A Note on Masking Generic Boolean Functions

Lauren De Meyer[1], Felix Wegener[2] and Amir Moradi[2]

[1] imec - COSIC, KU Leuven, Belgium
lauren.demeyer@esat.kuleuven.be
[2] Ruhr-University Bochum, Germany, Horst Görtz Institute for IT Security
firstname.lastname@rub.de

Abstract. Masking is a popular countermeasure to protect cryptographic implementations against side-channel attacks (SCA). In the literature, a myriad of proposals of masking schemes can be found. They are typically defined by a masked multiplication, since this can serve as a basic building block for any nonlinear algorithm. However, when masking generic Boolean functions of algebraic degree t, it is very inefficient to construct the implementation from masked multiplications only. Further, it is not immediately clear from the description of a masked multiplication, how to efficiently implement a masked Boolean function.
In this work, we fill this gap in the literature with a detailed description and investigation of a generic masking methodology for Boolean functions of any degree t at any security order d.

Keywords: SCA · DPA · Threshold Implementations · $d+1$ Masking · Hamming Graph · Graph Colouring

1 Introduction

Since the seminal works on Side-Channel Attacks (SCA) and more particularly Differential Power Analysis (DPA) by Kocher *et al.* [10, 11], masking has emerged as one of the most popular countermeasures. The idea is to split each sensitive variable x into multiple shares. When the splitting operation is an Exclusive OR (XOR), we refer to it as a Boolean masking. Many Boolean masking schemes have been proposed over the years, to name a few: ISW [9], TI [12], CMS [13], DOM [7]. It is well known that the non-linear parts of a circuit grow exponentially with the masking order, while linear operations can simply be duplicated and performed on each share independently, *i.e.* a linear increase in the area. As such, these works typically describe only a masked multiplication, *i.e.* a monomial of algebraic degree 2 ($x \cdot y$). In order to protect against d^{th}-order DPA, the minimal number of shares to split x and y into is $d+1$. The number of expansion shares of a multiplication is then $(d+1)^2$. More generally, it is known that a monomial of higher degree t expands to $(d+1)^t$ intermediate shares. However, the efficient masking of a generic Boolean functions, consisting of multiple monomials of algebraic degree up to t, does not trivially follow from the description of a single multiplication. Only a few examples can be found in the literature. Ueno *et al.* [14, 15] and Bozilov *et al.* [3] present examples of masked implementations for 4×4 Boolean functions appearing in respectively the AES and Prince ciphers with the minimal number of expansion shares $(d+1)^t$. However, it is not always possible to implement the entire Boolean function with the minimal number of intermediate shares $(d+1)^t$. Bozilov *et al.* [3] showed that a sharing with minimal number of intermediate shares $(d+1)^t$ exists for any t-degree Boolean function with $t+1$ input variables. In this work, we refine and prove this statement and introduce a generic methodology for masking any degree-t function, even with more than $t+1$ variables. In

Section 3, we investigate what properties allow a generic Boolean function to be shared with the minimal number of shares $(d+1)^t$. In Section 4, we present an algorithm which allows one to share any Boolean function with $s \times (d+1)^t$ shares with s minimal.

2 Preliminaries

2.1 Boolean Algebra

We define $(\mathrm{GF}(2), +, \cdot)$ as the field with two elements Zero and One. We denote the n-dimensional vector space defined over this field by $\mathrm{GF}(2)^n$. Its elements can be represented by n-bit numbers and added by bit-wise XOR. In contrast, the Galois Field $\mathrm{GF}(2^n)$ contains an additional field multiplication operation. It is well known that $\mathrm{GF}(2)^n$ and $\mathrm{GF}(2^n)$ are isomorphic.

A Boolean function F is defined as $F : \mathrm{GF}(2)^n \to \mathrm{GF}(2)$, while we call $G : \mathrm{GF}(2)^n \to \mathrm{GF}(2)^n$ a vectorial Boolean function. A (vectorial) Boolean function can be represented as a look-up table, which is a list of all output values for each of the 2^n input combinations. Alternatively, each Boolean function can be described by a unique representation - so-called normal form. Most notably the Algebraic Normal Form (ANF) is the unique representation of a Boolean function as a sum of monomials. In this work, we designate by $m \in \mathrm{GF}(2^n)$ the monomial $x_0^{m_0} x_1^{m_1} \dots x_{n-1}^{m_{n-1}}$ where $(m_0, m_1, \dots, m_{n-1})$ is the bitvector of m. The monomial's algebraic degree is simply its hamming weight: $\deg(m) = \mathrm{hw}(m)$. We can then write the ANF of any Boolean function F as

$$F(x) = \bigoplus_{m \in \mathrm{GF}(2^n)} a_m x_0^{m_0} x_1^{m_1} \dots x_{n-1}^{m_{n-1}}$$

The algebraic degree of F is the largest number of inputs occurring in a monomial with a non-zero coefficient:

$$\deg(F) = \max_{m \in \mathrm{GF}(2^n), a_m \neq 0} \mathrm{hw}(m)$$

2.2 Boolean Masking in Hardware

We denote the s_i-sharing of a secret variable x as $\boldsymbol{x} = (x_0, \dots, x_{s_i-1})$ and similarly an s_o-sharing of a Boolean function $F(x)$ as $\boldsymbol{F} = (F_0, \dots, F_{s_o-1})$. Each component function F_i computes one share y_i of the output $y = F(x)$. A correctness property should hold for any Boolean masking:

$$x = \bigoplus_{0 \leq j < s_i} x_j \Leftrightarrow F(x) = \bigoplus_{0 \leq j < s_o} F_j(\boldsymbol{x})$$

We define $\mathcal{S}(x)$ as the set of all correct sharings of the value x. Creating a secure masking of cryptographic algorithms in hardware is especially challenging due to glitches. Despite this major challenge, Nikova *et al.* [12] introduced a provably secure scheme against first-order SCA attacks in the presence of glitches, named Threshold Implementation (TI). A key concept of TI is the non-completeness property which we recall here.

Definition 1 (Non-Completeness)**.** A sharing $\boldsymbol{F}$ is non-complete if any component function F_i is independent of at least one input share.

Apart from non-completeness, the security proof of TI depends on a uniform distribution of the input sharing fed to a shared function $\boldsymbol{F}$. For example, when considering round-based block ciphers, the output of one round serves as the input of the next. Hence, a shared implementation of F needs to maintain this property of uniformity.

Definition 2 (Uniformity)**.** A sharing $\boldsymbol{x}$ of x is uniform, if it is drawn from a uniform probability distribution over $\mathcal{S}(x)$.

We call $\boldsymbol{F}$ a uniform sharing of $F(x)$, if it maps a uniform input sharing $\boldsymbol{x}$ to a uniform output sharing $\boldsymbol{y}$:

$$\exists c : \forall x \in \mathrm{GF}(2)^n, \forall \boldsymbol{x} \in \mathcal{S}(x), \forall \boldsymbol{y} \in \mathcal{S}(F(x)) : Pr(\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{y}) = c.$$

Finding a uniform sharing without using fresh randomness is often tedious [2, 1] and may be impossible. Hence, many masking schemes restore the uniformity by re-masking with fresh randomness. When targeting first-order security, one can re-mask s output shares with $s-1$ shares of randomness as such:

$$(F_0 \oplus r_0,\ F_1 \oplus r_1,\ \ldots,\ F_{s-2} \oplus r_{s-2},\ F_{s-1} \oplus \bigoplus_{0 \le j \le s-2} r_j)$$

Threshold Implementation was initially defined to need $s_i \geq td + 1$ shares with d the security order and t the algebraic degree of the Boolean function F to be masked. The non-completeness definition was extended to the level of individual variables in [13], which allowed the authors to reduce the number of input shares to $s_i = d + 1$, regardless of the algebraic degree. As a result, the number of output shares s_o increases to $(d + 1)^t$. For example, two shared secrets $\boldsymbol{a} = (a_0, a_1)$ and $\boldsymbol{b} = (b_0, b_1)$ can be multiplied into a 4-share $\boldsymbol{c} = (c_0, c_1, c_2, c_3)$ by just computing the cross products.

$$\begin{aligned} c_0 &= a_0 b_0 & c_1 &= a_0 b_1 \\ c_2 &= a_1 b_0 & c_3 &= a_1 b_1 \end{aligned}$$

The number of output shares can be compressed back to $d + 1$ after a refreshing and a register stage. This method was first applied to the AES S-box in [5] and lead to a reduction in area, but an increase in the randomness cost. A similar method for sharing 2-input AND gates with $d+1$ shares is demonstrated by Gross *et al.* in [7, 8]. In particular, they propose to refresh only the cross-domain products $a_i b_j$ for $i \neq j$, resulting in a fresh randomness cost of $\binom{d+1}{2}$ units. In [14], Ueno *et al.* demonstrate a general method to find a $d + 1$-sharing of a non-quadratic function with $d + 1$ input shares in a non-complete way by suggesting a probabilistic heuristic that produces $(d + 1)^n$ output shares in the worst case, where n stands for the number of variables.

Masking Cubic Boolean Functions with $d + 1$ shares. Each cubic monomial abc can be trivially masked with $d + 1$ input shares and $(d + 1)^3$ output shares (one for each crossproduct). For example, a first-order sharing (*i.e.* $d = 1$) of $z = abc$ is given in (1).

$$\begin{aligned} z_0 &= a_0 b_0 c_0, & z_1 &= a_0 b_0 c_1, & z_2 &= a_0 b_1 c_0, & z_3 &= a_0 b_1 c_1, \\ z_4 &= a_1 b_0 c_0, & z_5 &= a_1 b_0 c_1, & z_6 &= a_1 b_1 c_0, & z_7 &= a_1 b_1 c_1 \end{aligned} \tag{1}$$

The result can be compressed back into $d + 1$ shares after a refreshing and register stage. Our refreshing strategy resembles that of Domain Oriented Masking [7] in such a way that we apply the same bit of fresh randomness to cross-share terms and do not re-mask inner-share terms:

$$\begin{aligned} z'_0 &= [z_0]_{reg} \oplus [z_1 \oplus r_0]_{reg} \oplus [z_2 \oplus r_1]_{reg} \oplus [z_3 \oplus r_2]_{reg} \\ z'_1 &= [z_4 \oplus r_2]_{reg} \oplus [z_5 \oplus r_1]_{reg} \oplus [z_6 \oplus r_0]_{reg} \oplus [z_7]_{reg} \end{aligned} \tag{2}$$

Note that every term after refreshing *e.g.* z_0 or $z_1 \oplus r_0$, is stored in a dedicated register before going to the XOR chain which produces z'_0 and z'_1.

The most basic way to mask a more general t-degree function is thus to expand each monomial into $(d + 1)^t$ shares. However, this is wildly inefficient for a Boolean function

which combines many monomials. On the other hand, it is impossible to keep certain monomials together without violating non-completeness. We devise a sharing method that keeps as many monomials as possible together by splitting the function into a *minimum* number of sub-functions. These sub-parts are functions such as for example $z = abc \oplus abd$, for which it is trivial to find a non-complete sharing. For each sub-function, we create independent sharings, each with $(d+1)^t$ output shares, and recombine them during the compression stage.

3 Sharing Matrices

We introduce a matrix notation in which each column represents a variable to be shared and each row represents an output share domain. Output share j only receives share M_{ij} of variable i. For example, the sharing matrix M of the sharing in Equation (1) is

$$M = \begin{array}{c} \begin{matrix} a & b & c \end{matrix} \\ \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \end{array} \begin{matrix} \\ z_0 \\ z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \\ z_6 \\ z_7 \end{matrix} \tag{3}$$

From this matrix, it is clear that a correct and non-complete sharing for the cubic function $z = abc$ exists, since the 2^3 rows of the matrix are unique, *i.e.* each of the 2^3 possible rows occur in the matrix. Moreover, this Sharing matrix implies a correct and non-complete sharing for any function $z = f(a, b, c)$. Note also that each column is balanced, *i.e.* there are an equal number of 0's and 1's. It is also possible to add a fourth column, such that any submatrix of three columns consists of unique rows:

$$M' = \begin{array}{c} \begin{matrix} a & b & c & d \end{matrix} \\ \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \end{array} \begin{matrix} \\ z_0 \\ z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \\ z_6 \\ z_7 \end{matrix} \tag{4}$$

Hence, the matrix M' demonstrates the possibility to find a correct and non-complete sharing with eight output shares for any combination of cubic monomials defined over four variables a, b, c, d. Note that the non-completeness follows from the fact that each output share (row) only receives one share of each input (column) by construction. To generalize this observation, we introduce the following concepts:

Definition 3 (Sharing Vector)**.** We call a vector v of length $(d+1)^t$ with entries $v_i \in \{0, \ldots, d\}$ a (t, d)-*Sharing Vector*, if and only if it is balanced, *i.e.* each entry occurs an equal number of times:

$$\forall \tau \in \{0, \ldots, d\} : \#\{i | v_i = \tau\} = (d+1)^{t-1}$$

Definition 4 (Sharing Matrix)**.** We call a $(d+1)^t \times c$ matrix M with entries $M_{ij} \in \{0, \ldots, d\}$ a *(t, d)-Sharing Matrix*, if and only if every column M_j is a (t, d)-Sharing Vector and if every $(d+1)^t \times t$ sub-matrix of M contains unique rows.

3.1 How to construct Sharing Matrices

The main question in creating masked implementations is thus how to find such a (t, d)-Sharing Matrix. Below, we present both provable theoretical and experimental results:

3.1.1 Exact Results

Lemma 1. *A (t, d)-Sharing Matrix with t columns exists and is unique up to a reordering of rows.*

Proof. A (t, d)-Sharing Matrix has exactly $(d+1)^t$ rows. If the matrix has t columns, then each row is a t-length word with base $d+1$. The existence of such a matrix follows trivially from choosing as its rows all $(d+1)^t$ elements from the set $\{0, \ldots, d\}^t$. The uniqueness follows from the fact that the rows must be unique, hence each of the $(d+1)^t$ elements can occur exactly once. Up to a permutation of the rows, this matrix is thus unique. □

Lemma 1 is equivalent to the fact that it is trivial to mask t-variable functions of degree t (*e.g.* $z = abc$) with $(d+1)^t$ output shares but also functions such as $z = abc + abd$ (since c and d can use the same Sharing Vector).

Lemma 2. *A $(t, 1)$-Sharing Matrix has at most $c = t + 1$ columns.*

Proof. We prove this Lemma by showing that the $t + 1^{\text{th}}$ column M_t exists and is unique. Consider the Sharing Matrix M from Lemma 1 with t columns and 2^t rows. We reorder the rows as in a Gray Code. This means that every two subsequent rows have only one coordinate (or bit) different. Equivalently, since there are t columns, any two subsequent rows have exactly $t - 1$ coordinates in common. Consider for example row i and $i + 1$. We have the following properties:

$$\exists! \bar{j} \text{ s.t.} \qquad M_{i,\bar{j}} \neq M_{i+1,\bar{j}} \tag{5}$$

$$\forall j \in \{0, \ldots, t-1\} \setminus \{\bar{j}\} : \qquad M_{i,j} = M_{i+1,j} \tag{6}$$

Recall that by definition of Sharing Matrix M, any two rows may have at most $t - 1$ coordinates in common. For row i and $i + 1$, these coordinates already occur in the first t columns (22), hence for the last column we must have:

$$M_{i,t} \neq M_{i+1,t}$$

Since this condition holds for every pair of subsequent rows i and $i + 1$, we can only obtain the alternating sequence $\ldots 010101 \ldots$ as the last column M_t. This column is therefore unique up to an inversion of the bits. An example for $t = 3$ is shown below:

$$M = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \xrightarrow{\text{Gray Code}} \begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \end{matrix} \rightarrow M_t = \begin{matrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{matrix} \text{ OR } \begin{matrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{matrix} \tag{7}$$

The example shows clearly that adding both columns to the matrix would violate the Sharing Matrix definition, since a 3-column submatrix including both new columns cannot have unique rows. Hence, the $t+1^{\text{th}}$ column is unique and thus a $(t,1)$-Sharing Matrix has at most $t+1$ columns. Note also that the labels 0/1 in the last column correspond to a partitioning of the rows in the first t columns based on odd or even hamming weight. □

An alternative proof using graph theory is shown in Appendix A.

While the relation between the degree t and the maximum number of columns in a (t,d)-Sharing Matrix is easily described for masking order $d=1$ (cf. Lemma 2), no simple formula can describe the relationship for higher orders. More general $(d+1)$-ary Gray Codes exist, but the proof of Lemma 2 does not result in uniqueness for $d>1$. We therefore construct an algorithmic procedure for finding Sharing Matrices for higher orders. The results are shown in Table 1.

3.1.2 Search procedure with backtracking

We start from the t-column (t,d)-Sharing Matrix from Lemma 1. To extend this matrix with another column M_t, we keep for each column element $M_{i,t}$ a list $\mathcal{L}_{i,t}$ of non-conflicting values $\in \{0,\ldots,d\}$. For each new column, these lists are initialized to all possible values. Without loss of generalization, we set the first element of the column to zero: $M_{0,t}=0$. For every row i with $t-1$ common coordinates, this element then needs to be removed from its list $\mathcal{L}_{i,t}$.

If there is a row r with a list of length 1 ($|\mathcal{L}_{r,t}|=1$), then the unique value in that list is chosen as the value $M_{r,t}$. Again, this value is subsequently removed from all lists $\mathcal{L}_{i,t}$ for which row i has $t-1$ coordinates in common with row r. This process continues until either the column M_t is complete, or until there are only lists of length >1. In the latter case, any element of the list $\mathcal{L}_{i,t}$ can be chosen as the value $M_{i,t}$. The choice is recorded so that it can later be revoked during backtracking. Whenever a value is assigned to a column element, the remaining lists are updated as before. When a column is fully determined, the next column is added in the same way. As soon as an empty list is obtained for one of the column elements, the algorithm backtracks to the last made choice. If for all possible choices empty lists occur, then the maximum number of columns is obtained and the algorithm stops.

A simplified version of the procedure is shown in Algorithm 3 in Appendix B. Note that optimizations are possible for the algorithm, but we leave this for future work since first-order security is the target in this work. According to the proof of Lemma 2, backtracking is not necessary for $d=1$.

Table 1: Maximum Number of Columns in (t,d)-Sharing Matrices

Degree t	**Order** $d=1$	**Order** $d=2$	**Order** $d=3$
2	3	4	5
3	4	4	6
4	5	5	5
5	6	6	6*
6	7	7	7*
7	8	8	8*

* Results of greedy search without backtracking

Table 1 shows that the maximum number of columns does not follow a simple formula for $d>1$. The results in Table 1 without additional indication have been obtained by exhausting all possible choices via backtracking which takes fractions of seconds for $d=1$ and up to several minutes for $d=2$ and multiple hours for the parameters $t=4, d=3$. As

this strategy becomes infeasible with larger matrices, we indicate results of greedy search without backtracking with an asterisk. This choice is made based on the observation that (for smaller parameters), if a solution exists, backtracking was never necessary to find it.

3.2 From Sharing Matrices to Sharings

Now consider a mapping $\rho : \{0, \ldots, n-1\} \to \{0, \ldots, c-1\}$ which assigns any input variable x_i to a single column of a Sharing Matrix. That column holds the Sharing Vector of that variable. For a monomial to be shareable according to those Sharing Vectors, each variable of that monomial must be mapped to a different column. We therefore introduce the concept of *compatibility* between monomials and a mapping ρ.

Definition 5 (Compatible Mappings)**.** A mapping $\rho : \{0, \ldots, n-1\} \to \{0, \ldots, c-1\}$ is compatible with a monomial $x_0^{m_0} x_1^{m_1} \ldots x_{n-1}^{m_{n-1}}$ of degree $hw(m) = t$ if it maps each variable in the monomial to a different Sharing Vector, *i.e.*

$$\forall i \neq j \in \{0, \ldots, n-1\} \text{ s.t. } m_i = m_j = 1 : \rho(i) \neq \rho(j)$$

Lemma 3. *Consider a set of monomials of degree $\leq t$ (of which at least one monomial has degree t) defined over a set of n variables with ANF*

$$\bigoplus_{m \in GF(2^n)} a_m x_0^{m_0} x_1^{m_1} \ldots x_{n-1}^{m_n - 1}$$

and a sharing of each variable x_i into $d+1$ shares. A correct and non-complete sharing of this set of monomials with $(d+1)^t$ output shares exists if and only if a (t,d)-Sharing Matrix can be constructed such that for each variable in the set of monomials, the Sharing Matrix has exactly one column corresponding to its Sharing Vector and such that for each monomial, the (up to) t variables of that monomial have different Sharing Vectors. In other words, there exists a single mapping $\rho : \{0, \ldots, n-1\} \to \{0, \ldots, c-1\}$ that is compatible with each monomial in the ANF:

$$\forall m \in GF(2^n) \text{ s.t. } a_m = 1 : \forall i \neq j \in \{0, \ldots, n-1\} \text{ s.t. } m_i = m_j = 1 : \rho(i) \neq \rho(j)$$

The mapping ρ assigns to each variable x_i column $\rho(i)$ of the Sharing Matrix as Sharing Vector.

The terms with degree lower than t also have to be compatible with the mapping ρ so that their variables are assigned to different Sharing Vectors. However, lower-degree terms naturally do not need to appear in each of the $(d+1)^t$ output shares. Given a monomial of degree $l < t$ and a set of l (t,d)-Sharing Vectors, it is trivial to choose the $(d+1)^l$ output shares for the monomial to appear in.

We note that our Sharing Matrices are very similar to the D_t^n-tables of Bozilov *et al.* [3], who independently demonstrated that any t-degree function with $t+1$ input variables can be shared with the minimal $(d+1)^t$ output shares. However, their work only treats the sharing of t-degree functions with exactly $t+1$ input variables. Since our goal is to find a sharing of t-degree functions with any number of input variables, we consider here the more general case where both the degree t and the number of variables n are unconstrained.

4 Sharing any ANF

Naturally, not any function is compatible with a (t,d)-Sharing Matrix. In what follows, we develop a heuristic method to determine efficient maskings with $d+1$ shares for any degree t-Boolean function starting from its unshared algebraic normal form (ANF). If a

compatibility mapping with a single Sharing Matrix cannot be found, our approach is to split the monomials of the ANF into a number of subgroups, each for which a (t, d)-Sharing Matrix and thus a correct and non-complete sharing exists. If the ANF is split into s subgroups, then the number of intermediate shares before compression is $s \times (d+1)^t$. Our methodology finds the optimal sharing in terms of parameter s. We do not claim optimality in the number of intermediate shares, since the minimum is not necessarily a multiple of $(d+1)^t$.

4.1 Our Heuristic.

We want to minimize the number of parts the ANF should be split into. This is equivalent to restricting the expansion of the number of shares and thus limiting both the required amount of fresh randomness and the number of registers for implementation.

We assume a (t, d)-Sharing Matrix of c columns is known at this point. A procedure for this was described in §3 and Algorithm 3. There are c^n possible mappings ρ to assign one of the c Sharing Vectors to each of n variables. In an initial preprocessing step, we iterate through all possible ρ and determine which t-degree monomials are compatible with it. During this process, we eliminate redundant mappings (*i.e.* with an identical list of compatible monomials) and the mappings without compatible monomials of degree t. Note that up to this point (including for algorithm 3), the specific function to be shared does not need to be known.

The next step is function-specific: We first attempt to find one mapping that can hold all the monomials of the ANF. Its existence would imply that all the monomials in the ANF can be shared using the same Sharing Matrix (see Lemma 3). This is not always possible and even extremely unlikely for ANFs with many monomials. If this first attempt is unsuccessful, we try to find a *split* of the ANF. A *split* is a set of mappings that jointly are compatible with all monomials in the ANF of the Boolean function, *i.e.* it implies a partition of the ANF into separate sets of monomials, each for which a Sharing Matrix exists. In this search, we first give preference to partitions into a minimal number of subfunctions, in order to minimize s such that the entire function expands into only $s \times (d+1)^t$ intermediate shares.

We note that our search is heuristic and we do not claim optimality except in the number of split groups s.

4.2 Implementation Details.

We encode mappings and ANFs which are dependent on n inputs as bitvectors with 2^n entries. An entry in the bitvector at position $m \in \mathrm{GF}(2^n)$ corresponds to one monomial $x_0^{m_0} x_1^{m_1} \dots x_{n-1}^{m_{n-1}}$ of degree $t = \mathrm{hw}(m)$ and prescribes whether this monomial is present in the ANF. Recall the ANF of an n-bit Boolean function F:

$$F(x) = \bigoplus_{m \in \mathrm{GF}(2^n)} a_m x_0^{m_0} x_1^{m_1} \dots x_{n-1}^{m_{n-1}}$$

We thus define the bitvector representations

$$\mathrm{rep}(F) = \sum_m a_m 2^m \qquad \text{and} \qquad \mathrm{rep}(\rho) = \sum_m \alpha_m^\rho 2^m$$

where $\alpha_m^\rho = 1$ if monomial m is compatible with mapping ρ. Consider for example the function $F = x_0x_2x_4 \oplus x_1x_5$:

$$\mathrm{rep}(x_0x_2x_4 \oplus x_1x_5) = \left(2^{2^0+2^2+2^4}\right) + \left(2^{2^1+2^5}\right) = \texttt{0x400200000}$$

Now, we can determine whether for example a set of mappings (ρ_1, ρ_2) specifies a two-split for a Boolean function F as follows. Assuming both are represented as a 2^n-bit vector, we check if the following condition holds:

$$\text{rep}(\rho_1) \mid \text{rep}(\rho_2) \mid \text{rep}(F) = \text{rep}(\rho_1) \mid \text{rep}(\rho_2),$$

where $\mid$ refers to the Boolean OR-operation. The condition evaluates to *true* whenever all monomials of the ANF of F are also compatible monomials with at least one of the mappings ρ_1 or ρ_2.

Algorithm 1 Preprocessing of mappings

Input: n: number of input bits; t: $\deg(F)$; c: number of columns of (t, d)-Sharing Matrix
Output: L: list of mappings; α: compatibility α^{ρ}_{m}
1: $L \leftarrow \{(\rho(0), \ldots, \rho(n-1)) | \rho(i) \in \{0, \ldots, c-1\}\}$
2: **for** $\rho \in L$ **do**
3: **for** $m \in \text{GF}(2^n)$ s.t. $\text{hw}(m) \leq t$ **do**
4: $\alpha^{\rho}_{m} \leftarrow 0$
5: **if** $\rho(i) \neq \rho(j) \forall i \neq j$ s.t. $m_i = m_j = 1$ **then**
6: $\alpha^{\rho}_{m} \leftarrow 1$
7: **end if**
8: **end for**
9: **if** $\exists \hat{\rho} \in L$ s.t. $\text{rep}(\hat{\rho}) = \text{rep}(\rho)$ **or** $\max_{m, \alpha^{\rho}_{m}=1} \text{hw}(m) < t$ **then**
10: $L \leftarrow L \setminus \{\rho\}$
11: **end if**
12: **end for**

The preprocessing step is illustrated in Algorithm 1 and creates a list of mappings L. The list initially contains all c^n possible mappings, *i.e.* all assignments of n variables x_i to one of c Sharing Vectors (1). We iterate over L (2). For each monomial m up to the target degree t (3), we check whether it is compatible with the mapping ρ, *i.e.* whether for any two variables in the monomial m they do not have the same Sharing Vector (5). After all compatible monomials for one mapping ρ have been determined, we check for a duplicate - another mapping $\hat{\rho}$ with an identical list of compatible monomials - and eliminate it. We also check whether the mapping ρ is compatible with at least one monomial of the target degree t and otherwise discard it (9,10). The runtime of the entire preprocessing step is bounded by $\mathcal{O}(2^n \cdot c^n)$.

Algorithm 2 Search for l-split

Input: L: list of mappings; α: compatibility α^{ρ}_{m}; F: target function
Output: S: a list of l-splits
1: $S \leftarrow \emptyset$
2: **for** $(\rho_1, \ldots, \rho_l) \in L^l$ **do**
3: **if** $\text{rep}(\rho_1) \mid \ldots \mid \text{rep}(\rho_l) \mid \text{rep}(F) = \text{rep}(\rho_1) \mid \ldots \mid \text{rep}(\rho_l)$ **then**
4: $S \leftarrow S \cup \{(\rho_1, \ldots, \rho_l)\}$
5: **end if**
6: **end for**

Algorithm 2 demonstrates the search for an l-split of mappings for a specific target function F. Its run-time is $|L|^l = \mathcal{O}(c^{ln})$. In practice, the computation for our first-order secure AES design with the parameters $c = t + 1 = 4$, $l = 2$, $n = 8$ takes $3.08s$ for Algorithm 1 and $5.73s$ for Algorithm 2 on a recent Desktop PC[1].

Conclusion

In this work, we investigated the properties of t-degree Boolean functions that can be masked with the minimal number of expansion shares $(d+1)^t$. We proposed the notion of

[1] Averaged over 100 computations

Sharing Matrices as a way of formalizing whether a non-complete masking for a generic Boolean function exists. We proved that a $(t, 1)$-Sharing Matrix has at most $c = t + 1$ columns, which implies that any t-degree Boolean function with $t + 1$ variables can be masked with $(d + 1)^t$ intermediate shares. An interesting question for future research is whether a formal result for higher orders (cf. Table 1) can be found. We further introduced a methodology for splitting any t-degree Boolean function into s subfunctions with s minimal, such that a (t, d)-Sharing Matrix exists for each subfunction, which can hence be shared with the minimal number of intermediate shares.

The above methodology was applied to significantly optimize the implementations of De Meyer *et al.* [6]. The results are described in the extended version [16]. Since these implementations are FPGA-specific, the subfunctions in Algorithm 2 are chosen to minimize the number of variables they depend on. It is trivial to change this for ASICs. The methodology improved upon the AES implementation of [6] by 21.5% in the number of FPGA Look-up Tables (LUTs), 25.8% in the number of flip-flops and 33.3% in the number of slices.

Acknowledgements

The work described in this paper has been supported in part by the German Federal Ministry of Education and Research BMBF (grant nr. 16KIS0666 SysKit_HW) and the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972. Lauren De Meyer is funded by a PhD fellowship of the Fund for Scientific Research - Flanders (FWO).

References

[1] Tim Beyne and Begül Bilgin. Uniform first-order threshold implementations. In Roberto Avanzi and Howard M. Heys, editors, *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers*, volume 10532 of *Lecture Notes in Computer Science*, pages 79–98. Springer, 2016.

[2] Begül Bilgin, Svetla Nikova, Ventzislav Nikov, Vincent Rijmen, and Georg Stütz. Threshold implementations of all 3x3 and 4x4 S-boxes. 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 76–91. Springer, 2012.

[3] Dusan Bozilov, Miroslav Knezevic, and Ventzislav Nikov. Optimized threshold implementations: Securing cryptographic accelerators for low-energy and low-latency applications. *IACR Cryptology ePrint Archive*, 2018:922, 2018.

[4] Andries Brouwer and Willem Haemers. *Spectra of Graphs*, chapter 12: Distance-Regular Graphs, page 178. Springer New York, 2012.

[5] Thomas De Cnudde, Oscar Reparaz, Begül Bilgin, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Masking AES with d+1 shares in hardware. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, volume 9813 of *Lecture Notes in Computer Science*, pages 194–212. Springer, 2016.

[6] Lauren De Meyer, Amir Moradi, and Felix Wegener. Spin me right round: Rotational symmetry for FPGA-specific AES. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3), 2018.

[7] Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. *IACR Cryptology ePrint Archive*, 2016:486, 2016.

[8] Hannes Groß, Stefan Mangard, and Thomas Korak. An efficient side-channel protected AES implementation with arbitrary protection order. In Helena Handschuh, editor, *Topics in Cryptology - CT-RSA 2017 - The Cryptographers' Track at the RSA Conference 2017, San Francisco, CA, USA, February 14-17, 2017, Proceedings*, volume 10159 of *Lecture Notes in Computer Science*, pages 95–112. Springer, 2017.

[9] Yuval Ishai, Amit Sahai, and David A. Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer, 2003.

[10] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[11] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[12] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *Information and Communications Security, 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006, Proceedings*, volume 4307 of *Lecture Notes in Computer Science*, pages 529–545. Springer, 2006.

[13] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating masking schemes. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 764–783. Springer, 2015.

[14] Rei Ueno, Naofumi Homma, and Takafumi Aoki. A systematic design of tamper-resistant galois-field arithmetic circuits based on threshold implementation with (d + 1) input shares. In *47th IEEE International Symposium on Multiple-Valued Logic, ISMVL 2017, Novi Sad, Serbia, May 22-24, 2017*, pages 136–141. IEEE Computer Society, 2017.

[15] Rei Ueno, Naofumi Homma, and Takafumi Aoki. Toward more efficient DPA-resistant AES hardware architecture based on threshold implementation. In Sylvain Guilley, editor, *Constructive Side-Channel Analysis and Secure Design - 8th International Workshop, COSADE 2017, Paris, France, April 13-14, 2017, Revised Selected Papers*, volume 10348 of *Lecture Notes in Computer Science*, pages 50–64. Springer, 2017.

[16] Felix Wegener, Lauren De Meyer, and Amir Moradi. Spin me right round rotational symmetry for FPGA-specific AES: Extended version. *Journal of Cryptology*, Jan 2020.

A Masking and Graph Colouring

In Section 3, we raised the question of how many columns a (t, d)-Sharing Matrix can have. We can connect this problem to that of finding balanced colourings of a graph.

Graph Colouring. Consider a graph $(\mathcal{V}, \mathcal{E})$ with vertices $\mathcal{V} = \{0, \ldots, d\}^t$ corresponding to the rows of a t-column Sharing Matrix M. In other words, the vertices of $\mathcal{G}$ are words of length t with base $d+1$. There are $(d+1)^t$ vertices in total. Let two vertices in $\mathcal{G}$ be connected by an edge when their labels differ in exactly one coordinate, *i.e.* their Hamming distance is one[2]. Such a graph is called a Hamming graph $H(t, d+1)$. The case $d = 1$ is better known as a Hypercube graph [4]. It automatically follows that each pair of connected vertices $\{v_1, v_2\} \in \mathcal{E}$ have exactly $t-1$ coordinates in common. Recall, that in a (t, d)-Sharing Matrix, no two rows may have t common elements. The problem of finding column $t+1$ is thus equivalent to assigning to each vertex v a label $\mathcal{L}(v) \in \{0, \ldots, d\}$ such that $\forall\{v_1, v_2\} \in \mathcal{E} : \mathcal{L}(v_1) \neq \mathcal{L}(v_2)$. An example of such a labeling for $t = 3$ and $d = 1$ was shown in Eqn 3. Hence, if we can find a valid $(d+1)$-colouring $\mathcal{L}$ of the graph $H(t, d+1)$, then this implies the existence of a (t, d)-Sharing Vector that can be added to the Sharing Matrix M as extra column.

Given this equivalence, we can also provide an alternative proof for Lemma 2:

Proof. We consider the case $d = 1$, *i.e.* the vertices of $H(t, 2)$ are bitvectors of length t and $H(t, 2)$ defines a t-dimensional hypercube. We show the existence and uniqueness of the $t + 1^{\text{st}}$ column by showing the existence and uniqueness of a 2-colouring of the graph. It is well known that all hypercube graphs are bipartite, *i.e.* can be coloured with only two colours. This proves the existence of a $t+1$-column $(t, 1)-$Sharing Matrix for any t. Next, we show the uniqueness of this column by showing that the 2-colouring of a hypercube graph is unique up to an inversion of the colours. Figure 1 depicts two 1-hypercubes ($t = 1$) and shows clearly that a 2-colouring of the vertices is unique up to an inversion of the colours. We refer to the colouring as $\mathcal{L}^t$ and its inverse $\bar{\mathcal{L}}^t$. By definition, they have two properties:

$$\forall\{v_i, v_j\} \in \mathcal{E} : \mathcal{L}^t(v_i) \neq \mathcal{L}^t(v_j) \text{ and } \bar{\mathcal{L}}^t(v_i) \neq \bar{\mathcal{L}}^t(v_j) \tag{8}$$

$$\forall v_i : \mathcal{L}^t(v_i) \neq \bar{\mathcal{L}}^t(v_i) \tag{9}$$

Now, we show by induction that a $t+1$-dimensional hypercube only has a unique colouring $\mathcal{L}^{t+1}$ and its inverse $\bar{\mathcal{L}}^{t+1}$. Consider a t-dimensional hypercube graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, which can only be coloured using $\mathcal{L}^t$ or $\bar{\mathcal{L}}^t$. From this graph, we construct a hypercube graph of dimension $t+1$ with vertices $\mathcal{V}' = \mathcal{V} \times \{0, 1\}$ and edges

$$\mathcal{E}' = \{\{(v_i, 0), (v_i, 1)\}, \forall v_i \in \mathcal{V}\} \cup \{\{(v_i, k), (v_j, k)\}, \forall\{v_i, v_j\} \in \mathcal{E}, k \in \{0, 1\}\}$$

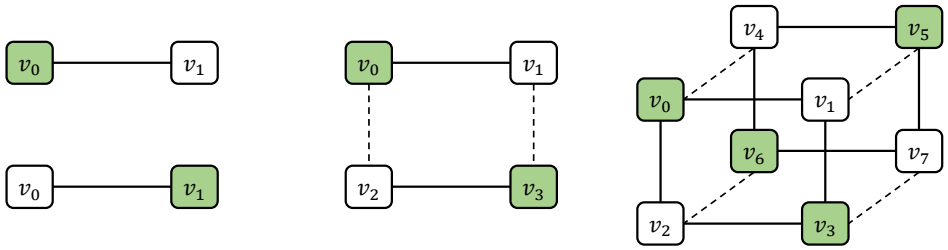
Naturally, a valid colouring $\mathcal{L}^{t+1}$ has to agree with either $\mathcal{L}^t$ or $\bar{\mathcal{L}}^t$ on the subgraphs $\mathcal{G}_0, \mathcal{G}_1$ with nodes $\mathcal{V} \times \{0\}$ and $\mathcal{V} \times \{1\}$, as both are isomorphic to $\mathcal{G}$, hence

$$(\mathcal{L}^{t+1}|_{\mathcal{G}_0},\ \mathcal{L}^{t+1}|_{\mathcal{G}_1}) \in \{(\mathcal{L}^t, \mathcal{L}^t), (\bar{\mathcal{L}}^t, \bar{\mathcal{L}}^t), (\mathcal{L}^t, \bar{\mathcal{L}}^t), (\bar{\mathcal{L}}^t, \mathcal{L}^t)\}$$

Now, edges of the form $\{(v_i, 0), (v_i, 1)\}$ and the colouring property (32) prohibit the choice of equal labellpoings. Hence, only two possibilities for $\mathcal{L}^{t+1}$ remain, which are identical up to an inversion:

$$(\mathcal{L}^{t+1}|_{\mathcal{G}_0},\ \mathcal{L}^{t+1}|_{\mathcal{G}_1}) = (\mathcal{L}^t, \bar{\mathcal{L}}^t),$$
$$(\bar{\mathcal{L}}^{t+1}|_{\mathcal{G}_0},\ \bar{\mathcal{L}}^{t+1}|_{\mathcal{G}_1}) = (\bar{\mathcal{L}}^t, \mathcal{L}^t),$$

[2]Note that we use the general (non-binary) notion of Hamming Weight which counts the number of different coordinates (not bits)



(a) Hypercube, $H(1,2)$ (b) Hypercube, $H(2,2)$ (c) Hypercube, $H(3,2)$

Figure 1: (a) unique 2-colouring of $H(1,2)$ up to inversion (b) extension of $H(1,2)$ to $H(2,2)$ (c) extension of $H(2,2)$ to $H(3,2)$. Dashed lines indicate new edges.

□

As before, the proof cannot be generalized for $d > 1$. In Section 3, we therefore provided specific numbers in Table 1. With this Appendix, we mean to show that the problem of finding non-complete maskings is related to finding the number of $d+1$-colourings of Hamming graphs. To the best of our knowledge, there is not yet a formula to describe this number. We note that not all colourings can be transformed into columns for the Sharing Matrix, since many of them are equivalent up to a renaming of the colours.

B Finding Sharing Matrices

Algorithm 3 Backtracking Procedure for constructing (t, d)-Sharing Matrices

$M \leftarrow$ from Lemma 1
$c \leftarrow t$
while True **do**
    for $i \in \{1, \ldots, (d+1)^t - 1\}$ **do**
        $\mathcal{L}_{i,c} \leftarrow \{0, \ldots, d\}$
    end for
    $\mathcal{L}_{0,c} \leftarrow \{0\}$
    while M_c not completely determined **do**
        if $\exists r$: $\mathcal{L}_{r,c} = \emptyset$ **then**
            Break
        else if $\exists r$: $|\mathcal{L}_{r,c}| = 1$ **then**
            $M_{r,c} \leftarrow \mathcal{L}_{r,c}[0]$
        else
            Pick r, l (& record backtrackpoint)
            $M_{r,c} \leftarrow \mathcal{L}_{r,c}[l]$
        end if
        for $i \in \{1, \ldots, (d+1)^t - 1\} \setminus \{r\}$ **do**
            if $\#\{j : M_{i,j} = M_{r,j}\} = t - 1$ **then**
                $\mathcal{L}_{i,c} \leftarrow \mathcal{L}_{i,c} \setminus \{M_{r,c}\}$
            end if
        end for
    end while
    if M_c not completely determined **then**
        if Backtracking possible **then**
            Jump to last backtrackpoint
        else
            Stop Algorithm
        end if
    end if
    $c \leftarrow c + 1$
end while