IACR Transactions
ISSN XXXX-XXXX, Vol. 0, No. 0, pp. 1–31. DOI:XXXXXXXX

Compact Circuits for Efficient Möbius Transform

Subhadeep Banik[1] and Francesco Regazzoni[1,2]

[1]Universita della Svizzera Italiana, Lugano, Switzerland. subhadeep.banik@usi.ch
[2]University of Amsterdam, Amsterdam, Netherlands. f.regazzoni@uva.nl

Abstract. Möbius transform is a linear circuit used to compute the evaluations of a Boolean function over all points on its input domain. The operation is very useful in finding the solution of a system of polynomial equations over $GF(2)$ for obvious reasons. However the operation, although linear, needs exponential number of logic operations (around $n \cdot 2^{n-1}$ bit xors) for an n-variable Boolean function. As such the only known hardware circuit to efficiently compute the Möbius Transform requires silicon area that is exponential in n. For Boolean functions whose algebraic degree is bound by some parameter d, recursive definitions of the Möbius Transform exist that requires only $O(n^{d+1})$ space in software. However converting the mathematical definition of this space-efficient algorithm into a hardware architecture is a non-trivial task, primarily because the recursion calls notionally lead to a depth-first search in a transition graph that requires context switches at each recursion call for which straightforward mapping in hardware is difficult. In this paper we look to overcome these very challenges in an engineering sense. We propose a space efficient sequential hardware circuit for the Möbius Transform that requires only polynomial circuit area (i.e. $O(n^{d+1})$) provided the algebraic degree of the Boolean function is limited to d. We show how this circuit can be used as a component to efficiently solve polynomial equations of degree at most d by using fast exhaustive search. We propose three different circuit architectures for this, each of which used the Möbius Transform circuit as a core component. We show that asymptotically, all the solutions of a system of m polynomials in n unknowns and algebraic degree d over $GF(2)$ can be found using a circuit of silicon area proportional to $m \cdot n^{d+1}$ and physical time proportional to $2 \cdot \log_2(n-d) \cdot 2^{n-d}$.

Keywords: Boolean Functions, Möbius transform, Solution of Equation System.

1 Introduction

Several cryptanalytic problems can be reduced to instances of solving a system of multivariate polynomial equations over $GF(2)$. For example, block ciphers with low multiplicative complexity like LowMC [ARS+15] employ only 3-bit S-boxes of algebraic degree 2. It is known that given any single plaintext-ciphertext pair from an r-round instance of LowMC gives rise to a system of equations in the secret key-bits of algebraic degree $2^{r/2}$ [Din21]. It is also known that forging a signature in public key signature schemes like UOV can be done by solving a set of quadratic equations over $GF(2)$ [KPG99]. Other than this there are specific problems in combinatorics like the graph-coloring problem (i.e. given a graph decide whether it can be colored using k colors with no two adjacent vertices assigned the same color) which can be reduced to an instance of solving multi-variate polynomials in $GF(2)$ [Bar09, Appendix C].

The problem can be stated in the following way: given n indeterminates $x_1, x_2, \ldots, x_n$, and m polynomials $f_i \in \mathbb{F}[x_1, x_2, \ldots, x_n]$ (for $i \in [1, m]$), where $\mathbb{F}$ is any finite field. The task is to find common solutions $x^* \in \{0,1\}^n$, such that $f_i(x^*) = 0$ for all i. Over any finite field $\mathbb{F}$, the problem is NP-complete already when the polynomials are quadratic.

This is why the problem is extremely important in cryptography. Hereafter, we will focus on the case of the Boolean field $\mathbb{F} = GF(2)$.

1.1 Previous Work

To the best of our knowledge, there have been two previous works on hardware/software architectures for fast exhaustive search over $GF(2)$. The main idea is as follows: the secret x^*, we are looking for is obviously a point which evaluates to zero for all the f_i. Thus at the index x^*, the truth tables of all the Boolean polynomials f_i will contain the constant 0. Hence, we are looking for the indices x^* at which the logical **OR** of all the truth tables of all the f_i's is 0. In [BCC+10], the authors use the Gray code technique to evaluate the truth table of each polynomial f_i. Gray codes are linear codes which have the property that successive codewords differ by only one bit. There are many methods of constructing such codes in literature, and one of the simplest way is to define the i-th code word as $g_i = i \oplus (i \gg 1)$ (the $\gg$ denotes the rotate right operator). For example the eight 3-bit codewords listed sequentially are: 000, 001, 011, 010, 110, 111, 101, 100. Take any polynomial f_i: we want to evaluate f_i over all 2^n points of its input domain. Then it is more efficient to do this evaluation in the order specified by the Gray code, i.e. first $f_i(g_0)$, then $f_i(g_1), f_i(g_2) \ldots$ etc. The reason for this is as follows: note $f_i(g_0) = f_i(\vec{0})$ is just the constant term of f_i, thereafter if t is the only bit-position where the successive codewords g_j and g_{j+1} differ in, and we already have the value of $f_i(g_j)$ then we can use a Taylor-like expansion formula for Boolean functions to compute $f_i(g_{j+1})$:

$$f_i(g_{j+1}) = f_i(g_j) \oplus \frac{\delta f_i}{\delta x_t}(g_j). \tag{1}$$

Here $\frac{\delta f_i}{\delta x_t}$ is the 1st order derivative of the function f_i at the point x_t. For example if $f_i = x_1x_2 \oplus x_3 \oplus x_1x_4x_5$, then $\frac{\delta f_i}{\delta x_1} = x_2 \oplus x_4x_5$ and $\frac{\delta f_i}{\delta x_2} = x_1$, $\frac{\delta f_i}{\delta x_3} = 1$ etc. It is known that the derivative has algebraic degree at least one less than the original function, and so if the derivative is not a constant or a degree one function we recursively evaluate the derivative term in Equation (1) with another round of Taylor expansion. The method obviously works best if the function f_i is quadratic, but [BCC+10] showed that it works for moderately higher degree functions too if some of the derivatives are precomputed and stored in memory. In a follow up work [BCC+13], the same authors proposed a hardware circuit for the problem, however, for only degree 2 functions (that did not need pre-computations). The problem with this approach is the initial pre-computation of derivatives one needs to do that costs a significant computations and thus energy when translated into hardware. Furthermore the pre-computations have to be done for each polynomial f_i, which implies that derivative computations for one set of polynomials can not be reused for another set and so a reduction of complexity via amortization over several polynomial systems is also not possible.

Another method to compute the truth table of a Boolean polynomial from its algebraic expression is via the Möbius Transform. This method does not require pre-computations. The transform can be simply evaluated as $\vec{v} = M_n\vec{u}$, where $\vec{u}$ is the $2^n \times 1$ algebraic normal form (ANF) vector of any n-variable Boolean function, M_n is the $2^n \times 2^n$ binary Möbius matrix, and $\vec{v}$ is the truth-table of the function, with its i-th element being the function evaluation at the binary string representation of i. As we will soon see, a naive interpretation of this method requires time and space exponential in n to compute. However there exist more subtle methods to compute the matrix-vector product given above in polynomial space (bounded by n^{d+1} where d is the algebraic degree of the Boolean polynomial). Translating this to hardware is a non-trivial task as the underlying algorithm is significantly complex. In this paper, we will propose strategies to translate the Möbius Transform algorithm into a hardware circuit and we will demonstrate how to overcome

the engineering challenges involved. We then show how multiple instances of the above Möbius Transform circuit can be efficiently utilized to solve or perform fast exhaustive search for roots of equation systems over $GF(2)$ whose degree is bound by some constant d. We show that asymptotically, with silicon footprint proportional to $m \cdot n^{d+1}$ we can describe a circuit that finds roots of a system of m polynomial equations of degree d in n unknowns over $GF(2)$.

1.1.1 Comparison with Linearization Algorithms

Linearization based algorithms like XL [CKPS00] and Elimlin [CB07] also attempt to find the solution of a system of Boolean equations through matrix manipulation techniques like Gaussian Elimination (GE). The idea is to rewrite every higher degree monomial in the equation system as a new linear variable. This converts a system of m equations of any arbitrary algebraic degree d to a system of m linear equations in around $O(n^d)$ extended variables. Using hardware accelerators for GE like the SMITH framework [BMP+06], one could also describe a circuit that finds roots of the system using silicon area proportional to $m \cdot n^d$. However as shown in [Bar09, Sec 12.3], such an approach will generate basis vectors for a space containing an exponential number of false solutions, and it is not immediately clear how efficient circuit architectures can be described to eliminate them.

1.2 Contribution and Organization

In this paper we present a novel hardware architecture for Möbius transform for n-variable Boolean functions of degree $\leq d$ that requires silicon resources that are polynomially bounded by n^{d+1}. We use the recursive definition of the transform found in [Din21, Sec 4.2], and identify and solve the engineering difficulties of translating such an algorithm into hardware. Parallel instances of this architecture can be combined to construct hardware solvers that find roots of an underlying equation system over $GF(2)$ by exhaustive search. We describe the architectures of three such solvers the last of which is able to find all roots of any system of m Boolean equations in n unknowns and algebraic degree d in circuit area proportional to $m \cdot n^{d+1}$ and physical time proportional to $2 \cdot \log_2(n-d) \cdot 2^{n-d}$ units.

The rest of the paper is organized in the following manner. Section 2 presents some preliminary lemmas and definitions in this field. In Section 3 we look at the recursive definition of Möbius transform and we explain in detail how the hardware circuit for the same is designed. In Section 4, we first show how to combine multiple instances of the Möbius Transform circuit that produces a solver that finds at least one root of the underlying equation system. We then list two variants of this architecture, the last of which is able to find all the roots of the equation system. Section 5 concludes the paper.

2 Definitions and Preliminaries

Boolean function: An n-variable Boolean function is a map from $\{0,1\}^n \rightarrow \{0,1\}$ and it can be uniquely represented by its algebraic expression, called algebraic normal form or ANF. The algebraic expression of such a function using the $(\oplus, \cdot)$ basis can be written as

$$f(\vec{x}) = f(x_0, x_1, \ldots, x_{n-1}) = \bigoplus_{i \in \{0,1\}^n} a_i x^i$$

Here $i := i_0 i_1 \cdots i_{n-1}$ is the binary string of length n, with i_j as the individual bits and x^i is defined as $\prod x_j^{i_j}$. The ANF vector $\vec{u} = [a_0, a_1, \ldots, a_{2^n-1}]$ is defined as the 2^n-length string of all the a_i's. For example, consider the 3-variable function $f = 1 \oplus x_0 x_1 \oplus x_2 \oplus x_0 x_2$. We can write this as $x_0^0 x_1^0 x_2^0 \oplus x_0^1 x_1^1 x_2^0 \oplus x_0^0 x_1^0 x_2^1 \oplus x_0^1 x_1^0 x_2^1$. The function can be expressed as a length 8 bit-vector $\vec{u}$ with bits at locations given by the binary strings 000, 110,

$$M_3 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Figure 1: An example of the Möbius matrix M for $n = 3$

001 and 101 i.e. 0, 6, 1 and 5 set to 1 and the rest of the bits 0, which is to say that $a_0 = a_1 = a_5 = a_6 = 1$ and the rest of the $a_i = 0$.

The algebraic degree of the function (provided the function is not identically null) is defined as the maximum hamming weight of the string i such that $a_i = 1$. Thus in the previous example, the algebraic degree is 2. For functions having degree d, all the coefficients a_i such that $hw(i) > d$ are naturally 0. Since there are exactly $\binom{n}{i}$ length n strings of hamming weight i, we can see that the ANF of degree d function can be expressed using $\binom{n}{\downarrow d} := \sum_{i=0}^{d} \binom{n}{i} < n^d$ binary coefficients.

Truth Table: The vector of evaluations of a Boolean function at all its input points is called its **Truth Table** (therefore this is a 2^n length vector). The ANF and the Truth table vectors of any Boolean function are closely related by the Möbius transform. Let $\vec{v} = [v_0, v_1, \ldots, v_{2^n-1}]$ be the truth-table of the function f, with its i-th element being the function evaluation at the binary string representation of i, i.e. $v_i = f(i_0, i_1, \ldots, i_{n-1})$. then it is well known that $\vec{v}$, $\vec{u}$ are related as $\vec{v} = M_n \cdot \vec{u}$, where M_n is the Möbius matrix of size $2^n \times 2^n$. The i, j-th element of this matrix m_{ij} is given as

$$m_{ij} = 1 \text{ if } j \preceq i \text{ and } 0 \text{ otherwise.}$$

The operator $\preceq$ is a partial order over all binary strings: we say that $j \preceq i$ if the binary string representing j is less than or equal to the binary string representing i in all indices. For example, $4 \preceq 5$, since 100 is less than 101 at all bit-locations, but $3 \not\preceq 4$ since 011 exceeds 100 in the last 2 bit-locations.

The Möbius matrix M_n has been widely studied in literature: for example it is well known that is lower-triangular and involutive i.e. $M_n^{-1} = M_n$. Thus both $\vec{v} = M_n \cdot \vec{u}$ and $\vec{u} = M_n \cdot \vec{v}$ hold. An example of the 8×8 Möbius matrix M_3, i.e. for $n = 3$ is shown in Figure 1. This helps us see an alternative recursive definition of M_n. If we define $M_1 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$, then for all $n > 1$, we have $M_n = M_1 \otimes M_{n-1}$, where $\otimes$ is the matrix tensor product.

Multiplication of a vector by this matrix can be quickly executed by the butterfly-like operations shown in Figure 2. The butterfly operation shaded in blue is actually multiplication of the input 2-bit vector by the matrix M_1. The figure tells us that for an n-variable function, the algorithm can be done in-place (without any additional memory) using around $n \cdot 2^{n-1}$ xor operations and 2^n space.

3 Implementing the Möbius Transform

Given Figure 2, we can think of many strategies to implement the basic transform if one has access to exponential silicon resources. The operation consists of n stages of sequential xor layers, with each layer having exactly 2^{n-1} xor operations over bits. Given this, one can think of several circuit strategies to implement this:

Expmob1 This architecture implements the circuit in Fig 2 as a single unrolled circuit, i.e. it implements all the n butterfly stages as dedicated circuits sequentially. Consider $\textbf{one}_i(x) : \{0,1\}^{n-1} \rightarrow \{0,1\}^n$ to be the function that inserts a 1 in the i-th MSB position of x, and $\textbf{zero}_i(x)$ to be a function that inserts a 0 in the same position, i.e $\textbf{one}_0(1001) = 1\ 1101$ and $\textbf{zero}_0(1001) = 0\ 1001$ etc. Note that there are a total of 2^{n-1} butterfly operations in each of the n stages. In the i-th stage (for $0 \leq i \leq n-1$), the j-th butterfly takes as input the bits in the position $\textbf{zero}_i(j)$ and $\textbf{one}_i(j)$ for all $0 \leq j \leq 2^{n-1} - 1$. This requires a total of $n \cdot 2^{n-1}$ number of 2-input xor gates in total. However such a circuit is able to compute the transformation in a single cycle.

Expmob2 This configuration is slightly different from the previous circuit, in the sense that we have only a single stage butterfly which we operate over n clock cycles to compute the transform, i.e. similar to round based circuits of block ciphers in which a single round function circuit is iterated over a given number of cycles to compute the transform. Unlike the round function of a block cipher the successive stages of xor layers are not exactly similar. For example, consider the topmost butterfly circuit in each stage in Fig 2. The 1st stage takes bits at positions 0 and 4 as input, the second stage takes bits 0 and 2, the third stage takes bits 0 and 1 and so on. So to create a round based circuit, it would seem that one would need multiple n to 1 multiplexers before each of the butterfly circuits. However this can be avoided using a simple observation. Consider π_n to be the following permutation:

$$\pi_n(2x) = x, \quad \text{and} \quad \pi_n(2x+1) = 2^{n-1} + x \quad \text{for all} \quad 0 \leq x < 2^{n-1}$$

The idea is that after the given stage of butterfly circuits, the bit at position i be shifted to position $\pi_n(i)$. Such a permutation over the bits requires only re-routing of wires and thus no additional silicon area. This is essentially the entire round function circuit which has to be executed for a total of n cycles for the transform to be computed. To see why this works, consider the following facts. Let B_n be the block diagonal matrix defined as $B_n = M_1 \otimes I_{n-1}$, where I_{n-1} is the identity matrix of size $2^{n-1} \times 2^{n-1}$. Note that B_n is transformation defined by the first stage of butterfly layer in Fig 2. Let P_n be the permutation matrix corresponding to π_n. Then it is easy to verify that the Möbius matrix $M_n = (P_n \cdot B_n)^n$.

3.1 Synthesis Results

In this section we will describe the flow of simulation followed for each of the circuits reported in the paper. The design was described at the RTL level using a hardware description language and functional correctness was first verified. Thereafter the circuit

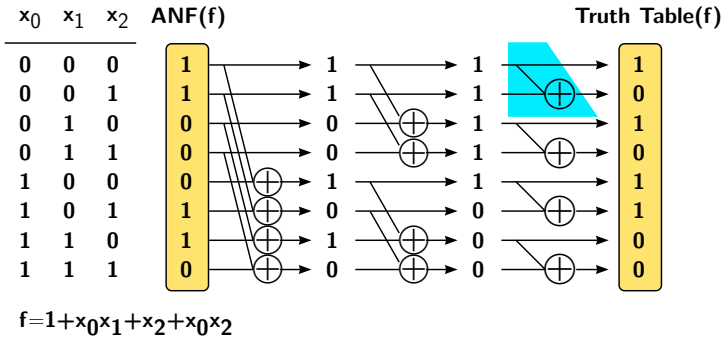
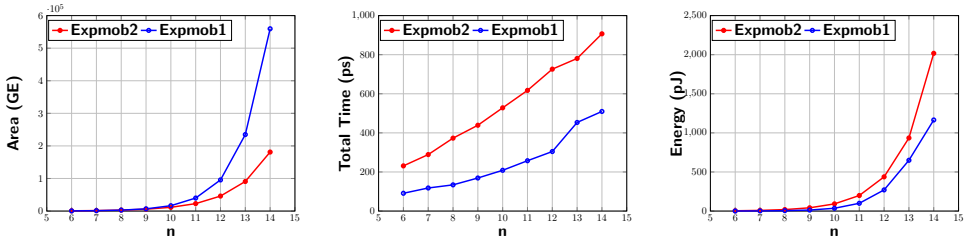


Figure 2: Möbius transform on $f = 1 \oplus x_0x_1 \oplus x_2 \oplus x_0x_2$. The blue shaded component represents one butterfly unit.



(a) Area (b) Time (c) Energy

Figure 3: Synthesis results for **Expmob1** and **Expmob2** circuits

was synthesized using the Nangate 15nm Open Cell Library [MMR+15], mainly to ensure that the results obtained can be reproduced readily. One of the utilities of the Möbius Transform, is in solving equation systems. In order to ensure that equation systems are solved as quickly as possible, the circuit compiler was instructed to specifically optimize the total critical path of the circuit. A timing analysis is then performed on the synthesized netlist using sufficient number of randomly generated test vectors, which outputs the switching statistic of every node in the circuit. This information is used by a power compiler software to estimate the average power consumed by the circuit. Energy is computed as the product of the average power and the total physical time taken for the circuit to execute a given operation.

In Figure 3, we present comparative synthesis results for the circuits **Expmob1** and **Expmob2**. It can be seen that **Expmob1** performs better than **Expmob2** in this regard, most probably due to the fact that additional hold/setup time constraints need to be met for **Expmob2** for writing on to the register in each cycle. Similarly the additional energy required for the n successive register writes makes **Expmob2** less energy efficient as compared to **Expmob1** as shown in Figure 3c[1]. For a detailed tabulation of the synthesis results, please see Table 2 in Appendix B.

However both these circuits require exponential amount of logic gates which starts to become a bottleneck as n increases. We have already seen that a degree d Boolean function can be represented with only $\binom{n}{\downarrow d} < n^d$ binary coefficients, which means that for small values of d the size of the ANF vector is polynomially bounded. Thus the size of the register that holds the ANF can be bounded by n^d. However, it is not possible to use **Expmob1**/**Expmob2** circuit to compute the transform on this reduced size register, since, although the initial ANF vector is small, the output of each layer of butterflies are progressively larger till it reaches 2^n (which is the expected size of the truth table) after the last stage.

3.2 Recursive Algorithm for Möbius transform

There exists algorithms that perform the basic transform (on functions limited to degree d) using polynomial space only, i.e. bounded by n^{d+1}. We state the algorithm appearing in [Din21, Sec 4.3]. The algorithm requires a notional depth first traversal in a transition graph as shall be explained shortly.

The principal question is how do we circumvent the fact that even if we begin with a ANF vector of size that is polynomially bounded, each butterfly stage is likely to produce an output that is of size larger than the input. First let us make the following observation

[1]Although the power/energy figures were computed at a clock frequency of 10MHz, it is well known [KDH+12, BBR15] that energy consumption is independent of clock frequency for medium to low leakage environments

taking Figure 2 as a reference: consider the initial ANF vector $A_0 = [1100\ 0110]$ and the vector $A_1 = [1100\ 1010]$ just after the first layer. The initial vector corresponds to the function

$$\begin{aligned} f(x_0, x_1, x_2) &= 1 \oplus x_0x_1 \oplus x_2 \oplus x_0x_2 = x_0 \cdot (x_1 \oplus x_2) \oplus (1 \oplus x_2) \\ &= x_0 \cdot \big[f(1, x_1, x_2) \oplus f(0, x_1, x_2)\big] \oplus f(0, x_1, x_2) \end{aligned}$$

Note that $\big[f(1, x_1, x_2) \oplus f(0, x_1, x_2)\big] := \frac{\delta f}{\delta x_0}$ is simply the derivative of f at the coordinate x_0. Both $\frac{\delta f}{\delta x_0}$ and $f(0, x_1, x_2)$ have number of variables which is 1 less than the original function, and it is obvious that both their algebraic degrees can not be more than that of the original function.

Now consider the vectors in the top and bottom halves of A_1 i.e. $A_{\mathsf{top}} = [1100]$ and $A_{\mathsf{bottom}} = [1010]$. It is easy to observe/verify the following:

A: A_{top} is the ANF vector for $f(0, x_1, x_2)$ (in this case $1 \oplus x_2$) and A_{bottom} is the ANF vector for $f(1, x_1, x_2)$ (in this case $1 \oplus x_1$).

B: Both $A_{\mathsf{top}}/A_{\mathsf{bottom}}$ are outputs of the butterfly layer in which the input is A_0. Whereas A_{top} is the arm of the butterfly that does not require xor computations, some xor computations are required for A_{bottom}.

C: The remaining steps from the 2nd stage onwards can be seen as the parallel application of the Möbius Transform on the reduced variable Boolean functions $f(0, x_1, x_2)$ and $f(1, x_1, x_2)$

Of course in the figure, both the transforms are computed parallelly, which requires 2^{n-1} space each and so the total space requirement is 2^n which is the same as the original. The idea behind the recursive transform is to do these 2 sub-transforms sequentially, i.e. one after the other so that the same space (i.e. register locations) can be used for both the transforms so that the cumulative space requirement does not add up. Let us state the algorithm now formally (Algorithm 1). The algorithm is parameterized by two quantities: number of variables n, and the maximum algebraic degree d that the underlying function can have.

Algorithm 1: Recursive Möbius Transform

Möbius (A_0, n, d)
Input: A_0: The compressed ANF vector of a Boolean function f
Input: n: Number of variables, d: Algebraic degree
Output: The Truth table of f

```
/* Final recursion step, i.e.  leaf nodes of recursion tree */
if n=d then
    Use the formula B = M_n · A_0 to output partial truth table B.
    /* Use either Expmob1/Expmob2 to do this */
end
else
    Declare an array T of size (n-1 choose ↓d) bits.
    /* Now we compute the 2 operations of the butterfly layer */
1   Store 1st butterfly output i.e. A_top in T (requires no xors).
    Call Möbius (T, n-1, d)
2   Store 2nd butterfly output i.e. A_bottom in T (requires some xors).
    Call Möbius (T, n-1, d)
end
```

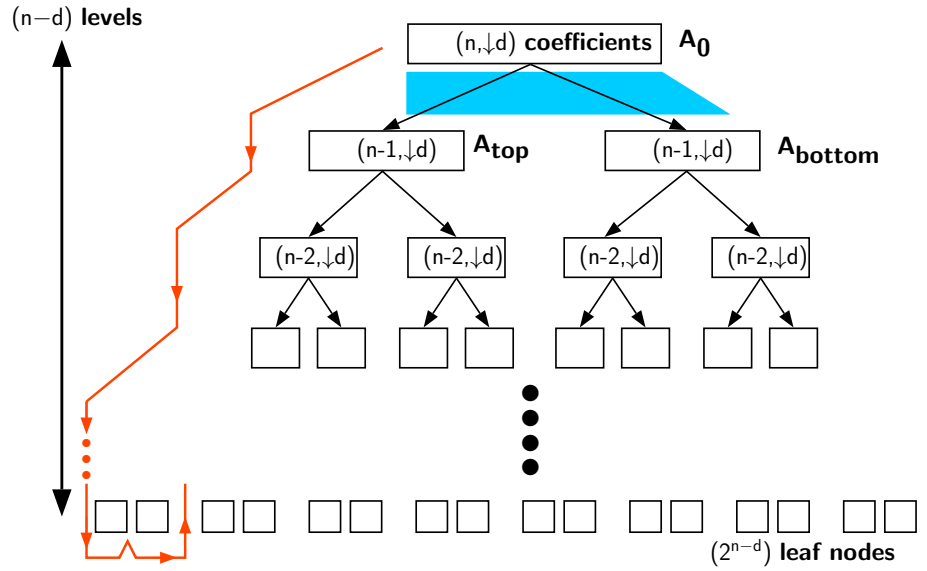


Figure 4: Recursion tree for the Möbius Transform algorithm. The blue shaded component roughly represents one arm of the butterfly unit. Note here $(x, \downarrow d) := \binom{x}{\downarrow d}$.

For the sake of simplicity, we have excluded many operational details in the above algorithm to give the reader a better idea of the flow of the algorithm. The space requirement of this algorithm is easy to estimate from the algorithm description. We start out with $\binom{n}{\downarrow d}$ coefficients required to store A_0. Thereafter every successive i-th recursion stage requires $\binom{n-i}{\downarrow d}$ additional memory for all $1 \leq i \leq n-d$. The final stage can use **Expmob1**/**Expmob2** to perform Möbius Transform in-place and no additional memory is required. However in our experiments we preferred to use **Expmob1** because it is slightly faster. Hence the total space requirement of this procedure is given by (for a proof of the following please see Appendix C):

$$S(n,d) = \sum_{i=0}^{n-d} \binom{n-i}{\downarrow d} \in O(n^{d+1}). \tag{2}$$

Notionally speaking the algorithm listed above describes a depth first recursion tree as shown in Figure 4, where each node in tree are connected to its two butterfly outputs. The depth first nature of the structure gives rise to complications even while implementing it in software. The problem with implementing such a routine, even in software, is the high number of context switches, that is needed to traverse one level down. In layman's terms, before we can do a downward dive in the tree, the current state information, variables etc has to be stored in a separate memory location (usually denoted as "call-stack"). This costs time/energy and makes the algorithm less attractive from a practical point of view.

3.3 Hardware circuit Polymob1

The goal obviously is to construct a circuit that does not take more than a total of $S(n,d)$ bits of register space. As such we are looking at a circuit architecture similar to the one shown in Figure 5.

To understand the challenges in this circuit, note that one needs to follow the flow given by the orange line in the recursion tree in Figure 4. Now there is one top-level register of size $\binom{n}{\downarrow d}$ storing the initial ANF vector A_0. There is only one $\binom{n-1}{\downarrow d}$ size register to store

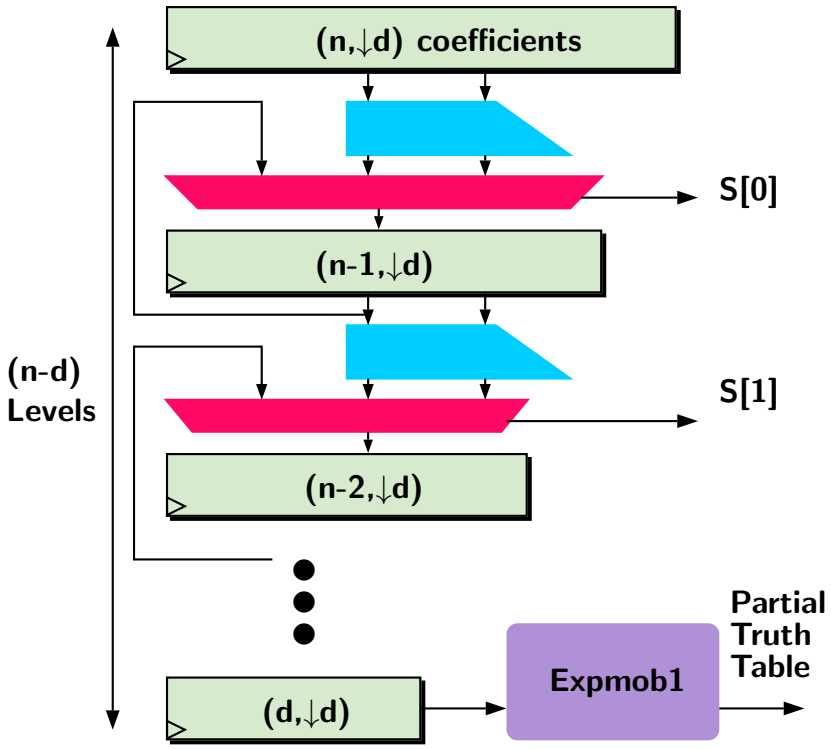


Figure 5: Hardware architecture **Polymob1** for the Möbius Transform algorithm. The blue shaded part roughly represents one arm of the butterfly unit. Note here $(x, \downarrow d) := \binom{x}{\downarrow d}$.

the second level coefficients A_{top} and A_{bottom}. This implies that if in the first clock cycle the 2nd register stores A_{top}, it must preserve this state till the entire left sub-tree rooted at this node is executed before it overwrites its state to A_{bottom}. Similarly there is only one $\binom{n-2}{\downarrow d}$ register to store potentially four ANF vectors (two each from the butterfly operation on A_{top} and A_{bottom}). Thus the engineering challenge is to ensure that each register at the successive levels store and preserve appropriate state vectors till it is time to overwrite them, and so this in a manner that minimizes the total number of clock cycles required to execute the Möbius Transform.

Thus we arrive at the architecture in Figure 5. Each i-th level has a single register of size $\binom{n-i}{\downarrow d}$ (for $0 \leq i \leq n-d$), and from $i = 1$ onwards each register is preceded by a 3:1 multiplexer of size $\binom{n-i}{\downarrow d}$. This is because each register must be able to accept 3 different inputs:

1. Its own output state, or in other words it must be able to preserve its state.
2. Either of the 2 outputs of the butterfly stage preceding it.

3.3.1 Architectural Details:

We begin by noting that a $3:1$ multiplexer is not necessary for the above architecture unless we add other functionalities to the circuit like the one described in Section 4.3. For executing the basic Möbius Transform a $2:1$ multiplexer will also serve the purpose. We explain this with the first couple of registers but the same principle holds for the registers in the lower levels too. Note that the second register in its lifetime can only store two vector values A_{top} and A_{bottom} depending on how far the execution has reached in the

process of the traversal of the recursion tree. Both of these are obtained from the butterfly operation on A_0 which resides on the register at the level just above. Thus the idea is to have a single 2:1 multiplexer separating the two registers, which takes as input the two outputs of the butterfly operation. When the 2nd level register would need to preserve state (A_{top} or A_{bottom}), it can be done by appropriately setting the select signal of the multiplexer: for example to preserve A_{bottom} we just need to set the select signal of the preceding mux so that it accepts the A_{bottom} signal from the previous butterfly stage.

It remains to be seen how one can effectively set the multiplexer signals. In order to do that let us try to observe a small example. We will make use of a more general notation for the successive ANF vectors instead of just $A_{\mathsf{top}}/A_{\mathsf{bottom}}$, since we have to accommodate ANFs at different levels. We use the notation $A[\ell]_b$ to denote the ANF vector at some level of the recursion tree: the ℓ term in the square braces denotes the level of the ANF vector in the recursion tree, and the term b which can be seen as a binary string or integer contains information about the coordinates over which the derivatives have been computed to obtain the function. For example, take the case when $n = 5,\ d = 2$. The ANF of original function $f(x_0, x_1, x_2, x_3, x_4)$, we denote by the notation $A[0]_{000}$: note that the subscript is a binary string of length $n - d$ (which is 3 in this example). This is because there are $n - d$ levels in the recursion tree, each obtained by taking derivative over some co-ordinate variable. The level 1 ANFs corresponding to the functions $f(0, x_1, x_2, x_3, x_4)$ and $\frac{\delta f}{\delta x_0} = f(0, x_1, x_2, x_3, x_4) \oplus f(1, x_1, x_2, x_3, x_4)$ are denoted by $A[1]_{000}$ and $A[1]_{100}$ respectively (thus A_{top} and A_{bottom} defined earlier are equal to $A[1]_{000}$ and $A[1]_{100}$ respectively in this new notation). Similarly the two level 2 functions obtained by applying the butterfly layer on $A[1]_{000}$ (by taking derivative over x_1) are denoted as $A[2]_{000}$ and $A[2]_{010}$. Similarly butterfly over $A[1]_{100}$ yields the two vectors $A[2]_{100}$ and $A[2]_{110}$.

Generalizing this: if $A[\ell]_b$ is the ANF vector at some level ℓ of the tree, then after applying the butterfly over the coordinate x_ℓ, the two output vectors are denoted as $A[\ell + 1]_b$ and $A[\ell + 1]_{b \oplus e_\ell}$, where e_t is the unit vector of length $n - d$ with 1 at the t-th position, eg. $e_0 = 100 \ldots 0, e_1 = 010 \ldots 0$ etc. At this moment, let us turn towards the example in Figure 6, where we have manipulated the select signals of each multiplexer so that the entire Möbius Transform is computed in $2^{n-d} = 8$ cycles, i.e. in each of the 8 cycles we get one partial truth table of size $\binom{d}{\downarrow d} = 2^d = 4$. Initially the top-level register would be initialized with $A[0]_{000}$ and the remaining registers would stay uninitialized. In the 2 cycles following this, the select signals of each multiplexer, is set to zero so that, after this each level ℓ register contains $A[\ell]_{000}$. Figure 6 shows us the flow of data in each of the 8 cycles succeeding this.

We introduce an additional notation: let $S[\ell]_t$ be the select signal of the multiplexer between the registers at levels ℓ and $\ell + 1$ at time t. Which is to say that if $S[\ell]_t = 0$ and the ANF vector at level ℓ at time t is $A[\ell]_b$, then at time $t + 1$, the ANF vector at level $\ell + 1$ is $A[\ell + 1]_b$, and if $S[\ell]_t = 1$ then the corresponding vector is $A[\ell + 1]_{b \oplus e_\ell}$ (this can also be written as $A[\ell + 1]_{b+e_\ell}$ since by design the coordinates of b at positions larger than ℓ are all 0). The two expressions can obviously be combined to give the single compact expression $A[\ell + 1]_{b+S[\ell]_t \cdot e_\ell}$ that caters for both values of $S[\ell]_t$. In Figure 6, we have done a series of assignments to the variables $S[\ell]_t$ (for $0 \leq \ell < n - d$ and $0 \leq t < 2^{n-d} - 1$) so that the vector at the bottommost level of the register chain is always $A[n - d]_t$ for all $0 \leq t < 2^{n-d}$. Since **Expmob1** is connected to the bottommost register, this ensures that the all the partial truth tables are faithfully computed and the circuit indeed computes the Möbius Transform of any five variable Boolean function of degree upto 2. However we are more interested in engineering the multiplexer signals for general values of n, d. To do so, equivalently consider the subscripts of the ANF vectors as integers, and return to the example in Figure 6. Initially all the subscripts at all the levels are zeros: thereafter we have the following subscripts assuming that all the $S[\ell]_t$'s are unknowns.

Figure 6: Dataflow for the first 8 cycles.

t	$\ell=0$	$\ell=1$	$\ell=2$	$\ell=3$
0	0	0	0	0
1	0	$4\cdot S[0]_0$	$2\cdot S[1]_0$	$S[2]_0$
2	0	$4\cdot S[0]_1$	$4\cdot S[0]_0+2\cdot S[1]_1$	$2\cdot S[1]_0+S[2]_1$
3	0	$4\cdot S[0]_2$	$4\cdot S[0]_1+2\cdot S[1]_2$	$4\cdot S[0]_0+2\cdot S[1]_1+S[2]_2$
4	0	$4\cdot S[0]_3$	$4\cdot S[0]_2+2\cdot S[1]_3$	$4\cdot S[0]_1+2\cdot S[1]_2+S[2]_3$
5	0	$4\cdot S[0]_4$	$4\cdot S[0]_3+2\cdot S[1]_4$	$4\cdot S[0]_2+2\cdot S[1]_3+S[2]_4$
6	0	$4\cdot S[0]_5$	$4\cdot S[0]_4+2\cdot S[1]_5$	$4\cdot S[0]_3+2\cdot S[1]_4+S[2]_5$
7	0	$4\cdot S[0]_6$	$4\cdot S[0]_5+2\cdot S[1]_6$	$4\cdot S[0]_4+2\cdot S[1]_5+S[2]_6$

Note that the above follows since $e_\ell = 2^{n-d-1-\ell}$ as an integer, and therefore $b + S[\ell]_t \cdot e_\ell = b + S[\ell]_t \cdot 2^{n-d-1-\ell}$. We have already seen that for this to serve our purpose, the integer values of the last column of the above table should be 0 to 7. In other words we need $S[\ell]_t$'s from the set $\{0,1\}$ which are solutions of the following system of equations over the integers.

$$\begin{aligned}
S[2]_0 &= 1\\
2\cdot S[1]_0 + S[2]_1 &= 2\\
4\cdot S[0]_0 + 2\cdot S[1]_1 + S[2]_2 &= 3\\
4\cdot S[0]_1 + 2\cdot S[1]_2 + S[2]_3 &= 4\\
4\cdot S[0]_2 + 2\cdot S[1]_3 + S[2]_4 &= 5\\
4\cdot S[0]_3 + 2\cdot S[1]_4 + S[2]_5 &= 6\\
4\cdot S[0]_4 + 2\cdot S[1]_5 + S[2]_6 &= 7
\end{aligned}$$

It can be verified that the assignments to the $S[\ell]_t$'s in Figure 6 satisfy the above equation system. We address the issue of the general case with the following theorem.

Theorem 1. *Given the circuit* ***Polymob1*** *in Figure 5, in which each of the registers have been initialized with the ANF vectors $A[\ell]_{0^{n-d}}$ for all $0 \le \ell \le n-d$. Then it is possible to engineer the multiplexer signals $S[\ell]_t$ for $0 \le t \le 2^{n-d}-2$, so that the circuit computes the Möbius Transform of an n-variable Boolean function of algebraic degree upto d in exactly 2^{n-d} clock cycles.*

Proof. We essentially have to prove that we can engineer the multiplexer signals $S[\ell]_t$ efficiently so that the subscripts of the ANF vectors at the bottommost i.e. level $n-d$, at $t = 0 \to 2^{n-d}-1$ are each equal to t itself. Generalizing the observations made above, we need $S[\ell]_t$'s from the set $\{0,1\}$ which are solutions of the following system of equations over the integers. Let $u := n-d$. Let i be a sequence variable and set $j := u-1-i$ for conciseness, then we have

$$\begin{array}{llllll}
 & & & & S[u-1]_0 & = 1\\
 & & & 2\cdot S[u-2]_0 & + S[u-1]_1 & = 2\\
 & & & & & \vdots\\
 & & & 2^i\cdot S[j]_0 & + \cdots + S[u-1]_i & = i+1\\
 & & & & & \vdots\\
2^{u-1}\cdot S[0]_0 & + 2^{u-2}\cdot S[1]_1 & + \cdots + 2^i\cdot S[j]_j & + \cdots & + S[u-1]_{u-1} & = u\\
2^{u-1}\cdot S[0]_1 & + 2^{u-2}\cdot S[1]_2 & + \cdots + 2^i\cdot S[j]_{j+1} & + \cdots & + S[u-1]_u & = u+1\\
 & & & & & \vdots\\
2^{u-1}\cdot S[0]_{2^u-u-1} & + 2^{u-2}\cdot S[1]_{2^u-u} & + \cdots + 2^i\cdot S[j]_{-i+2^u-2} & + \cdots & + S[u-1]_{2^u-2} & = 2^u-1
\end{array}$$

To solve the above equation system, observe that the right side always has a u-bit integer i.e. between 1 and 2^u-1. Not only that, the left side of each equation resembles

the decimal expansion of a u-bit binary string. For example the LHS of the last equation is the decimal expansion of the u-bit binary string $S[0]_{2^u-u-1},\ S[1]_{2^u-u}, \cdots,\ S[u-1]_{2^u-2}$. Thus a trivial way to solve the above equation system is to assign to the unknowns the values obtained from the binary representation of the corresponding integer in the right side. For example, since the binary form of 2^u-1 is the u-bit string of all 1s we can assign $S[0]_{2^u-u-1} = S[1]_{2^u-u} = \cdots = S[u-1]_{2^u-2} = 1$.

Thus we can see that a solution to the above equation system exists: however we will further show that each of the signals $S[\ell]_t$ can be efficiently generated using a reasonable amount of logic circuits. Using the method outlined above, we can immediately see that $S[u-1]_t = t+1 \bmod 2$ for all t. With some misuse of notation the above can be written as **NOT** ($t \bmod 2$), i.e. if we have a decimal up-counter implementing t, then the $S[u-1]_t$ signal can be implemented by inverting the least significant bit of t. Similarly the sequence $S[u-2]_t,\ t = 0, 1, 2, \ldots$ is the second lsb of the sequence $2, 3, 4, \ldots$, i.e. the second lsb of $t+2$. For the general case, let us look at the i-th column from the end of the above equation system which has been highlighted in green. It can be seen that the sequence $S[j]_t = S[u-1-i]_t,\ t = 0, 1, 2, \ldots$ is the $i+1$-th lsb of the sequence $(i+1), (i+2), (i+3), \ldots$, i.e. the $(i+1)$-th lsb of $t+i+1$. Thus to construct all the signals $S[\ell]_t$ all we need are the following circuit elements:

1. A u-bit decimal up-counter for the variable t.
2. A series of u incrementers (i.e. add by 1 circuits) to generate $t+1,\ t+2, \ldots, t+u$.

This proves the theorem statement.

□

Theorem 2. *Furthermore it is possible to design a control circuit that generates all the select signals of the multiplexers in the* ***Polymob1*** *circuit, incurring a total delay of* $2\log_2(n-d)$ *gates.*

Proof. As noted in the proof of Theorem 1, the control circuit consists of a u-bit decimal up-counter (where $u := n-d$) for the variable t and a series of u incrementers. However constructing the whole incrementer leads to a wastage of gates since we are only interested in generating the $(i+1)$-th lsb of $t+i+1$ for $i = 0, 1 \ldots, u-1$.

Consider any p-bit string $\vec{w} = w_{p-1}, w_{p-2}, \ldots, w_1, w_0$ (note that the indexing with starts from right side in this definition). Define the p-variable Boolean function $g_{p,\vec{w}}$ as follows

$$g_{p,\vec{w}} = \begin{cases} \prod_{w_i=0} \left[x_i \vee \bigvee_{j=i+1:w_j=1}^{p-1} x_j \right] \\ 1, \text{ when } p = 0 \text{ or } \vec{w} = 1^p. \end{cases}$$

For example the function $g_{8,\ 0001\ 0100} = (x_0 \vee x_2 \vee x_4) \cdot (x_1 \vee x_2 \vee x_4) \cdot (x_3 \vee x_4) \cdot x_5 \cdot x_6 \cdot x_7$ and $g_{3,111} = 1$. Each product term begins with a min index that has 0 in the sting $\vec{w}$. In the first example, in $\vec{w}$ indices 0,1,3,5,6,7 have 0. Then each min index is **OR**ed with indices larger than it that have 1 in $\vec{w}$. Further, if the length of $\vec{w}$ is more than p, we truncate $\vec{w}$ to its p least significant bits. We will prove that the $(i+1)$-th lsb of $x+i+1$ is given by the Boolean function $x_i \oplus g_{i,bin_i(i)}$, where $bin_i(i)$ is the binary encoding of i using i bits, i.e. prepended with leading zeros when necessary. For small i, this is easy to verify. Denoting x_j as the Boolean variable for the j-th bit of x, we know that for $i = 0$, the 1st lsb of $x+1$ is given by $x_0 \oplus 1 = x_0 \oplus g_{0,0}$. For $i = 1$, the 2nd lsb of $x+2$ can be computed thus: when we add with 2, i.e. the string "10" the 1st lsb location generates no carry. The result of addition in the 2nd lsb location is therefore $x_1 \oplus 1 \oplus 0 = 1 \oplus x_1 = x_1 \oplus g_{1,1}$.

For general values of i, we proceed as follows. Let $bin_i(i) = c_{i-1}, c_{i-2}, \ldots, c_0$, where each $c_j \in \{0, 1\}$. Of these let the locations $0 \leq n_1 < n_2 < \cdots < n_s \leq i-1$ be such that $c_{n_k} = 1$ for $k = 1$ to s, and the remaining c_j's be 0. When adding two strings a, b, the carry out bit in the j-th position can be written as **maj**$(a_j, b_j, carry_{j-1})$ (where **maj** is

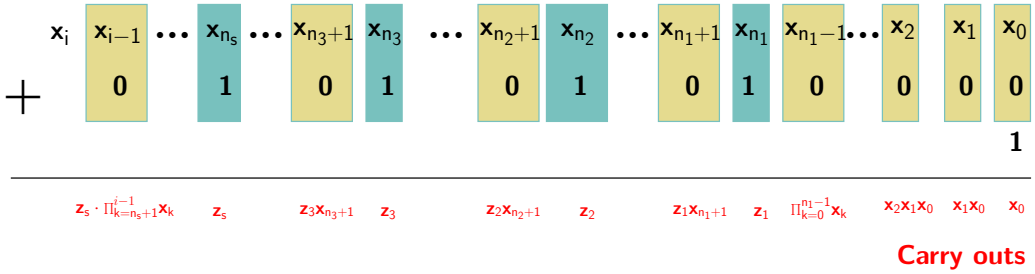


Figure 7: Visual representation of the addition

the majority function). We use two properties of this function: **(1) maj**$(x, y, 0) = xy$ and **(1) maj**$(x, y, 1) = x \vee y$. Figure 7 visually represents the process of addition by the constant $i + 1$. Using the above property of the majority function, the figure becomes self explanatory: however we still have to explain the symbols z_j for $j = 1$ to s, which are the carry-outs for the position n_j. By using the second property, we have

$$z_1 = x_{n_1} \vee \prod_{k=0}^{n_1-1} x_k = \prod_{k=0}^{n_1-1} (x_{n_1} \vee x_k) = g_{n_1,i}(x)$$

The above follows because of the Boolean identity $A \vee BC = (A \vee B)(A \vee C)$, and i in the subscript of g is the truncation of $bin_i(i)$ to the appropriate number of bits. Following the same logic we now have

$$\begin{aligned} z_2 &= x_{n_2} \vee \left(z_1 \cdot \prod_{k=n_1+1}^{n_2-1} x_k \right) \\ &= (x_{n_2} \vee z_1) \cdot \prod_{k=n_1+1}^{n_2-1} (x_{n_2} \vee x_k) = \left(x_{n_2} \vee \prod_{k=0}^{n_1-1} (x_{n_1} \vee x_k) \right) \cdot \prod_{k=n_1+1}^{n_2-1} (x_{n_2} \vee x_k) \\ &= \prod_{k=0}^{n_1-1} (x_{n_2} \vee x_{n_1} \vee x_k) \cdot \prod_{k=n_1+1}^{n_2-1} (x_{n_2} \vee x_k) = g_{n_2,i}(x) \end{aligned}$$

Following this chain of arguments, it is straightforward to show that $z_s = g_{n_s,i}(x)$ and that the carry out of the $(i-1)$-th location is $z_s \cdot \prod_{k=n_s+1}^{i-1} x_k = g_{i,bin_i(i)}(x)$. Thus it follows that the sum we are looking for is $x_i \oplus g_{i,bin(i)}(x)$.

The expression for $g_{i,bin(i)}(x)$ naturally has the longest circuit depth for $i = u - 1 = n - d - 1$. The number of product terms in the expression is bounded by $n - d$. Therefore the depth required to construct the product terms, if the and gates were arranged in a binary tree like manner is around $\log_2(n - d)$. Furthermore, each collection of bracket containing terms that are **OR**-ed together can also have a maximum of $n - d$ terms, which implies each such term can also be constructed using $\log_2(n - d)$ depth. Putting this together we arrive at $2 \cdot log_2(n - d)$. We also have to account for the decimal up-counter t which counts from $0 \to 2^{n-d} - 1$ in steps of 1. But it is well known in circuit theory that the maximum depth required in this up-counter is only $log_2(n - d)$ (i.e. for the update bit of the msb flip-flop which is $t_{n-d-1} \oplus \prod_{k=0}^{n-d-2} t_k$). □

3.3.2 Representation of the ANF vector:

So far we have avoided some of the finer operational details of the circuit to concentrate on the macro-level issues of dataflow through the circuit. One of the important topics we

have not dealt with so far, is the issue of representing any degree-limited ANF coefficient set as a bit vector. The uncompressed ANF vector of an n-variable Boolean function has 2^n entries and mapping each coefficient into an array can be done canonically as explained earlier in Sec 2 and further shown in Figure 2. For example the x_0x_2 term has coefficient 1: since the term can be written as $x^{101} := x_0^1 \cdot x_1^0 \cdot x_2^1$, the exponent vector 101 (5 in decimal) denotes the position where a one is inserted in the array. However when we are dealing with functions of a small degree d, coefficients of all terms of degree larger than d are zero and so in order to accommodate the potentially $\binom{n}{\downarrow d}$ non-zero coefficients we must be able to map them into an array of equal length, i.e. we need to decide which array location a given coefficient is going to reside in. This is important to decide for the following reasons

1. We have left the issue of the ordering in Lines 1,2 in Algorithm 1 open. The ANF vector should be so represented so that constructing the vectors $A_{\mathsf{top}}/A_{\mathsf{bottom}}$ from A_0 should be efficient at all levels of recursion.

2. The ANF representation should be such that we can efficiently use **Expmob1** at the leaf nodes of the recursion tree.

3. The circuit constructed for some $n = n^*,\ d = d^*$, should produce correct result when used for all $n < n^*$ and $d < d^*$, i.e. the circuit should work seamlessly for all smaller and lower degree Boolean functions.

Let $H(n, d)$ be the set of all binary strings of length n whose hamming weight is less than or equal to d, where we will treat the elements of this set as both binary strings and integers. The goal is to construct a mapping $\chi_{n,d} : H(n, d) \rightarrow \left[0, \binom{n}{\downarrow d} - 1\right]$, so that the coefficient of the x^D term for any $D \in H(n, d)$ is placed at location $\chi_{n,d}(D)$ in the compressed ANF vector. From the description of **Expmob1** in Section 3, the following things can be seen

a) if we are using a butterfly circuit to construct the derivative wrt to any variable x_ℓ, then the two inputs to the circuit are the coefficients at x^D and $x^{D\oplus e_\ell}$. Wlog, let us assume that the ℓ-th bit of D is zero i.e. $D \cdot e_\ell = 0$.

b) The coefficient of x^D is copied as is from A_0 to A_{top} (or if we follow the terminology developed later: from $A[\ell]_b$ to $A[\ell+1]_b$). The coefficients of x^D and $x^{D\oplus e_\ell}$ are added and copied to $A[\ell + 1]_{b+e_\ell}$.

c) If D be such that $hw(D \oplus e_\ell) > d$, then this last addition is not necessary since the coefficient of $x^{D\oplus e_\ell}$ is 0 by assumption.

However note that the size of the vectors $A[\ell]_b$ and $A[\ell + 1]_b$ are $\binom{n-\ell}{\downarrow d}$ and $\binom{n-\ell-1}{\downarrow d}$ respectively. So if any $D \in H(n - \ell, d)$, then we ought to not only decide what $\chi_{n-\ell,d}(D)$ would be but also in which locations of $A[\ell + 1]_b/A[\ell + 1]_{b+e_\ell}$, the butterfly outputs would go to. It seems we need to determine a series of mappings $\chi_{n-\ell,d}$, however we will see how only unified mapping will take care of our requirements. Let $\chi_{n,d}(D) = y$ if D be the y-th largest integer with hamming weight less than or equal to d. For example when $n = 8,\ d = 2$, we have $\chi_{8,2}(u) = u$ for $0 \leq u \leq 6$, and $\chi_{8,2}(8) = 7$, $\chi_{8,2}(9) = 8$, $\chi_{8,2}(10) = 9$, $\chi_{8,2}(12) = 10$ etc.

Note that we have $f(x_0, x_1, x_2, \ldots) = x_0 \cdot \frac{\delta f}{\delta x_0}(x_1, x_2, \ldots) \oplus f(0, x_1, x_2, \ldots)$. Rewrite this as $x_0 \cdot f_1(x_1, x_2, \ldots) \oplus f_2(x_1, x_2, \ldots)$, where $f_1 = \frac{\delta f}{\delta x_0}$ and $f_2 = (0, x_1, x_2, \ldots)$. Note that $A[1]_{00\ldots}$ gets the ANF vector of f_2 and $A[1]_{10\ldots}$ gets the ANF vector of $f_1 \oplus f_2$ after the butterfly operation. Therefore we have the following transitions

1. Algebraically f_2 is simply all terms of f with the terms containing x_0 removed. In terms of ANF, f_2 is therefore simply the terms contained at the indices of type $0 \parallel s$

(where s is any $(n-1)$-bit string) in the uncompressed ANF vector. These are simply copied to the index s in f_2. In the compressed world, therefore, all entries at location $\chi_{n,d}(0 \parallel s)$ of $A[0]_{00\ldots}$ should go to location $\chi_{n-1,d}(s)$ of $A[1]_{00\ldots}$. However note that when expanded as integers, $0 \parallel s$ and s give rise to the same integer. Thus for ease of use $\chi_{n-1,d}(s)$ can simply be denoted as $\chi_{n,d}(0 \parallel s)$, and if we view the arguments of these functions as integers we do not need to define any $\chi_{n-i,d}$ separately.

2. Similarly for $f_1 \oplus f_2$, in the uncompressed form, all terms at indices $0 \parallel s$ are added with terms at $1 \parallel s$ and copied to $0 \parallel s$. Thus in the compressed form we should add terms at locations $\chi_{n,d}(0 \parallel s)$, $\chi_{n,d}(1 \parallel s)$ (if $1 \parallel s$ has hamming weight less than or equal to d) of $A[0]_{00\ldots}$ and copy it to location $\chi_{n,d}(s)$ of $A[1]_{10\ldots}$.

3. The same idea applies to all the levels of the recursion tree.

4. Note that in $\chi_{n,d}$ all integers of hamming weight less than or equal to d are mapped to itself. Thus at the lowest leaves of the recursion tree, we can apply the canonical version of Möbius Transform as used in **Expmob1**.

We are yet to determine if the mapping $\chi_{n,d}$ can be computed efficiently. The following lemma addresses this computational issue.

Lemma 1. *For positive integers n, d with $d \leq n$, and $s \in H(n,d)$, let $s = 2^{i_0} + 2^{i_1} + \cdots$, be the binary expansion of the integer s, where $i_0 > i_1 > \cdots \geq 0$. Then we have*

$$\chi_{n,d}(s) = \binom{i_0}{\downarrow d} + \binom{i_1}{\downarrow d-1} + \cdots$$

where we extend the definition of $\binom{x}{\downarrow y}$ as follows:

$$\binom{x}{\downarrow y} = \begin{cases} \sum_{i=0}^{y} \binom{x}{i} & \text{if } x \geq y, \\ 2^x & \text{otherwise.} \end{cases}$$

Proof. As per the definition of $\chi_{n,d}$, given s we have to count how many integers strictly less than s have hamming weight bound by d. It is obvious that this number for any 2^m is simply $\binom{m}{\downarrow d}$, i.e. number of m-bit strings of hamming weight less than or equal to d. Hence the number of such strings in the range $[0, 2^{i_0})$ is $\binom{i_0}{\downarrow d}$. The number of such integers in the range $[2^{i_0}, 2^{i_0} + 2^{i_1})$ are strings which have 1 in the i_0-th position and of hamming weight less than or equal to $d-1$ in the last i_1 bits, and therefore equal to $\binom{i_1}{\downarrow d-1}$. Taking this argument forward for the successive $i_2,\ i_3, \ldots$, we arrive at the required result. □

3.4 Helping Circuit Compiler synthesize faster

The above lemma shows that the map $\chi_{n,d}(\cdot)$ can be efficiently computed. However for ease of synthesis, one may want to precompute and store a few of the above values to help the circuit compiler construct an optimal circuit especially when n becomes larger. One could store all values of $\chi_{n,d}(s),\ \forall s \in H(n,d)$ for this purpose, but note that the arguments "s" of this function are not exactly contiguous integers and thus we would not be able to store the function table in any continuous memory structure like an array. We could employ a hash table for this purpose, however designing a good collision free hash function for this purpose is an open problem.

Another method we could employ is to store the adjacency matrix of a graph that we describe below. Note that at the ℓ-th recursion step, we need access to locations $\chi_{n,d}(0 \parallel s)$, $\chi_{n,d}(1 \parallel s)$ of the current register, where s is an $n-\ell-1$ bit string. Imagine the graph $G = (V, E)$, in which the elements of $\left[0, \binom{n}{\downarrow d} - 1\right]$ are nodes and each node α in this set is

connected with at most $n-d$ types of edges to at most $n-d$ neighbors. An edge of type ℓ, (for $0 \leq \ell < n-d$) connects α to $\beta := \chi_{n,d}\left[\chi_{n,d}^{-1}[\alpha] \oplus e_\ell\right]$ if $hw(\beta) \leq d$ and unconnected otherwise. This is helpful because at step ℓ of the recursion tree, if $\alpha = \chi_{n,d}(0 \parallel s)$ then the two inputs to the butterfly circuit can be equivalently seen as the wires at locations α and β, as it can be easily deduced that $\beta = \chi_{n,d}(1 \parallel s)$. One can now define the reduced adjacency matrix AM of size $\binom{n}{\downarrow d} \times (n-d)$ such that

$$AM[\alpha, \ell] = \begin{cases} \chi_{n,d}\left[\chi_{n,d}^{-1}[\alpha] \oplus e_\ell\right], & \text{if } hw(\chi_{n,d}^{-1}[\alpha] \oplus e_\ell) \leq d \\ 0 & \text{otherwise.} \end{cases}$$

Thus the ℓ-th recursion step can be re-written from:

- For all $n-\ell-1$ bit strings s with hw $\leq d$
  1. $A[\ell+1]_b(\chi_{n,d}(s)) \leftarrow A[\ell]_b(\chi_{n,d}(0 \parallel s))$
  2. If $hw(\chi_{n,d}(1 \parallel s)) \leq d$:
     $A[\ell+1]_{b+e_\ell}(\chi_{n,d}(s)) \leftarrow A[\ell]_b(\chi_{n,d}(0 \parallel s)) \oplus A[\ell]_b(\chi_{n,d}(1 \parallel s))$
  3. Else $A[\ell+1]_{b+e_\ell}(\chi_{n,d}(s)) \leftarrow A[\ell]_b(\chi_{n,d}(0 \parallel s))$

to the following equivalent form that uses the AM matrix:

- For $\alpha = 0$ to $\binom{n-\ell-1}{\downarrow d} - 1$
  1. $A[\ell+1]_b(\alpha) \leftarrow A[\ell]_b(\alpha)$
  2. If $AM[\alpha, \ell] \neq 0$
     $A[\ell+1]_{b+e_\ell}(\alpha) \leftarrow A[\ell]_b(\alpha) \oplus A[\ell]_b(AM[\alpha, \ell])$
  3. Else $A[\ell+1]_{b+e_\ell}(\alpha) \leftarrow A[\ell]_b(\alpha)$

Using the 2nd description is much easier to write an RTL code for describing the Möbius Transform circuit in any hardware description language. Additionally, the circuit compiler also outputs the optimized netlist faster. In Appendix A, we outline an algorithm to generate AM efficiently in polynomial time.

3.5 Further Utilities

Using the circuit for smaller functions: The circuit once constructed for some upper limit (n, d) also caters for Boolean functions for any number of variables $n_0 < n$. Since any n_0-variable Boolean function (for $n_0 < n$) is also an n-variable Boolean function, i.e. with the additional variables set to zero, the only thing we need to do is to embed the ANF of the n_0-variable Boolean function as a the ANF of an n-variable Boolean function with appropriate zero padding. This is aided by the fact that $\chi_{n,d}$ has been defined in a manner so that $\chi_{n-1,d}(s)$ is the same as $\chi_{n,d}(0 \parallel s)$ for any $(n-1)$-bit string s. Thus in order to embed any $(n-1)$-variable Boolean function we simply add the coefficient corresponding to s in $\chi_{n,d}(0 \parallel s)$ and place 0 in $\chi_{n,d}(1 \parallel s)$. Since the function $\chi_{n,d}$ is monotonous this would amount to filling up locations $\left[0, \binom{n-1}{\downarrow d} - 1\right]$ with coefficients of the smaller Boolean function and padding the remaining i.e. $\left[\binom{n-1}{\downarrow d}, \binom{n}{\downarrow d} - 1\right]$ locations with zeros. By induction on i, the same applies to any arbitrary $(n-i)$-variable function.

Finding truth table when some variables are fixed to constants: Often one is interested to find solutions to a system of equations in which a fraction of variables has been fixed to some given constant [Cou02, CM03]. Since our strategy in solving a system of polynomial

equations is to compute the **OR** the respective truth tables (see Sec 1.1), we would therefore be interested to find the truth table of a polynomial when some variables are fixed. To do this, we could either first simplify the given Boolean polynomial by fixing some individual variables to constants and then using the corresponding reduced ANF vector as input to the circuit, after appropriately zero padding it. However this naturally requires additional computations, i.e to simplify the original polynomial in the first place.

However if the t variables to be fixed are lexicographically the first t variables of the system (for any $t \leq n-d$) then we can do better. We see from Figure 6, that at the i-th stage the ANF vector at the bottom most register is $A[n-d]_{bin_{n-d}(i)}$. As a result the truth table output after the **Expmob1** circuit is $f(bin_{n-d}(i), \ldots)$, i.e. in which the first $(n-d)$ bits of f is already set to $bin_{n-d}(i)$. Thus one can use this method to extract the truth tables when the number of variables to be fixed are less than $n-d$. Note that one may think that one would need to wait exactly i cycles to obtain the tables, which can be counterproductive if i is large. Note that Fig 6 already starts with $A[3]_{000}$ in the bottom most register at $t=0$, as in the previous 3 cycles, i.e. $t=-3,-2,-1$, the corresponding $S[i]_t$'s were all set to zeros. Instead if all of these were set to 1, then at $t=0$, the signal in the bottom most register would be $A[3]_{111}$, and we would get the truth table of $f(1,1,1,\ldots)$ from the **Expmob1** circuit. Similarly by adjusting the initial select signals we can get the truth table where the first $(n-d)$ variables are fixed to any arbitrary constant in the first cycle itself.

3.6 Synthesis Results

For the actual synthesis, we can do some optimizations as follows: In figure 6, we can see that the topmost register of size $\binom{n}{\downarrow d}$ essentially holds a constant value throughout the lifetime of the Möbius Transform operation, and as such it can be removed from the circuit if the ANF signal is assumed as available on the input wires to the circuit. Using this tweak, we again synthesized the **Polymob1** circuit using the Nangate 15 nm open cell library for various values of $n \in [8, 20]$ and $d \in [2, 4]$. Note that the values of d chosen apply to a number of instances of cryptanalytic problems known in literature. For example, it is known that cryptanalysis of signature schemes like UOV amounts to solving a set of quadratic ($d=2$) Boolean equations. The public key in the signature scheme **PICNIC v3.0**, consists of a single plaintext/ciphertext pair generated by the LowMC block cipher using the secret key as the block cipher key. The designers recommend using 4-round instances of the block cipher for this purpose, which reduces the relations between the plaintext, ciphertext and key to Boolean equations of degree 4 in the unknown key [Din21]. Thus finding the secret key amounts to cryptanalysis of the block cipher using the single plaintext/ciphertext pair available as the public key of the signature, which amounts to solving n degree 4 equations in n unknowns.

The results are presented in Table 1. As stated earlier, the circuits were synthesized to minimize the total critical path, which allows us to clock them using higher frequencies. The minimum time T_{min} taken to compute the transform is calculated as $2^{n-d} + (n-d)$ times the critical path T_{cr} of the circuit. Since T_{cr} increases logarithmically and the number of cycles increases exponentially wrt $(n-d)$, some interesting tradeoffs can be observed: for example to compute the Möbius Transform of quadratic Boolean functions one may either use the circuit for $d=2,3$ or 4. Because of the exponential dependence on $n-d$, the total physical time taken to compute the transform undoubtedly decreases with increase in d: however it has to be paid for with larger circuit area and energy consumption. Furthermore, from a comparison between Tables 1 and 2, it can also be seen that the energy consumed by the **Polymob1** circuits is an order of magnitude larger than the corresponding **Expmob1**/**Expmob2** circuits for similar values of n. This is to be expected primarily because **Polymob1** is essentially a serialized circuit that performs the transform using exponential amount of time (in $n-d$) whereas **Expmob1**/**Expmob2** either

Table 1: Results for $d = 2, 3, 4$ for the **Polymob1** circuit. Power reported at 10 MHz.

	$d = 4$					$d = 3$				
n	Area GE	T_{cr} (ps)	T_{min} (ns)	Power (μW)	Energy (nJ)	Area GE	T_{cr} (ps)	T_{min} (ns)	Power (μW)	Energy (nJ)
8	2078	40.57	0.811	13.04	0.026	1468	41.11	1.521	8.93	0.033
9	3535	44.87	1.660	21.19	0.078	2265	41.47	2.903	13.46	0.094
10	5780	46.77	3.274	33.44	0.234	3338	45.48	6.134	19.61	0.265
11	9040	66.95	9.038	52.35	0.706	4682	60.25	15.906	27.40	0.723
12	13525	75.95	20.051	76.79	2.027	6563	48.69	25.367	37.87	1.973
13	20024	61.74	32.167	109.80	5.720	8900	74.46	76.992	50.73	5.245
14	28754	73.61	76.113	157.00	16.233	11792	72.00	148.248	66.65	13.723
15	40706	73.18	150.678	222.00	45.709	15228	74.82	307.361	85.97	35.315
16	56683	79.71	327.449	309.40	127.101	19457	68.95	565.735	109.90	90.173
17	77402	78.07	640.564	420.60	345.102	24253	83.31	1366.117	138.80	227.605
18	102916	90.26	1480.084	562.60	921.567	30839	81.32	2665.014	173.90	570.096
19	134835	93.46	3063.899	735.80	2412.173	37781	76.57	5019.317	213.20	1397.569
20	174268	101.48	6652.217	951.40	6236.617	45653	83.87	10994.434	260.20	3410.936

	$d = 2$				
n	Area GE	T_{cr} (ps)	T_{min} (ns)	Power (μW)	Energy (nJ)
8	856	49.24	3.445	5.23	0.036
9	1160	51.82	6.996	7.14	0.096
10	1506	47.20	12.461	8.96	0.237
11	1984	58.09	30.265	11.67	0.608
12	2552	69.16	71.511	14.49	1.498
13	3123	65.45	134.761	17.77	3.658
14	3817	70.22	288.464	21.26	8.731
15	4584	70.96	582.227	25.47	20.896
16	5445	71.81	1177.540	30.43	49.896
17	6495	74.33	2346.761	36.23	118.756
18	7575	75.33	4938.032	42.43	278.157
19	8766	86.07	11282.830	49.35	646.950
20	10168	81.94	21481.554	57.22	1500.091

take constant or linear time to execute.

4 Solving Polynomial equations of degree $\leq d$

One of the primary uses of the Möbius Transform circuit is in finding solutions of a system of Boolean polynomials bounded by some algebraic degree d. Recapitulating, if $f_1,\ f_2,\ f_3, \ldots,\ f_m$ are the m polynomials whose common root we are aiming to find, then the root $r \in \{0,1\}^n$ is an n-bit vector which simultaneously satisfies $f_1(r) = f_2(r) = \cdots = f_m(r) = 0$. We can combine the above in a single equation:

$$\bigvee_{i=1}^{m} f_i(r) = 0$$

In other words, we take the truth tables of each f_i and cumulatively compute the logical **OR** of them. The common root(s) will be indices at which the vector of cumulative **OR** of the tables have 0 in them. However note that the Möbius Transform circuit we have constructed outputs the truth table in parts. We have seen that when at the lowest register the ANF vector is $A[n-d]_b$, then the circuit outputs the truth table of the function $f(b, x_{n-d}, \ldots, x_{n-1})$, i.e. when the first $n-d$ bits have been set to the constant b. With this information let us begin to see circuit configurations that compute the root of a system of equations.

4.1 Polysolve1

Let's say we have m Boolean equations f_i we need to simultaneously solve. We begin by having m copies of the Möbius Transform circuit in parallel, each for one of the polynomials f_i. At the b-th step, we have the truth tables $f_i(b, \ldots)$ output from each of the circuits.

We then have a layer of **OR** gates to compute the logical disjunction of all the truth tables. After we have done this we have the combined truth table vector of length 2^d bits, whose zeroes give us the roots of the equation. The following cases may occur

- The vector is the all 1 bit-string. This indicates that there is no root of the underlying system in which the first $n-d$ bits are set to the constant b.
- The vector contains a single 0 at some position t, whose binary encoding is given by $bin_d(t)$. In this case the root of the system of equations is $b \parallel bin_d(t)$.
- The vector contains a multiple 0s, which indicates that there are multiple roots of the system beginning with b. We may wish to find all such roots, or any one of them. For the moment let us concentrate on finding any one of them.

If the task is to find only one such root, the most efficient way to find this would be a priority encoder, that will encode to binary the first occurrence of zero in the 2^d-bit string. The circuit is described pictorially in Figure 8a. We describe some micro-level details of the circuit below:

OR Network: In order to compute the disjunction of m vectors of length 2^d each, it is obvious that we need $(m-1)\cdot 2^d$ number of 2-input **OR** gates. However we can ensure that the network has a total latency bounded by $\lceil \log_2 m \rceil$ **OR** gates by arranging the gates in inverted binary tree like manner, in which each level would contain around half the gates contained in the previous level. For example if $m = 8$, the first level would have a total of 4 **OR** gates of width 2^d bits, the next level 2, and the final level a single gate. This makes the total latency of the network equal that incurred in three **OR** gates.

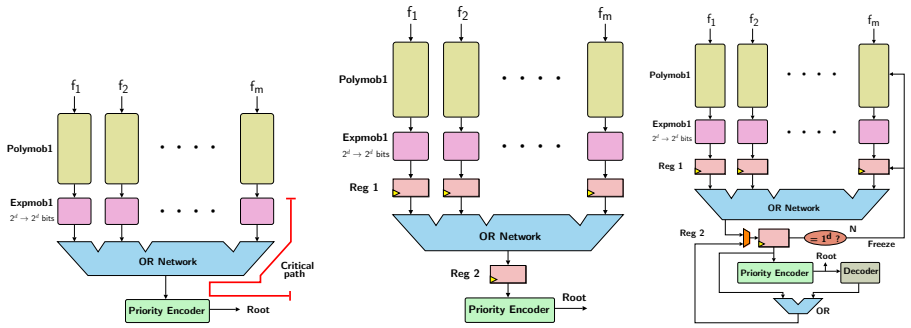
Priority Encoder: For a $2^d \rightarrow d$ bit priority encoder, the functionality can be simply described as a look-up table if d is small enough. For larger d, we can also use recursive description of the encoder functionality. In both cases, the critical path in the encoder is known to be proportional to d gates.

Circuit Area: If we do away with the first level register in the **Polymob1** circuit the total number of scan flip-flops required for the successive registers in the m **Polymob1** instances is $m \cdot \sum_{i=1}^{n-d} \binom{n-i}{\downarrow d} < m \cdot (n-1)^{d+1} < m \cdot n^{d+1}$. It is not difficult to work out that the total number of xor gates required is $m \cdot \sum_{i=1}^{n-d} \binom{n-i-1}{\downarrow d}$. Since the area of a 2-input xor gate is much less than that of a scan flip-flop the total area can be loosely upper bound by $m \cdot n^{d+1}$. The remaining circuit elements contribute $md \cdot 2^{d-1}$ xor gates (for the **Expmob1** circuits), $(m-1)\cdot 2^d$ **OR** gates (for the **OR** network) and the area required for the priority encoder (this will be proportional to 2^d). For small d, this can be ignored wrt $m \cdot n^{d+1}$.

Total Critical Path: As shown in Figure 8a, the total critical path in this architecture is due to the combination of **Expmob1**, the **OR** network and the encoder (call this τ_A). Since we have seen that each Möbius Transform takes around 2^{n-d} clock cycles to output all the truth tables, this implies that the circuit will take at least $\tau_A \cdot 2^{n-d}$ amount of physical time to solve the system of equations.

4.2 Polysolve2

In order to break up the long chain of combinatorial circuitry after the Möbius Transform computations one could install pipeline stages in between them as shown in Figure 8b. The introduction of the pipeline stages requires only $m+1$ registers of size 2^d bits each (m for Reg1 and one more for Reg2 as shown in Figure 8b) and reduces the delay caused due to the long chain of combinatorial elements, and increases the computation time by only two cycles. However the breaking up of this combinatorial path means that the critical path



(a) **Polysolve1** (b) **Polysolve2** (c) **Polysolve3**

Figure 8: Circuits for solving m equations

in the circuit will now most likely be due to the series of $u = n - d$ incrementer circuits required for generating the select signals for the multiplexers in the Möbius Transform circuit or the optimized version of it described in Theorem 2.

Note that (as we shall see shortly) for smaller values of d that we report in this paper (i.e less than 4), there is not much difference between the critical paths of **Polysolve1** and **Polysolve2**. For smaller values pf d, the total critical path τ_A is not very high and the circuit compiler effectively balances out various parts of the netlist so that the total critical path of the **Polysolve1** circuit is comparable with the **Polysolve2** circuit. However as d increases, **Polysolve2** performs much better wrt total circuit latency.

4.3 Polysolve3

So far **Polysolve1**/**Polysolve2** have been focused to find only a single root in the series of partial truth tables generated from each f_i. However some applications may need the underlying hardware accelerator to find all the roots of a given equation system. There are a few solutions to the above problem we could consider. First, the circuit may choose to communicate the disjunction of partial truth tables back to the processor, without applying the encoder. The root would then be extracted by the processor using its own instruction set architecture. Second, instead of a priority encoder, the 2nd register (**Reg 2**) in Fig 8b, could be additionally equipped with bitwise shift functionality. After the disjunction of the m truth tables is loaded on to it, the bits would be shifted out serially with another counter maintaining the index of the bit shifted out. Now if one of the shifted out bits is zero, then the index counter can be used to construct the root. However this would require freezing the operations of the Möbius Transform circuit for exactly 2^d cycles, i.e. the Möbius Transform circuit does not produce another partial truth table till the processing of the current table is completed. This implies that the underlying registers of the **Polymob1** circuit would now actually need a 3:1 multiplexer preceding it to help in freezing the dataflow. However this increases the number of cycles required to execute the operation by a factor of 2^d i.e. from 2^{n-d} to $2^d \cdot 2^{n-d} = 2^n$ cycles.

However the solution we propose here will require exactly $R + 2^{n-d}$ cycles, where R is the total number of roots of the underlying equation system. The main issue arises when the disjunction of truth tables contains multiple zeros. In that case a priority encoder only fishes out the location of the zero which is numerically smallest. However consider the event when this actually happens: using the inverse of an encoder i.e. a decoder, one can convert the encoded vector V back to a 2^d vector of hamming weight one, with one at the V-th location. We then **OR** this vector with the current vector in **Reg 2** and update it in

the next cycle. The updated vector has one less 0 than the original vector in **Reg 2**. If this is now the all one vector then there are no more roots to fish out, else we repeat the process to decrease the number of zeros in **Reg 2** by one, till it has the all one vector.

Example 1. If $d = 4$, and the **OR** of the truth tables is $T_0 = 1011\ 1111\ 1111\ 0111$, then the priority encoder in the first cycle outputs 0001 which is the index of the first 0. The decoder outputs $D_0 = 0100\ 0000\ 0000\ 0000$, which after **OR** with T_0 becomes $T_1 = T_0 \vee D_0 = 1111\ 1111\ 1111\ 0111$, and has one less zero than T_0, and is written back to **Reg2**. In the next cycle we get the next root 1100 from the priority encoder which decodes to $D_1 = 0000\ 0000\ 0000\ 1000$. Therefore we have $T_2 = T_1 \vee D_1 = 1111\ 1111\ 1111\ 1111$ which is now the all one string.

During this time the **Polymob1** pipeline will have to be frozen (and thus a 3:1 mux functionality is needed in the **Polymob1** circuit), and it is not difficult to see that if each of the i disjunction of the partial truth tables (for $i \in [0, 2^{n-d} - 1]$) has r_i roots (with $\sum r_i = R$) then the i-th step will execute for exactly $r_i + 1$ cycles. To see why, note that there are two scenarios: **(a)** when the disjunction of the partial truth tables is the all one string, it means that that the pipeline immediately moves on to the next partial truth table and thus only spends one cycle here, and **(b)** when the string has one or more than one zero the mechanism reduces the number of zeros in the string by one every cycle, as explained above, till the all one string is reached. This needs $1 + r_i$ cycles. Therefore the total number of clock cycles required is $\sum_{i=0}^{2^{n-d}-1}(1 + r_i) = 2^{n-d} + \sum_{i=0}^{2^{n-d}-1} r_i = R + 2^{n-d}$. The circuit is described diagrammatically in Figure 8c.

Circuit Area: The only significant addition to the **Polysolve1** circuit is the additional 2:1 muxes before each of the scan flip-flops in the **Polymob1** circuit (thus achieving 3:1 multiplexer functionality). Thus the circuit area is now bound by $m \cdot n^{d+1}$ scan flip-flop and 2:1 muxes.

Total Critical Path: As n increases, the critical path is expected to be due to the select signal generation of the **Polymob1** circuit which has been shown to be proportional to $2 \cdot \log_2(n-d)$. If the underlying clock signal has this period then the total physical time taken to solve the system will be proportional to $2 \cdot \log_2(n-d) \cdot (2^{n-d} + R) \approx 2 \cdot \log_2(n-d) \cdot 2^{n-d}$.

4.4 Synthesis Results

In Figure 9, we present synthesis figures for the three solvers for the range of values of $n \in [8, 20]$ and $d \in [2, 4]$ (for a complete tabulation of the results please see Tables 3,4,5 in Appendix B). We have let $m = n$, so that the circuits directly output the roots of the underlying equation system. In [BCC+10], the authors had used a different strategy: if there were n equations to solve, then the hardware circuit consisted of only $\lceil \log_2 n \rceil$ parallel solvers that output around $2^{n-\lceil \log_2 n \rceil}$ potential candidate solutions for the system. These candidates had to undergo another filtering step (performed in software) to reveal the actual solutions of the system. While this approach is also possible in our work, we preferred to let the hardware circuit be self-sufficient, i.e. able to find the true roots of the system on its own. In the figure, we plot the silicon area, the total physical time taken (which is the product of the critical path and the number of clock cycles required) and the total energy consumed. The red and the blue curves for each plot (denoting the **Polysolve1** and **Polysolve2** circuit respectively) are almost coincident, which implies that for the range of values of d we have chosen, the circuits have similar performance metrics. The figures of time and energy for the **Polysolve3** circuit has been computed assuming that $R = n$. It is evident that the **Polysolve3** circuit is much larger than the corresponding **Polysolve1**/**Polysolve2** circuit since we need to use additional 2:1 multiplexer in each of the internal registers of the **Polymob1** circuit to periodically freeze the dataflow.

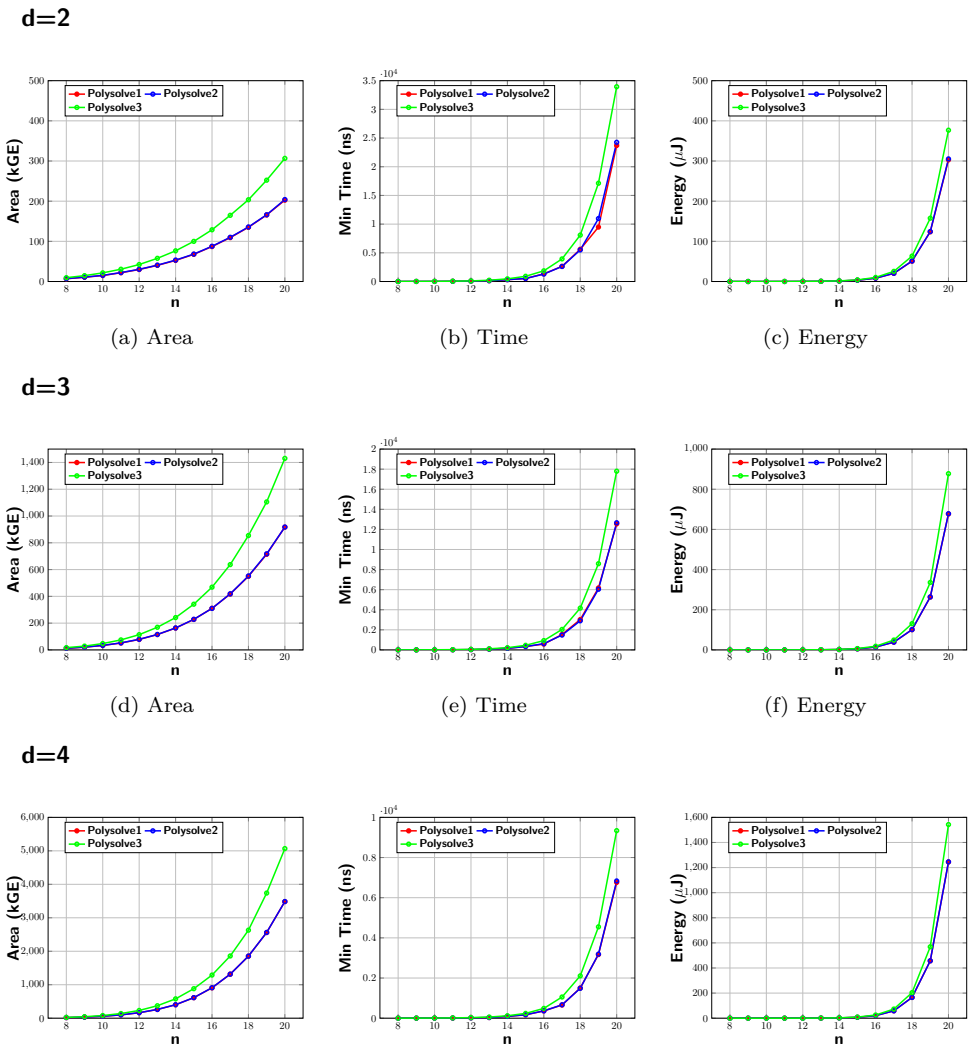


Figure 9: Synthesis results for **Polysolve1**/**Polysolve2**/**Polysolve3** circuits

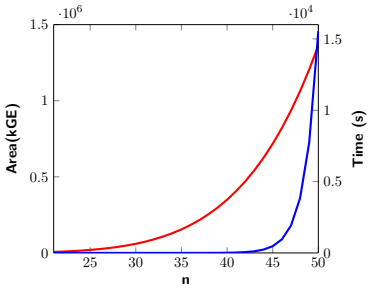


Figure 10: Estimates for circuit area and time for larger n

4.5 Estimates for larger n

For larger values of $n > 20$, simulation takes upwards of 24 hours for each instance of **Polysolve3**, and so we report only estimates of the silicon area and total physical time using projections from the gathered data. For $n \in [21, 50]$ we can estimate the circuit area using the formula $m \cdot \sum_{i=1}^{n-d} \binom{n-i}{\downarrow d}$. The unit here is the area of one scan flip-flop and one 2:1 mux. In the Nangate 15nm OCL, this would amount to around 11.5 GE of silicon area. Further, extrapolating from Table 3, we can estimate (albeit crudely) that the delay of one **AND, OR, XOR** gate to be around 20ps. From this we could estimate the total physical time for finding all the solutions of the system using the expression $2 \cdot log_2(n-d) \cdot 2^{n-d} \cdot 20$ ps. The estimate is shown in Fig 10. We can project that to solve a system in $n = 50$ unknowns of degree 4, the circuit should take around 15000 seconds.

5 Conclusion

In this paper, we propose, design and evaluate hardware architectures to perform Möbius Transform of Boolean functions using only polynomial amount of silicon area. In a nutshell, this is a serialized implementation of the basic transform and uses around 2^{n-d} clock cycles to generate the entire truth table of the Boolean function. The immediate application of the circuits is to use it to solve an underlying system of low degree equations over GF(2), a problem which occurs in many cryptanalytic attacks on real world cryptosystems. We further describe architectures for such equation solvers which keeps the critical path of the circuit to a minimum. The main conclusion of the paper is the demonstration that a system of m Boolean equations in n variables and algebraic degree upto d can be solved using silicon area proportional to $m \cdot n^{d+1}$ gates and using physical time proportional to $2^{n-d} \cdot \log_2(n-d)$. We demonstrate this using the Nangate 15nm Open Cell Library, for which we report the simulated results in the final part of this paper.

Appendix A: Fast generation of the AM graph

The following algorithm generates the adjacency matrix for the AM graph in polynomial time.

Algorithm 2: Generation of AM

Generate $AM\ (n, d)$
Input: n: Number of variables, d: Algebraic degree
Output: The adjacency matrix AM of size $\binom{n}{\downarrow d} \times (n-d)$

for $s \leftarrow$ *the k-th string in* $H(n,d)$ **do**
  Compute $\alpha := \chi_{n,d}(s) = \binom{i_0}{\downarrow d} + \binom{i_1}{\downarrow d-1} + \cdots$ `/*Assuming` $s = 2^{i_0} + 2^{i_1} + \cdots$ `* /`
  for $\ell \leftarrow 1 \rightarrow n-d$ **do**
    Compute $s' \leftarrow s \oplus e_\ell$
    if $hw(s') \leq d$ **then**
      Compute $\beta := \chi_{n,d}(s')$
      Assign $AM[\alpha, \ell] \leftarrow \beta$
    end
    else
      Assign $AM[\alpha, \ell] \leftarrow 0$
    end
  end
end

Appendix B: Detailed Synthesis Results

Table 2: Synthesis results for the **Expmob1**/**Expmob2** circuits.

n	Circuit	Area (μm^2)	Area (kGE)	T_{cr} (ps)	T_{min} (ns)	Power (μW)	Energy (pJ)
6	**Expmob1**	95.600	486.247	91.05	91.05	6.538	0.654
	Expmob2	157.041	798.750	38.53	231.18	6.410	3.846
7	**Expmob1**	233.128	1185.750	118.05	118.05	18.073	1.807
	Expmob2	300.958	1530.750	41.29	289.03	12.106	8.474
8	**Expmob1**	572.473	2911.750	134.02	134.02	50.918	5.092
	Expmob2	573.702	2918.000	46.63	373.04	23.521	18.817
9	**Expmob1**	1333.936	6784.750	169.12	169.12	128.900	12.890
	Expmob2	1138.754	5792.000	48.78	439.02	46.407	41.766
10	**Expmob1**	3227.910	16418.000	208.57	208.57	357.500	35.750
	Expmob2	2255.831	11473.750	52.82	528.20	91.662	91.662
11	**Expmob1**	7855.473	39955.000	257.55	257.55	993.300	99.330
	Expmob2	4486.152	22817.749	56.12	617.32	180.800	198.880
12	**Expmob1**	18784.420	95542.500	304.49	304.49	2706.000	270.600
	Expmob2	9046.868	46014.749	60.50	726.00	364.700	437.640
13	**Expmob1**	46181.007	234888.750	453.37	453.37	6494.300	649.430
	Expmob2	17860.706	90844.247	60.05	780.65	719.100	934.830
14	**Expmob1**	110026.212	559622.251	509.58	509.58	11650.400	1165.040
	Expmob2	35611.803	181130.994	64.77	906.78	1441.000	2017.400

Table 3: Synthesis results for $d = 4$ for the **Polysolve1**/**Polysolve2**/**Polysolve3** circuits.

n	Circuit	Area (μm^2)	Area (kGE)	T_{cr} (ps)	T_{min} (ns)	Power (mW)	Energy (μJ)
8	**Polysolve1**	3258.93	16.576	49.18	0.98	0.105	0.002
	Polysolve2	3464.04	17.619	48.30	1.06	0.115	0.003
	Polysolve3	4600.23	23.398	70.25	1.97	0.144	0.004
9	**Polysolve1**	6196.15	31.515	61.62	2.28	0.192	0.007
	Polysolve2	6467.71	32.896	62.05	2.42	0.202	0.008
	Polysolve3	8649.92	43.996	86.10	3.96	0.268	0.012
10	**Polysolve1**	11341.33	57.685	69.68	4.88	0.343	0.024
	Polysolve2	11504.76	58.516	69.53	5.01	0.348	0.025
	Polysolve3	15656.24	79.632	91.00	7.28	0.441	0.035
11	**Polysolve1**	19470.93	99.034	55.81	7.53	0.571	0.077
	Polysolve2	19660.06	99.996	61.50	8.43	0.577	0.079
	Polysolve3	27468.99	139.714	99.33	14.50	0.750	0.109
12	**Polysolve1**	32061.31	163.072	63.52	16.77	0.922	0.243
	Polysolve2	32403.60	164.813	62.42	16.60	0.938	0.250
	Polysolve3	45470.61	231.275	102.78	28.37	1.159	0.320
13	**Polysolve1**	51151.40	260.169	65.06	33.90	1.465	0.763
	Polysolve2	51410.19	261.486	63.93	33.44	1.477	0.773
	Polysolve3	73233.63	372.485	107.27	57.28	1.848	0.987
14	**Polysolve1**	78857.89	401.092	86.48	89.42	2.234	2.310
	Polysolve2	79188.69	402.774	86.68	89.80	2.248	2.328
	Polysolve3	113659.87	578.104	115.28	120.81	2.800	2.935
15	**Polysolve1**	120540.26	613.099	81.59	167.99	3.383	6.966
	Polysolve2	120752.11	614.177	84.58	174.32	3.386	6.978
	Polysolve3	173318.21	881.542	115.42	239.38	4.180	8.670
16	**Polysolve1**	178275.13	906.754	88.38	363.07	4.972	20.426
	Polysolve2	179029.08	910.589	88.52	363.82	4.992	20.515
	Polysolve3	253387.40	1288.795	118.22	487.54	6.143	25.332
17	**Polysolve1**	258573.43	1315.172	79.99	656.32	7.178	58.891
	Polysolve2	258640.96	1315.516	81.29	667.15	7.182	58.945
	Polysolve3	365524.15	1859.152	128.32	1055.05	8.852	72.780
18	**Polysolve1**	364296.97	1852.910	91.64	1502.71	10.083	165.339
	Polysolve2	364672.59	1854.821	90.74	1488.14	10.096	165.574
	Polysolve3	517131.23	2630.265	128.36	2107.16	12.408	203.688
19	**Polysolve1**	503285.79	2559.844	96.69	3169.79	13.938	456.920
	Polysolve2	503566.85	2561.273	97.32	3190.64	13.946	457.203
	Polysolve3	735066.48	3738.741	138.64	4547.67	17.344	568.918
20	**Polysolve1**	684938.51	3483.777	103.19	6764.31	18.985	1244.511
	Polysolve2	685280.90	3485.519	104.23	6832.69	18.998	1245.382
	Polysolve3	996337.12	5067.633	142.41	9338.11	23.525	1542.562

Table 4: Synthesis results for $d = 3$ for the **Polysolve1**/**Polysolve2**/**Polysolve3** circuits.

n	Circuit	Area (μm^2)	Area (kGE)	T_{cr} (ps)	T_{min} (ns)	Power (mW)	Energy (μJ)
8	**Polysolve1**	2321.06	11.805	47.33	1.75	0.072	0.003
	Polysolve2	2425.06	12.334	46.33	1.81	0.076	0.003
	Polysolve3	3322.92	16.901	66.15	2.98	0.098	0.004
9	**Polysolve1**	4002.74	20.359	54.33	3.80	0.122	0.009
	Polysolve2	4141.74	21.066	53.33	3.84	0.127	0.009
	Polysolve3	5576.69	28.364	78.26	6.18	0.161	0.013
10	**Polysolve1**	6522.57	33.175	52.49	7.09	0.197	0.027
	Polysolve2	6637.78	33.761	53.58	7.34	0.202	0.027
	Polysolve3	9329.05	47.450	87.42	12.68	0.254	0.037
11	**Polysolve1**	10140.60	51.578	59.87	15.81	0.297	0.078
	Polysolve2	10317.35	52.477	57.69	15.35	0.303	0.080
	Polysolve3	14702.40	74.780	92.76	25.51	0.384	0.106
12	**Polysolve1**	15345.20	78.050	58.69	30.58	0.447	0.233
	Polysolve2	15575.68	79.222	54.43	28.47	0.455	0.237
	Polysolve3	22380.08	113.831	95.53	50.92	0.573	0.305
13	**Polysolve1**	22527.00	114.578	73.69	76.20	0.647	0.669
	Polysolve2	22672.54	115.318	75.48	78.20	0.654	0.676
	Polysolve3	33268.28	169.211	105.02	109.96	0.818	0.856
14	**Polysolve1**	32087.26	163.204	77.76	160.11	0.917	1.888
	Polysolve2	32234.57	163.953	77.76	160.26	0.924	1.903
	Polysolve3	47457.73	241.382	105.22	218.12	1.167	2.420
15	**Polysolve1**	44728.37	227.500	83.39	342.57	1.281	5.260
	Polysolve2	44893.17	228.338	80.84	332.25	1.286	5.282
	Polysolve3	67116.07	341.370	110.99	457.61	1.637	6.751
16	**Polysolve1**	61031.00	310.420	70.43	577.88	1.734	14.227
	Polysolve2	60967.99	310.099	76.10	624.55	1.740	14.273
	Polysolve3	92120.92	468.551	112.82	927.49	2.218	18.233
17	**Polysolve1**	82029.48	417.223	93.62	1535.18	2.352	38.565
	Polysolve2	82390.69	419.061	90.15	1478.46	2.357	38.648
	Polysolve3	125284.51	637.230	124.26	2039.73	2.991	49.092
18	**Polysolve1**	108061.26	549.628	92.24	3023.90	3.078	100.893
	Polysolve2	108555.92	552.144	88.42	2898.85	3.085	101.122
	Polysolve3	167941.62	854.195	126.92	4163.10	3.984	130.692
19	**Polysolve1**	140420.53	714.216	94.08	6167.13	4.013	263.060
	Polysolve2	141072.87	717.534	92.17	6042.11	4.025	263.827
	Polysolve3	217305.90	1105.275	130.95	8586.52	5.123	335.933
20	**Polysolve1**	180307.72	917.092	95.78	12555.70	5.165	677.114
	Polysolve2	180399.19	917.558	96.59	12662.08	5.173	678.137
	Polysolve3	281244.79	1430.485	135.74	17796.74	6.693	877.552

Table 5: Synthesis results for $d = 2$ for the **Polysolve1**/**Polysolve2**/**Polysolve3** circuits.

n	Circuit	Area		T_{cr}	T_{min}	Power	Energy
		(μm^2)	(kGE)	(ps)	(ns)	(mW)	(μJ)
8	**Polysolve1**	1341.95	6.825	54.60	3.82	0.043	0.003
	Polysolve2	1401.67	7.129	54.49	3.92	0.045	0.003
	Polysolve3	1879.72	9.561	75.43	5.88	0.055	0.004
9	**Polysolve1**	2082.42	10.592	61.78	8.34	0.064	0.009
	Polysolve2	2152.02	10.946	63.38	8.68	0.066	0.009
	Polysolve3	2870.38	14.599	87.10	12.54	0.081	0.012
10	**Polysolve1**	2933.05	14.918	56.44	14.90	0.089	0.023
	Polysolve2	3006.73	15.293	53.45	14.22	0.092	0.024
	Polysolve3	4226.88	21.499	92.16	25.25	0.111	0.030
11	**Polysolve1**	4289.54	21.818	64.94	33.83	0.128	0.067
	Polysolve2	4370.01	22.227	63.42	33.17	0.130	0.068
	Polysolve3	6012.62	30.582	91.18	48.51	0.158	0.084
12	**Polysolve1**	5861.33	29.812	68.97	71.31	0.173	0.178
	Polysolve2	5970.94	30.370	68.56	71.03	0.177	0.183
	Polysolve3	8309.74	42.265	98.14	102.65	0.209	0.219
13	**Polysolve1**	7895.63	40.159	68.50	141.04	0.230	0.474
	Polysolve2	8004.50	40.713	67.87	139.88	0.234	0.482
	Polysolve3	11349.20	57.725	106.39	220.44	0.293	0.606
14	**Polysolve1**	10317.59	52.478	77.81	319.64	0.302	1.239
	Polysolve2	10469.72	53.252	79.29	325.88	0.306	1.258
	Polysolve3	15016.72	76.379	111.85	461.05	0.373	1.536
15	**Polysolve1**	13298.47	67.639	63.50	521.02	0.384	3.148
	Polysolve2	13419.04	68.253	63.81	523.69	0.388	3.183
	Polysolve3	19643.69	99.913	110.51	908.39	0.481	3.952
16	**Polysolve1**	17107.40	87.013	81.31	1333.32	0.500	8.194
	Polysolve2	17246.94	87.722	79.22	1299.21	0.505	8.285
	Polysolve3	25323.50	128.802	113.30	1859.71	0.620	10.170
17	**Polysolve1**	21478.05	109.243	80.49	2638.70	0.617	20.221
	Polysolve2	21642.51	110.079	79.91	2619.85	0.624	20.441
	Polysolve3	32405.37	164.822	120.03	3936.98	0.770	25.259
18	**Polysolve1**	26530.14	134.939	85.62	5612.56	0.774	50.737
	Polysolve2	26684.96	135.727	83.92	5501.29	0.779	51.067
	Polysolve3	40015.18	203.528	123.36	8088.72	0.965	63.301
19	**Polysolve1**	32567.23	165.645	72.36	9485.60	0.944	123.722
	Polysolve2	32660.32	166.119	83.69	10971.01	0.949	124.340
	Polysolve3	49603.02	252.294	130.64	17127.95	1.202	157.592
20	**Polysolve1**	39816.95	202.519	90.44	23709.93	1.155	302.902
	Polysolve2	40122.48	204.073	92.54	24260.66	1.164	305.185
	Polysolve3	60267.33	306.535	129.47	33944.70	1.437	376.703

Appendix C: Proof of Equation (2)

We need to prove the following:

$$S(n,d) = \sum_{i=0}^{n-d} \binom{n-i}{\downarrow d} \in O(n^{d+1}).$$

To prove this we make use of the hockey-stick identity [Jon96] which states that $\sum_{m=d}^{n} \binom{m}{d} = \binom{n+1}{d+1}$. Note that expanding out $S(n,d)$ we get

$$S(n,d) = \begin{array}{ccccccc} \binom{n}{d} & + & \binom{n}{d-1} & + & \cdots & + & \binom{n}{0} & + \\ \binom{n-1}{d} & + & \binom{n-1}{d-1} & + & \cdots & + & \binom{n-1}{0} & + \\ \binom{n-2}{d} & + & \binom{n-2}{d-1} & + & \cdots & + & \binom{n-2}{0} & + \\ \vdots & & \vdots & & \ddots & & \vdots & \\ \binom{d}{d} & + & \binom{d}{d-1} & + & \cdots & + & \binom{d}{0} & \end{array}$$

Applying the hockey-stick identity on each column we get

$$S(n,d) < \binom{n+1}{d+1} + \binom{n+1}{d} + \cdots + \binom{n+1}{1}$$

Using mathematical induction it is easy to prove the hypothesis $\mathcal{P}(d) : \sum_{i=0}^{d} \binom{n}{i} < n^d$, for all $d \geq 2,\ n > d$. The base case for $d = 2$, amounts to $n(n-1)/2+n+1 < n^2 \Rightarrow n^2 > n+2$, which holds for all $n > 2$. Taking $\mathcal{P}(d)$ to be true we have

$$\begin{aligned} \mathcal{P}(d+1) : \sum_{i=0}^{d+1} \binom{n}{i} &< n^d + \binom{n}{d+1} \\ &< n^d + \frac{n^{d+1}}{(d+1)!} = n^d \left(1 + \frac{n}{(d+1)!}\right) < n^{d+1} \end{aligned}$$

Therefore we have $S(n,d) < (n+1)^{d+1}$, from which we can conclude it is $O(n^{d+1})$.

References

[ARS+15] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, pages 430–454, 2015.

[Bar09] Gregory Bard, editor. *Algebraic Cryptanalysis*. Springer, 2009.

[BBR15] Subhadeep Banik, Andrey Bogdanov, and Francesco Regazzoni. Exploring energy efficiency of lightweight block ciphers. In Orr Dunkelman and Liam Keliher, editors, *Selected Areas in Cryptography - SAC 2015 - 22nd International Conference, Sackville, NB, Canada, August 12-14, 2015, Revised Selected Papers*, volume 9566 of *Lecture Notes in Computer Science*, pages 178–194. Springer, 2015.

[BCC+10] Charles Bouillaguet, Hsieh-Chung Chen, Chen-Mou Cheng, Tung Chou, Ruben Niederhagen, Adi Shamir, and Bo-Yin Yang. Fast exhaustive search for polynomial systems in F_2. In Stefan Mangard and François-Xavier Standaert,

editors, *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *Lecture Notes in Computer Science*, pages 203–218. Springer, 2010.

[BCC+13] Charles Bouillaguet, Chen-Mou Cheng, Tung Chou, Ruben Niederhagen, and Bo-Yin Yang. Fast exhaustive search for quadratic systems in F_2 on FPGAs. In Tanja Lange, Kristin E. Lauter, and Petr Lisonek, editors, *Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers*, volume 8282 of *Lecture Notes in Computer Science*, pages 205–222. Springer, 2013.

[BMP+06] Andrey Bogdanov, M. C. Mertens, Christof Paar, Jan Pelzl, and Andy Rupp. A parallel hardware architecture for fast gaussian elimination over GF(2). In *14th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2006), 24-26 April 2006, Napa, CA, USA, Proceedings*, pages 237–248. IEEE Computer Society, 2006.

[CB07] Nicolas T. Courtois and Gregory V. Bard. Algebraic cryptanalysis of the data encryption standard. In Steven D. Galbraith, editor, *Cryptography and Coding, 11th IMA International Conference, Cirencester, UK, December 18-20, 2007, Proceedings*, volume 4887 of *Lecture Notes in Computer Science*, pages 152–169. Springer, 2007.

[CKPS00] Nicolas T. Courtois, Alexander Klimov, Jacques Patarin, and Adi Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In Bart Preneel, editor, *Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding*, volume 1807 of *Lecture Notes in Computer Science*, pages 392–407. Springer, 2000.

[CM03] Nicolas T. Courtois and Willi Meier. Algebraic attacks on stream ciphers with linear feedback. In Eli Biham, editor, *Advances in Cryptology - EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, May 4-8, 2003, Proceedings*, volume 2656 of *Lecture Notes in Computer Science*, pages 345–359. Springer, 2003.

[Cou02] Nicolas T. Courtois. Higher order correlation attacks, XL algorithm and cryptanalysis of toyocrypt. In Pil Joong Lee and Chae Hoon Lim, editors, *Information Security and Cryptology - ICISC 2002, 5th International Conference Seoul, Korea, November 28-29, 2002, Revised Papers*, volume 2587 of *Lecture Notes in Computer Science*, pages 182–199. Springer, 2002.

[Din21] Itai Dinur. Cryptanalytic applications of the polynomial method for solving multivariate equation systems over GF(2). In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part I*, volume 12696 of *Lecture Notes in Computer Science*, pages 374–403. Springer, 2021.

[Jon96] CH Jones. Generalized hockey stick identities and n-dimensional block walking. *Fibonacci Quarterly*, 34(3):280–288, 1996.

[KDH+12] Stéphanie Kerckhof, François Durvaux, Cédric Hocquet, David Bol, and François-Xavier Standaert. Towards green cryptography: A comparison of

lightweight ciphers from the energy viewpoint. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, volume 7428 of *Lecture Notes in Computer Science*, pages 390–407. Springer, 2012.

[KPG99] Aviad Kipnis, Jacques Patarin, and Louis Goubin. Unbalanced oil and vinegar signature schemes. In Jacques Stern, editor, *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, volume 1592 of *Lecture Notes in Computer Science*, pages 206–222. Springer, 1999.

[MMR+15] Mayler Martins, Jody Maick Matos, Renato P. Ribas, André Reis, Guilherme Schlinker, Lucio Rech, and Jens Michelsen. Open cell library in 15nm freepdk technology. In *Proceedings of the 2015 Symposium on International Symposium on Physical Design*, ISPD '15, page 171–178, New York, NY, USA, 2015. Association for Computing Machinery.