivalence class of n-character strings over the alphabet $\{0, 1\}$, where two arrangements are equivalent if one can be obtained from the other by applying cyclic rotations.*

Definition 3. *The lexicographical representation of a binary necklace N is the member of $[N]$ with the maximum number of leading 0's.*

B Circuit for the Tang-Maitra functions based on discrete-log representation

The circuit in Figure 3 can be used to compute the Tang-Maitra function when the inputs are in the discrete-log representation. The operation $\frac{x}{y}$ is computed as $x - y \pmod{2^k - 1}$, with complexity $\mathcal{O}(k)$. After that, Tr_1^k is computed as an RSBF. In this Section, we give a circuit for any RSBF, with sub-exponential complexity $\mathcal{O}(k^2 + 2^k/k^2)$.

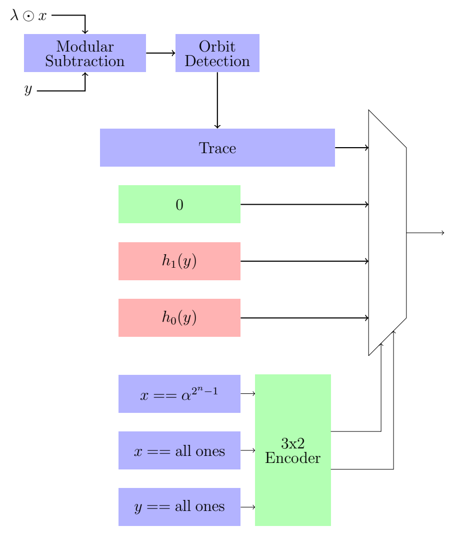

Fig. 3: Discrete-log circuit for a Tang-Maitra function

Rotation symmetric Boolean function Circuits Let f be a rotation symmetric Boolean function in k variables, i.e., $f(\rho_k^i(x)) = f(x))$, for all $0 \leq i \leq k-1$. Hence, $[x]$ is an equivalence class (orbit) that includes all the rotations of x, i.e., $[x] = \{\rho_k^i(x) | 0 \leq i \leq k - 1\}$. We choose the representative of that class to be

$\rho_k^r(x)$, such that $\rho_k^r(x) \geq \rho_k^i(x)$, for all $0 \leq i \leq k-1$. In other words, it is the rotation of x that has the maximum integer value. For more details of rotation symmetric Boolean function we refer to [FF98,Fon99]. This is the lexicographical representation of $[x]$ based on the alphabet $\{0,1\}$.

Lemma 3. *A rotation symmetric Boolean function (RSBF) of k variables has a circuit complexity bounded by $\mathcal{O}(k^2 + 2^k/k^2)$.*

Lemma 4. *The discrete-log implementation of the Tang-Maitra function of n variables, where n is even, has a circuit complexity bounded by $\mathcal{O}(2^k+k^2+2^k/k^2)$, where $n = 2k$.*

Proof. In order to convert any x to its lexicographical orbit representation, the orbit detection circuit generates all the k rotations of x, then chooses the value of x that has the maximum integer value using a selection tree that consists of $k-1$ two-input MAX circuits. Every two-input MAX circuit consists of $k+1$ integer subtractor ($6k+6$ gates) and k 2×1 MUXes, $3K$ gates. Hence, the orbit detection circuit has a complexity of around $9k^2 - 3k - 6$ gates. After the lexicographical orbit representation has been detected, a circuit decides whether the given orbit functional value is 0 or 1. This circuit expects only 1 of the lexicographical representations, which, according to Burnside's Lemma and [SM08, Theorem 3], are $N_O = \frac{1}{k}\sum_{d|k}\phi(d)2^{\frac{k}{d}}$, where ϕ is Euler's phi-function. Hence, $n_x = 2^k - N_O$ values in the Truth table of such circuit can be set as DON'T CARES 'X'. In [Spi80], the author gave an analysis of the circuit complexity of combinational circuits with a large number of DON'T CARES. The number of AND/OR/NOT gates was given by

$$L_\infty = (1-d)H(p)L_\infty(G),$$

where $d = \frac{n_x}{2^k}$, $p = \frac{n_1}{(1-d)2^k}$, $H(p) = -p\log(p) - (1-p)\log(1-p)$ and $L_\infty(G) = \frac{2^k}{k}$. By substitution for the case of the trace circuit, the number of gates is $\frac{N_O}{n}H(p)$, where $H(p) \leq 1$. Hence, the circuit complexity is $\mathcal{O}(\frac{N_O}{n})$, and from Burnside's Lemma, it can be expressed as $\mathcal{O}(\frac{2^k}{k})$. Hence, the overall complexity of this construction is $\mathcal{O}(k^2 + 2^k/k^2)$. □