Boolean fault-resistant masking and white-boxability of lightweight cryptography

Chloé Gravouil

COLLEGE MATHS, TELECOMS
DOCTORAL INFORMATIQUE, SIGNAL
BRETAGNE SYSTEMES, ELECTRONIQUE

Université de Rennes

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES

ÉCOLE DOCTORALE N° 601
Mathématiques, Télécommunications, Informatique, Signal, Systèmes, Électronique
Spécialité : *Mathématiques et leurs Interactions*

Par
Chloé GRAVOUIL

Masquage Booléen Résistant aux Attaques par Fautes et White-Boxabilité de Primitives Cryptographiques Légères

Unité de recherche : Institut de Recherche Mathématique de Rennes

Rapporteurs avant soutenance :

Guilhem CASTAGNOS, Maître de conférence, Université de Bordeaux
Matthieu RIVAIN, Ingénieur et CEO, Cryptoexperts Paris

Composition du Jury :

Examinateurs : Guilhem CASTAGNOS, Maître de conférence, Université de Bordeaux
Pierre-Alain FOUQUE, Professeur, Université de Rennes
Sihem MESNAGER, Professeure, Université Paris VIII
Matthieu RIVAIN, Ingénieur et CEO, Cryptoexperts Paris
Karine VILLEGAS, Ingénieure, Kudelski IoT Security Lausanne
Dir. de thèse : Sylvain DUQUESNE, Professeur, Université de Rennes

Invité(s) :

Eric PIRET, Ingénieur et Manager, EDSI Cesson-Sévigné

Remerciements

Tout d'abord, je voudrais remercier EDSI et son directeur Jean-Claude Fournier ainsi que le Groupe Kudelski pour m'avoir donné l'opportunité de cette thèse.
Je veux également exprimer ma gratitude envers mon superviseur Eric Piret, mon directeur de thèse Sylvain Duquesne ainsi que les experts du Groupe Kudelski Karine Villegas et Brecht Wyseur pour m'avoir guidée durant ces quatre années, pour leurs conseils et toujours utiles remarques sur mes travaux.

Je tiens également à remercier Pierre-Alain Fouque, Sihem Mesnager et Karine Villegas d'avoir accepté de faire partie de mon jury, et à remercier Matthieu Rivain et Guilhem Castagnos d'avoir accepté les rôles de rapporteurs de cette thèse.

Merci à mes collègues stagiaires Alex Charlès et Tom Beaumont pour leur bonne humeur, nos heures de réflexions communes et nos nombreux débats sur la philosophie des mathématiques et tant d'autres sujets. J'étends ces remerciements à mes collègues d'EDSI et notamment mes collègues de bureau durant cette thèse : Béatrice, Laurent, Pierre, Samuel et Kévin pour leur gentillesse et leur assistance technique.

Pour finir, merci à ma famille pour leur soutien durant ces quatre années.

Contents

Résumé en français

La cryptographie en boîte blanche (white-box cryptography) est le domaine de la cryptographie dédié à la conception d'implémentations de primitives cryptographiques sûres face à un attaquant ayant le contrôle total du dispositif sur lequel est déployée cette implémentation. L'implémentation en boîte blanche d'une primitive est donc une implémentation dont un attaquant ne peut pas retrouver la clé même en ayant son contrôle total, ainsi que le contrôle de la plateforme sur laquelle elle est exécutée.

L'un des enjeux majeurs de sécurité auquel la cryptographie boîte blanche doit répondre est la résistance aux attaques par canaux cachés (*side-channel attacks*). A cette fin, les designers ont pour but d'éliminer ou atténuer au maximum toute dépendance entre les variables de l'implémentation et ses données sensibles, comme les clés secrètes. L'une des contre-mesures classiques pour cela est l'utilisation de schémas de masquage. Néanmoins, les implémentations mettant en œuvre des schémas de masquage sont vulnérables à un autre type d'attaques : les attaques par faute, dans lesquelles un attaquant perturbe intentionnellement le fonctionnement normal de l'implémentation dans le but d'extraire de potentielles informations de cette exécution modifiée.

De plus, au delà d'assurer la sécurité de leurs implémentations dans ce modèle d'attaques, les concepteurs d'implémentations en boîte blanche doivent également prendre en compte leurs coûts tout en optimisant leurs performances. En d'autres termes, la question du compromis entre la sécurité, les coûts et les performances d'une implémentation cryptographique demeure dans le domaine de la cryptographie en boîte blanche. La cryptographie *lightweight* (légère) est le domaine de la cryptographie dédié aux implémentations compatibles avec des dispositifs aux capacités limitées. Ces dispositifs, de par leurs cas d'usages, sont fréquemment vulnérables aux attaques en boîte blanche. Par conséquent, la question de la "white-boxabilité" des algorithmes *lightweight* se pose également.

La contribution de cette thèse est double. Dans la première partie, nous discutons l'adéquation à une implémentation en boîte blanche des dix primitives finalistes du concours de standardisation des algorithmes *lightweight* du NIST. Nous développons par la suite une implémentation en boîte blanche tabularisée de GIFT, la principale sous-fonction cryptographique de GIFT-COFB. Pour finir, nous décrivons une attaque différentielle sur cette construction, et étudions l'adéquation des critères de résistance d'une SBox à cette attaque avec les critères de choix de la SBox de GIFT.

Dans la seconde partie de ce manuscrit, nous décrivons la construction de notre schéma de masquage de l'opération bit-à-bit AND résistant à l'introduction de fautes, et pouvant être implémenté avec uniquement des opérations bit-à-bit. Pour cela, ce schéma utilise un code correcteur d'erreurs, et plus précisément un code correcteur d'erreurs BCH. Nous décrivons également comment les opérations NOT et XOR peuvent être implémentées afin d'être compatibles avec ce schéma de masquage, pour qu'il puisse être appliqué aux implémentations bitslicées de toute primitive cryptographique.

1 White-Boxabilité des Primitives Finalistes du Concours de Standardisation des Algorithmes *Lightweight* du NIST

Cette thèse débute par l'analyse de la "white-boxabilité" des finalistes du concours de standardisation des algorithmes *lightweight* du NIST [SMC+21]. En effet, les primitives cryptographiques *lightweight* sont largement déployées sur des dispositifs aux capacités limitées, qui sont fréquemment sujets aux attaques en boîte blanche en raison de leurs cas d'usages. La question de la résistance de ces candidats à la standardisation face à ces attaques s'est donc posée.

Après analyse des spécifications des dix finalistes, nous avons établi que la primitive la plus adéquate à être implémentée en boîte blanche est GIFT-COFB, et plus précisément sa principale sous-fonction cryptographique GIFT [BPP+17a]. Pour cela, différents critères ont été pris en compte, comme la sécurité de la primitive à la publication d'un état (*state*), ou la répartition des variables secrètes (notamment liées à la clé) tout au long de la primitive [CG22].

L'idée derrière la conception de cette implémentation consiste à encoder les sorties de chaque instance de la SBox de 4 bits de GIFT, afin de construire une nouvelle table de substitution de 8 bits, nommée *TBox*. A cette fin, tout d'abord, une seconde table de substitution de 4 bits est choisie aléatoirement. Les quatre ensembles de 2 bits consécutifs en sortie de la TBox sont obtenus en utilisant 4 encodages de 2 bits différents, avec en entrée un bit de sortie de la SBox de GIFT et un bit de sortie de la seconde table de substitution. Pour que l'implémentation conserve la fonctionnalité de GIFT, les décodages sont appliqués en entrée des TBoxes correspondantes de la ronde suivante, déterminées en fonction de la permutation de 128 bits de GIFT.

Néanmoins, nous démontrons que la connaissance par l'attaquant de la SBox de GIFT implique la possibilité d'une attaque différentielle visant deux rondes consécutives de GIFT afin déterminer la clé utilisée. Pour finir, nous étudions l'existence de potentielles SBoxes de 4 bits vérifiant les propriétés de celle de GIFT tout en empêchant la possibilité de cette attaque.

2 Un Nouveau Schéma de Masquage Résistant Aux Attaques par Fautes

Le *bitslicing* consiste à implémenter une primitive comme un circuit combinatoire en software [MDLM18]. Nombre d'implémentations en boîte blanche sont basées sur des implémentations bitslicées elle-mêmes composées d'opérations bit-à-bit, sur lesquelles sont appliqués des schémas de masquage.

Les schémas de masquage étant par nature des contre-mesures contre les attaques par canaux cachés, nous avons développé un nouveau schéma de masquage composé d'opérations bit-à-bit et résistant aux attaques par faute [Gra23]. Plus précisément, ce schéma peut être appliqué sur des implémentations bitslicées de toutes primitives cryptographiques et corriger de potentielles fautes

sans détériorer ou stopper l'exécution dès leur détection. Ces implémentations vont donc toujours retourner des résultats, qui sont corrects même en cas d'introduction de fautes, ce qui constitue un réel bénéfice dans ce domaine des schémas de masquage résistants aux fautes.

Pour cela, nous utilisons un code correcteur d'erreurs BCH, puisque c'est un code cyclique qui permet donc une facile gestion de la parité du poids de Hamming de ses mots de code, et puisque son processus de correction peut être implémenté avec uniquement des opérations bit-à-bit (AND, OR, XOR, NOT). La valeur binaire de chaque mot de code est cette parité. Ainsi, les mots de code de poids de Hamming pair représentent la valeur 0, et les mots de code de poids de Hamming pair représentent la valeur 1.

2.1 Schémas de Masquage

Dans un corps $\mathbb{K}$, masquer une variable X consiste à construire n_{in} sous-variables x_i nommées *shares*, telles que $X = x_0 \oplus \ldots \oplus x_{n_{in}-1}$. Par construction, chaque ensemble d'au plus $n_{in}-1$ *shares* est indépendant de la variable originale X, c'est-à-dire

$$\begin{gathered}\forall i \in \{0,\ldots,n_{in}-1\}, \forall v \in \mathbb{K}, \forall (v_0,\ldots,v_{i-1},v_{i+1},\ldots,v_{n_{in}-1}) \in \mathbb{K}^{n_{in}-1}, \\ P((X_0,\ldots,X_{i-1},X_{i+1},\ldots,X_{n_{in}-1}) = (v_0,\ldots,v_{i-1},v_{i+1},\ldots,v_{n_{in}-1}) | X = v) = \\ P((X_0,\ldots,X_{i-1},X_{i+1},\ldots,X_{n_{in}-1}) = (v_0,\ldots,v_{i-1},v_{i+1},\ldots,v_{n_{in}-1}))\end{gathered}$$

Les valeurs des *shares* $x_0,\ldots,x_{n_{in}-2}$ sont choisies aléatoirement, puis la dernière *share* $x_{n_{in}-1}$ est calculée telle que $x_{n_{in}-1} = X \oplus x_0 \oplus \ldots \oplus x_{n_{in}-2}$. Comme développé dans [BBP+16], la complexité de l'extraction d'information sur X devient exponentielle en le nombre de *shares* n_{in}.

Pour une fonction f avec n entrées X_i et une sortie Y, un schéma de masquage dans lequel chaque entrée est divisée en n_{in} *shares* $X_{i,j}$ et chaque sortie en n_{out} *shares* Y_j décrit n_{out} sous-fonctions F_j telles que

$$\begin{aligned}
Y_0 &= F_0(X_{0,0},\ldots,X_{0,n_{in}-1},\ldots,X_{n-1,0},\ldots,X_{n-1,n_{in}-1}) \\
Y_1 &= F_1(X_{0,0},\ldots,X_{0,n_{in}-1},\ldots,X_{n-1,0},\ldots,X_{n-1,n_{in}-1}) \\
&\vdots \\
Y_{n_{out}-2} &= F_{n_{out}-2}(X_{0,0},\ldots,X_{0,n_{in}-1},\ldots,X_{n-1,0},\ldots,X_{n-1,n_{in}-1}) \\
Y_{n_{out}-1} &= F_{n_{out}-1}(X_{0,0},\ldots,X_{0,n_{in}-1},\ldots,X_{n-1,0},\ldots,X_{n-1,n_{in}-1})
\end{aligned},$$

avec

$$\begin{cases} X_i = \bigoplus_{j=0}^{n_{in}-1} X_{i,j} \quad \forall i \in \{0,\ldots,n-1\} \\ Y = \sum_{j=0}^{n_{out}-1} Y_j = f(X_0,\ldots,X_{n-1}) \end{cases}$$

2.2 Détermination des Paramètres

Afin de garantir sa sécurité contre les *probing attacks* de premier ordre, nous avons construit notre schéma pour qu'il respecte conjointement trois propriétés, à savoir *correctness*, *non-completeness* et *uniformity* ([NRR06]). Ces propriétés impliquent respectivement que la somme des *shares* en sortie du schéma est bien égale au résultat de la fonction d'origine avec les mêmes entrées, que chaque *share* en sortie ne dépend jamais de tous les *shares* d'une même entrée et que les valeurs possibles des *shares* d'une même sortie sont équiprobables.

Respecter conjointement ces propriétés introduit des contraintes sur le choix des paramètres de notre schéma. C'est pourquoi nous avons construit notre schéma de masquage avec 4 *shares* pour chaque entrée et 4 *shares* en sortie. Cela a également influencé le choix des paramètres du code correcteur d'erreurs BCH mis en oeuvre, à savoir sa longueur et son polynôme générateur. De même, le fait que chaque mot de code représente la parité de son poids de Hamming en valeur binaire implique que le polynôme générateur soit de parité impaire, afin qu'il soit possible de construire des mots de code de parités différentes.

2.3 Design du Schéma de Masquage

Pour la conception de ce nouveau schéma de masquage, nous nous plaçons dans le modèle d'attaque *one-bit flipping fault*, c'est-à-dire que nous considérons qu'un attaquant est capable d'aléatoirement remplacer la valeur d'un bit par son opposée dans l'implémentation. En effet, c'est un modèle d'attaque par faute susceptible d'être utilisé par un attaquant contre une implémentation bitslicée. Nous construisons ce schéma pour qu'il puisse résister à l'introduction d'au plus deux telles fautes dans ses *shares* d'entrée.

Pour pouvoir corriger d'éventuelles fautes dans chaque *share* d'entrée, corriger chaque *share* individuellement nécessiterait 8 corrections. Afin d'éviter autant de corrections et donc d'améliorer les performances du schéma, nous construisons des sous-sommes de *shares* d'entrée. Nous montrons que le nombre minimum de sous-sommes à construire pour qu'une faute détectée par la correction d'une sous-somme soit attribuée à la correcte entrée tout en conservant la propriété de *non-completeness* est 3.

Nous effectuons par la suite diverses multiplications entre les parités des *shares* en entrée, et avec les parités des polynômes aléatoirement choisis. Ces valeurs sont rassemblées dans une matrice sur laquelle sont appliqués différents masques, eux-mêmes mots du code, dans le but de calculer les parités des *shares* de sortie. Pour assurer la pérennité du schéma dans le cas d'exécutions successives dans l'implémentation d'une primitive cryptographique et assurer que les *shares* en sortie n'ont pas subi de faute, nous construisons ces *shares* comme étant des mots du code BCH utilisé. Enfin, afin de respecter la propriété d'*uniformity*, ces *shares* sont construits de manière équiprobable.

Introduction

3 Thesis Introduction

3.1 Security Models and White-Box Cryptography

Cryptographic primitives were first designed so that an attacker having access to only its inputs and corresponding outputs would not be in capacity of retrieving the secret key. This security model has been labeled as *"black-box" security model*, where implementing a cryptographic primitive is supposed to be secure. Hence, any weakness in an implementation can only arise from the design of the primitive itself and potential relations between its inputs and outputs.

Nevertheless, this assumption rapidly turned out to not be realistic. Indeed, implementations are particularly sensitive to attacks labeled as "side-channel attacks". Those types of attacks exploit flaws of the implementation that are correlations between some sensitive data of the algorithm, for instance the secret key bits, and physical data leakage during some executions of this implementation. Those data leakages can be of different types, as for example execution time, electromagnetic emanations, or power consumption of the device executing the implementation [Cad05]. Consequently, side-channel countermeasures aim to eliminate any relation between sensitive data of the primitive and those physical data leakages [PR13]. For instance, in the timing attack case, conditional statements depending on the sensitive data must be avoided. An attacker exploiting those flaws as well as potential weaknesses in the design of the primitive spotted in the black-box security model is considered being an attacker in the *"grey-box" security model*.

Over the course of the last twenty years, the development of devices like Internet of Things (IoT) devices has led to an increasing need of cryptography. Likewise, the expansion of subscription television and then streaming services has brought a need of cryptography to ensure correct and secure Digital Right Managment (DRM). Nevertheless, the open nature of these devices and services constitute an additional threat to the security of these cryptographic primitives. Indeed, a potential attacker can then have total access over the execution platform of the algorithm and its implementation : he can even be the owner of the device. This model of attacker is considered an attacker in the *"white-box" security model*. A primitive being secure in this model imply that it is, first of all, secure in the grey-box model and thus naturally in the black-box model, and that it is not possible to recover the key for an attacker having total access to the implementation and its execution platform.

To protect the key from such attacker, one of the first and most known method that was proposed by Chow *et al.* [CEJvO02] in 2002 consists in tabularizing then encoding the implementation. Their original idea was to turn the AES algorithm into a giant look-up table mapping each possible plaintext to its corresponding ciphertext, consequently avoiding any manipulation of the key. Nevertheless, such a look-up table matching all 128-bit plaintexts to 128-bit ciphertexts would be too heavy and thus unrealistic ($2^{128} * 128 = 2^{135}$ bits). Therefore, the chosen solution was to build a network of encodings look-up tables. Nonetheless, this scheme has been broken many times in the literature

([BGEC04], [MRP13], [LR13]).

Indeed, side-channel attacks inherited from the grey-box attack model are one of the main threats a white-box implementation need to thwart. The Chow *et al.* implementation is notably sensitive to this type of attacks. Among the variety of countermeasures to side-channel attacks, masking is the most developed topic.

3.2 Variable Sharing and Masking Schemes

In a field $\mathbb{K}$, masking a variable X consists in splitting it into n_{in} sub-variables $X_0, \ldots, X_{n_{in}-1}$ of $\mathbb{K}$ named *shares*, such that $X = X_0 \oplus \ldots \oplus X_{n_{in}-1}$. Furthermore, each tuple of at most $(n_{in} - 1)$ variables X_i is independent from X, i.e.

$$\begin{gathered}\forall i \in \{0, \ldots, n_{in} - 1\}, \forall v \in \mathbb{K}, \forall (v_0, \ldots, v_{i-1}, v_{i+1}, \ldots, v_{n_{in}-1}) \in \mathbb{K}^{n_{in}-1}, \\ P((X_0, \ldots, X_{i-1}, X_{i+1}, \ldots, X_{n_{in}-1}) = (v_0, \ldots, v_{i-1}, v_{i+1}, \ldots, v_{n_{in}-1}) | X = v) = \\ P((X_0, \ldots, X_{i-1}, X_{i+1}, \ldots, X_{n_{in}-1}) = (v_0, \ldots, v_{i-1}, v_{i+1}, \ldots, v_{n_{in}-1}))\end{gathered}$$

To that end, the values of the shares $X_0, \ldots, X_{n_{in}-2}$ are chosen uniformly at random in $\mathbb{K}$, and the value of the last share $X_{n_{in}-1}$ is computed so that $X = X_0 \oplus \ldots \oplus X_{n_{in}-1}$. Therefore, if the value of a variable Z is correlated to a sensitive value X then, as each share Z_i separately is independent from Z, they are independent from the sensitive value X. As stated in [BBP+16], the tuple of shares $(Z_i)_{0 \leqslant i \leqslant n_{in}-1}$ still depends on X but, because of the noise, the complexity of the extraction of information then becomes exponential in the number of shares n_{in}. Each set of n_{in} sub-variables X_i such that $X = X_0 \oplus \ldots \oplus X_{n_{in}-1}$ is named a n_{in}-sharing of X.

On a larger scale, *masking schemes* describe how, for a given function f with n inputs X_k and m outputs $Y_k = f_k(X_0, \ldots, X_{n-1})$, the sharings of the outputs of f are built as functions of the sharings of its inputs. As an example, if each input is split into n_{in} shares and each output is split into n_{out} shares, a masking scheme $(F_{i,j})_{0 \leqslant i < m, 0 \leqslant j < n_{out}}$ describes the $n_{out} * m$ sub-functions $F_{i,0}, \ldots, F_{i,n_{out}-1}$ with $0 \leqslant i < m$ such that

$$\begin{aligned}
Y_{i,0} &= F_{i,0}(X_{0,0}, \ldots, X_{0,n_{in}-1}, \ldots, X_{n-1,0}, \ldots, X_{n-1,n_{in}-1}) \\
Y_{i,1} &= F_{i,1}(X_{0,0}, \ldots, X_{0,n_{in}-1}, \ldots, X_{n-1,0}, \ldots, X_{n-1,n_{in}-1}) \\
&\vdots \\
Y_{i,n_{out}-2} &= F_{i,n_{out}-2}(X_{0,0}, \ldots, X_{0,n_{in}-1}, \ldots, X_{n-1,0}, \ldots, X_{n-1,n_{in}-1}) \\
Y_{i,n_{out}-1} &= F_{i,n_{out}-1}(X_{0,0}, \ldots, X_{0,n_{in}-1}, \ldots, X_{n-1,0}, \ldots, X_{n-1,n_{in}-1})
\end{aligned},$$

with

$$\begin{cases} X_k = \sum_{j=0}^{n_{in}-1} X_{k,j} \text{ for all } k \in \{0, \ldots, n-1\} \\ Y_k = \sum_{j=0}^{n_{out}-1} Y_{k,j} = f_k(X_0, \ldots, X_{n-1}) \text{ for all } k \in \{0, \ldots, m-1\} \end{cases}$$

For the most usual case of a function f admitting n inputs and one output, if each input is split into n_{in} shares and the output is split into n_{out} shares, a masking scheme $(F_i)_{0 \leqslant i < n_{out}}$ of f describes the n_{out} sub-functions $F_0, \ldots, F_{n_{out}-1}$ such that

$$\begin{aligned} Y_0 &= F_0(X_{0,0}, \ldots, X_{0,n_{in}-1}, \ldots, X_{n-1,0}, \ldots, X_{n-1,n_{in}-1}) \\ Y_1 &= F_1(X_{0,0}, \ldots, X_{0,n_{in}-1}, \ldots, X_{n-1,0}, \ldots, X_{n-1,n_{in}-1}) \\ &\vdots \\ Y_{n_{out}-2} &= F_{n_{out}-2}(X_{0,0}, \ldots, X_{0,n_{in}-1}, \ldots, X_{n-1,0}, \ldots, X_{n-1,n_{in}-1}) \\ Y_{n_{out}-1} &= F_{n_{out}-1}(X_{0,0}, \ldots, X_{0,n_{in}-1}, \ldots, X_{n-1,0}, \ldots, X_{n-1,n_{in}-1}) \end{aligned},$$

with

$$\begin{cases} X_i = \bigoplus_{j=0}^{n_{in}-1} X_{i,j} \quad \forall i \in \{0, \ldots, n-1\} \\ Y = \sum_{j=0}^{n_{out}-1} Y_j = f(X_0, \ldots, X_{n-1}) \end{cases}$$

Such masking scheme is noted a (n_{in}, n_{out})-masking scheme of the function f, and precisely a n_s-masking scheme if $n_s = n_{in} = n_{out}$.

3.3 Research Questions

The primary purpose of this thesis was to develop improvements to the white-box cryptographic primitives used in the Kudelski Group products, that are mostly symmetric white-box cryptographic primitives. Consequently, I started my PhD researches by constituting a state-of-art of white-box cryptography, and particularly symmetric white-box cryptography. I then chose to focus in the field of bitsliced implementations using masking schemes.

Two of the main threats to cryptographic primitives and notably cryptographic primitives in the white-box attack model are side-channel attacks and fault attacks. Fault attacks consist of disrupting the correct functioning of the cryptographical primitive to observe faulty behaviour of variables depending on sensitive data ([GT04], [Ott05]). Leaked information can then be processed with statistic or analytic methods, thereby disclosing all or part of sensitive data involved in the computation. The masking schemes designed with fault resistance in mind usually adopt the strategy of detecting faults instead of correcting

them. At the detection of a fault, they abort or deteriorate the execution of the implementation. Masking schemes are by design countermeasures against side-channel attacks, therefore I aimed to develop a masking scheme that could combine resistance to side-channel attacks as well as resistance to fault attacks, and that would only be using Boolean operations in order to be applied on Boolean implementations. More precisely, this masking scheme would not only detect faults but correct them, allowing the execution of the implementation to carry on in a correct manner.

With this aim in mind, I introduced error-correcting codes in this new masking scheme, and more precisely BCH error-correcting codes since the corresponding decoding can be performed with only Boolean operations.

On the other hand, one of the main challenges cryptography needs to deal with consists in balancing the security of a cryptographic primitive with its costs and performances. This is notably important when the considered cryptographic primitive is deployed on a device with constrained capacities, like for example the IoT (Internet of Things) products developed by the Kudelski Group. Consequently, performances and costs remain important parameters to take into consideration when designing a white-box secure implementation.

To that end, the company offered an internship to Alex Charlès so we would work in collaboration to evaluate the "white-boxability" of the primitives finalists of the NIST Lightweight Cryptography Standardization Contest. This criterion was not part of the Standardization Contest criteria, therefore the aim of this study was to pick the Lightweight Contest finalist the most suitable to first undergo a primary layer of white-boxing, and then be bitsliced to apply the masking scheme developed during my PhD. We selected GIFT-COFB and more precisely its core cryptographic function GIFT, and developed a tabularized encoding solution to apply to GIFT before bitslicing, based on Chow et al. white-box AES ([CEJvO02]).

3.4 Thesis Overview

In the preamble of this dissertation we will first detail the tools used for the thesis, i.e. the notations and theoretical basis to our study. Then, we will present an overview of the state-of-the-art of white-box cryptography contests and bitsliced masking schemes.

This first part of this dissertation will discuss in section 6 the study of the white-boxability of the NIST Lightweight Cryptography Standardization Contest finalists. We selected GIFT-COFB and built a tabularized white-box implementation of GIFT, the main cryptographic block of this finalist, that is described in section 7. This work has been presented at the NIST Lightweight Cryptography Workshop 2022 [CG22]. Finally, a differential attack on this implementation will be introduced, as well as a potential evolution of the GIFT SBox so that our implementation could resist this attack. This work is described in a paper currently being finalized.

The second part of the dissertation presents in section 9 the rationale behind the design of the new fault resistant masking scheme, before describing this

design in section 10. Finally, section 11 details the application of the scheme to a global implementation of a cryptographic primitive. This masking scheme is presented in [Gra23].

4 Tools for the Thesis

4.1 Notations

This dissertation will use the following notations :

- For a random variable X, we note $P(X)$ its *probability distribution* and H its *entropy* $(H(X) = -\sum_x P(X = x)\log_2(P(X = x)))$.
- For random variables X and Y,
  - We note $H(X|Y) = \sum_x \sum_y P(X = x, Y = y)\log\frac{P(X=x,Y=y)}{P(Y=y)}$ the *conditional entropy* of X given Y.
  - $I(X;Y) = H(X) - H(X|Y)$ is the *mutual information* between X and Y.
- We consider $\mathbb{K}$ a field with $char(\mathbb{K}) = 2$.
- For an array $a \in \mathbb{K}^n$, we note $HW(a)$ the Hamming weight of a, i.e. the number of non-zero elements of a.
- For $a, b \in \mathbb{K}^n$, we note $d_{HW}(a, b) = HW(a - b)$ the Hamming distance between a and b.
- For an array $y = (y_{n-1}, \ldots, y_0) \in \mathbb{K}^n$, we note $y(X) = y_{n-1}X^{n-1} + \ldots + y_1X + y_0 \in \mathbb{K}[X]$ its corresponding polynomial.
- For a 128-bit array $b = (b_0, b_1, ..., b_{127})$, we note $b = B_0B_1...B_{15}$ its decomposition in 16 bytes.
- We note the parity of a polynomial $P \in \mathbb{K}[X]$ to be $HW(P)$ mod 2.
- For a set S, we note $\#S$ its cardinality.
- For $(x_{n-1}, \ldots, x_0) \in \mathbb{F}_2^n$ such that $\sum_{i=0}^{n-1} x_i 2^i = x \in \mathbb{N}$, we note $(x)_n = (x_{n-1}, \ldots, x_0)$.
  - For $(x_{n-1}, \ldots, x_0) \in \mathbb{F}_2^n$ such that $(x_{n-1}, \ldots, x_0) = (x)_n$ and S an SBox with n-bit inputs, we note $S[x_{n-1} \ldots x_0] = S[x]$.
- For $x \in \mathbb{F}_2$, we note $\overline{x} = x \oplus 1$.
  - For $(x_{n-1}, \ldots, x_0) \in \mathbb{F}_2^n$, we note $\overline{x_{n-1} \ldots x_0}$ the array $\overline{x_{n-1}} \ldots \overline{x_0} \in \mathbb{F}_2^n$.
  - Particularly, for $(a, b, c, d) \in \mathbb{F}_2^4$, we note $\overline{abcd}$ the array $\overline{a}\overline{b}\overline{c}\overline{d}$.
- For $v \in \mathbb{F}_2^2$ and $v_0 \in \mathbb{F}_2$, we note $v = v_0\_$ if the value of the first bit of v is known to be v_0 and the value of the second bit is yet to be determined.
- We note $\bullet$ the scalar product of $\mathbb{F}_2^4$.
- We note a *nibble* to be a unit of four bits, corresponding to half a byte.

4.2 Probability Property

The following property on sums of products of uniform, independent and identically distributed random variables over $\mathbb{F}_2$ will be used in subsubsection 9.1.5.

Property 1. *Let $n \in \mathbb{N}^\star$. Let $a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}$ be $2n$ uniform, independent and identically distributed (i.i.d.) random variables over $\mathbb{F}_2$. Then,*

$$\mathbb{P}(a_0b_0 \oplus \ldots \oplus a_{n-1}b_{n-1} = 0) = \frac{10}{2^{n+2}} - \frac{1}{2^{n-1}} + \frac{1}{2}$$

Proof. We first suppose that $n = 1$. Then $\mathbb{P}(a_0b_0 = 0) = 1 - \mathbb{P}(a_0b_0 = 1)$. As a_0 and b_0 are uniform and i.i.d. variables over $\mathbb{F}_2$, $\mathbb{P}(a_0b_0 = 1) = \mathbb{P}(a_0 = 1)\mathbb{P}(b_0 = 1) = \frac{1}{2} * \frac{1}{2} = \frac{1}{4}$. Thus $\mathbb{P}(a_0b_0 = 0) = 1 - \frac{1}{4} = \frac{3}{4}$.

Simultaneously, for $n = 1$, $\frac{10}{2^{n+2}} - \frac{1}{2^{n-1}} + \frac{1}{2} = \frac{10}{2^{1+2}} - \frac{1}{2^{1-1}} + \frac{1}{2} = \frac{10}{8} - 1 + \frac{1}{2} = \frac{5}{4} - \frac{1}{2} = \frac{3}{4}$.

We now suppose that there exists $n \in \mathbb{N}^\star$ such that the property is verified for this value of n, i.e. for any $2n$ uniform and i.i.d. random variables over $\mathbb{F}_2$ $a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}$, $\mathbb{P}(a_0b_0 \oplus \ldots \oplus a_{n-1}b_{n-1} = 0) = \frac{10}{2^{n+2}} - \frac{1}{2^{n-1}} + \frac{1}{2}$. We then consider $a_0, \ldots, a_n, b_0, \ldots, b_n$ $2(n+1)$ uniform and i.i.d. random variables over $\mathbb{F}_2$. Subsequently,

$$\begin{aligned}
\mathbb{P}(a_0b_0 \oplus \ldots \oplus a_{n-1}b_{n-1} \oplus a_nb_n = 0) &= \mathbb{P}(a_0b_0 \oplus \ldots \oplus a_{n-1}b_{n-1} = 0 \cap a_nb_n = 0) \\
&+ \mathbb{P}(a_0b_0 \oplus \ldots \oplus a_{n-1}b_{n-1} = 1 \cap a_nb_n = 1) \\
&= \mathbb{P}(a_0b_0 \oplus \ldots \oplus a_{n-1}b_{n-1} = 0)\mathbb{P}(a_nb_n = 0) \\
&+ \mathbb{P}(a_0b_0 \oplus \ldots \oplus a_{n-1}b_{n-1} = 1)\mathbb{P}(a_nb_n = 1) \\
&= \left(\frac{10}{2^{n+2}} - \frac{1}{2^{n-1}} + \frac{1}{2}\right) * \frac{3}{4} \\
&+ \left(1 - \left(\frac{10}{2^{n+2}} - \frac{1}{2^{n-1}} + \frac{1}{2}\right)\right) * \frac{1}{4} \\
&= \frac{30}{2^{n+4}} - \frac{3}{2^{n+1}} + \frac{3}{8} + \frac{1}{4} - \frac{10}{2^{n+4}} + \frac{1}{2^{n+1}} \\
&- \frac{1}{8} \\
&= \frac{20}{2^{n+4}} - \frac{2}{2^{n+1}} + \frac{2}{8} + \frac{1}{4} \\
&= \frac{10}{2^{(n+1)+2}} - \frac{1}{2^{(n+1)-1}} + \frac{1}{2}
\end{aligned}$$

Therefore, by induction, the property is verified for every $n \in \mathbb{N}^\star$. □

4.3 Substitution Boxes Properties

In 1949, Shannon suggested to combine two different approaches to mitigate the cryptanalysis of a block cipher, namely diffusion and confusion ([Sha49], [Wys09]).

In a few words, diffusion consists in maximizing the propagation of any plaintext bit or key bit difference to all ciphertext bits. Ideally, flipping one of these plaintext or key bits would lead to a flip of each ciphertext bits with probability $\frac{1}{2}$. Diffusion is usually achieved by bit permutations and/or linear operations in block ciphers.

Confusion consists in complexifying as much as possible the dependence between plaintext, key and ciphertext bits. To that end, block ciphers use non-linear operations, usually defined as Substitution Boxes (SBoxes) implemented as look-up tables.

Property 2. *Let $n \in \mathbb{N}^\star$. There exists $2^n!$ n-bit to n-bit SBoxes.*

Property 3. *Let $m, n \in \mathbb{N}^\star$. An m-bit input to n-bit output SBox weighs $2^m * n$ bits.*

To achieve a good confusion, different properties of SBoxes can be considered, as listed in [BPP$^+$17b] :

Definition 1 (DDT)**.** Let S be a m-bit to n-bit SBox. The Difference Distribution Table (**DDT**) of S is the $2^m \times 2^n$-table defined such that

$$DDT(\delta_i, \delta_o) = \#\{x \in \mathbb{F}_2^m \mid S[x \oplus \delta_i] = S[x] \oplus \delta_o\} \text{ for } (\delta_i, \delta_o) \in \mathbb{F}_2^m \times \mathbb{F}_2^n$$

In particular, Banik *et. al.* introduce in [BPP$^+$17b] the 1-1 DDT, i.e. the sub-table of the Difference Distribution Table DDT composed of coefficients $DDT(\delta_i, \delta_o)$ such that the input and output differences $\delta_i \in \mathbb{F}_2^m$ and $\delta_o \in \mathbb{F}_2^n$ have both Hamming weight one.

Definition 2 (from [BPP$^+$17b])**.** The **differential score** of an SBox S is $\#GI + \#GO$, observed from 1-1 bit DDT.

- The Good Inputs (GI) observed from the 1-1 DDT are the input differences $\delta_i \in \mathbb{F}_2^m$ verifying $HW(\delta_i) = 1$ such that there does not exist an input $x \in \mathbb{F}_2^m$ and an output difference $\delta_o \in \mathbb{F}_2^n$ verifying $HW(\delta_o) = 1$ such that $S[x \oplus \delta_i] = S[x] \oplus \delta_o$.
- The Good Outputs (GO) observed from the 1-1 DDT are the output differences $\delta_o \in \mathbb{F}_2^n$ verifying $HW(\delta_o) = 1$ such that there does not exist an input $x \in \mathbb{F}_2^m$ and an input difference $\delta_i \in \mathbb{F}_2^m$ verifying $HW(\delta_i) = 1$ such that $S[x \oplus \delta_i] = S[x] \oplus \delta_o$.

All inputs (resp. outputs) that are not Good Inputs (resp. Good Outputs) are considered Bad Inputs (resp. Bad Outputs).

Definition 3 (LAT)**.** Let S be a 4-bit SBox. The Linear Approximation Table (**LAT**) of S is the 16×16-table defined such that

$$LAT(\alpha, \beta) = \#\{x \in \mathbb{F}_2^4 \mid \alpha \bullet x = \beta \bullet S[x]\} - 8 \text{ for } (\alpha, \beta) \in \mathbb{F}_2^4 \times \mathbb{F}_2^4$$

Similarly as for the DDT, Banik *et. al.* introduce in [BPP$^+$17b] the 1-1 LAT.

Definition 4 (from [BPP+17b])**.** The **linear score** of an SBox S is $\#GI + \#GO$, observed from 1-1 bit LAT.

- The Good Inputs (GI) observed from the 1-1 LAT are the input operands $\alpha \in \mathbb{F}_2^4$ verifying $HW(\alpha) = 1$ such that for each output operand $\beta \in \mathbb{F}_2^4$ $LAT(\alpha, \beta) = 0$, i.e. $\#\{x \in \mathbb{F}_2^4 \mid \alpha \bullet x = \beta \bullet S[x]\} = 8$.
- The Good Outputs (GO) observed from the 1-1 LAT are the output operands $\beta \in \mathbb{F}_2^4$ verifying $HW(\beta) = 1$ such that for each input operand $\alpha \in \mathbb{F}_2^4$, $LAT(\alpha, \beta) = 0$, i.e. $\#\{x \in \mathbb{F}_2^4 \mid \alpha \bullet x = \beta \bullet S[x]\} = 8$.

All inputs (resp. outputs) that are not Good Inputs (resp. Good Outputs) are considered Bad Inputs (resp. Bad Outputs).

Furthermore, potential supplementary properties of Substitution Boxes may fragilize the cryptographic primitives they are used in : for example, their linearity.

Property 4 (Cardinality of Linear SBoxes)**.** *Let $n \in \mathbb{N}_{\geqslant 2}$. The number of linear n-bit Substitution Boxes verify*

$$(2^n - 1) * (2^n - 2) * \ldots * (2^n - 2^{n-1}) = \prod_{i=0}^{n-1} (2^n - 2^i)$$

Proof. Let S be a n-bit linear SBox. For every x such that $0 \leqslant x < 2^n$,

$$S[(x)_n] = S[(x)_n \oplus (0)_n] = S[(x)_n] \oplus S[(0)_n],$$

thus $S[(0)_n] = (0)_n$.

By linearity, all values of the SBox only depend on the values of $S[(1)_n]$, $S[(2)_n], S[(2^2)_n], \ldots, S[(2^{n-1})_n]$. Indeed, for every $x \in \mathbb{N}$ such that $0 \leqslant x < 2^n$, there exists $(x_{n-1}, \ldots, x_0) \in \mathbb{F}_2^n$ such that $x = \sum_{i=0}^{n-1} x_i 2^{i-1}$. Subsequently, $S[(x)_n] = \bigoplus_{i=0}^{n-1} x_i S[(2^i)_n]$.

- As mentioned above, the value of $S[(0)_n]$ is fixed to be $(0)_n$. Therefore, there exists $2^n - 1$ potential values for $S[(1)_n]$, that are the non-zero values of $\mathbb{F}_2^n$. In the same manner, it implies that, given the value of $S[(1)_n]$ there exists $2^n - 2$ possible values of $S[(2)_n]$: the values of $\mathbb{F}_2^n \setminus \{S[(0)_n], S[(1)_n]\}$.

  Thus, at this point, the values of $S[(0)_n]$, $S[(1)_n]$, $S[(2)_n]$ and $S[(3)_n] = S[(1)_n] \oplus S[(2)_n]$ are fixed.
- It remains $2^n - 2^2$ possible values for $S[(2^2)_n]$: the elements of $\mathbb{F}_2^n \setminus \{S[(i)_n]$ for $0 \leqslant i \leqslant 3\}$. Then, once this value chosen, the 2^3 values $\{S[\sum_{i=0}^{2} x_i (2^i)_n]$ for $(x_2, x_1, x_0) \in \mathbb{F}_2^3\}$ are fixed.

- In the same way, it remains $2^n - 2^3$ different values for $S[(2^3)_n]$: the elements of $\mathbb{F}_2^n \setminus \{S[\sum_{i=0}^{2} x_i(2^i)_n]$ for $(x_2, x_1, x_0) \in \mathbb{F}_2^3\}$. Once this value has been chosen, the 2^4 first values of S ($\{S[\sum_{i=0}^{3} x_i(2^i)_n]$ for $(x_3, x_2, x_1, x_0) \in \mathbb{F}_2^4\}$) are fixed.

This process can be repeated : for the choice of the value of $S[(2^k)_n]$ there exists $2^n - 2^k$ possibilities : the elements of $\mathbb{F}_2^n \setminus \{S[\sum_{i=0}^{k-1} x_i 2^i]$ for $(x_{k-1}, \ldots, x_0) \in \mathbb{F}_2^k\}$. The last value to be chosen will be $S[(2^{n-1})_n]$, with $2^n - 2^{n-1}$ possibilities.

As a conclusion, as detailed above all values of the linear SBox S can be deduced from $S[(2^0)_n]$, $S[(2^1)_n], \ldots, S[(2^{n-1})_n]$, and there exists respectively $2^n - 2^0$, $2^n - 2^1, \ldots, 2^n - 2^{n-1}$ possibilities for each. Hence, the number of linear n-bit SBoxes verify

$$(2^n - 1) * (2^n - 2) * \ldots * (2^n - 2^{n-1}) = \prod_{i=0}^{n-1} (2^n - 2^i)$$

□

4.4 Bitslicing

Firstly introduced and applied on DES in [Bih97], bitslicing consists in implementing an operation as a combinatorial circuit in software ([MDLM18]). In other words, the basic idea of bitslicing is to implement an operation using only Boolean operators XOR, AND, OR and NOT on bits. To that end, n-bit variables of this operation are implemented as n variables of one bit.

Bitslicing look-up tables avoids cache-timing attacks, as the result value of the circuit is computed at each instance and cannot be stored into the cache. For example, [Kwa00] describes how to implement the 4-bit SBox of DES using 56 gates XOR, AND, OR or NOT, and the 3-bit SBox $S = [2, 0, 1, 6, 3, 5, 4, 7]$ can be written as $b_2 b_1 b_0 = S[a_2 a_1 a_0]$ with

$$\begin{aligned} x_0 &= a_0 \text{ AND } a_1 & \qquad x_4 &= x_2 \text{ AND } a_1 \\ x_1 &= a_0 \text{ XOR } a_1 & \qquad b_0 &= x_4 \text{ XOR } a_2 \\ x_2 &= \text{ NOT } a_0 & \qquad b_1 &= x_2 \text{ XOR } a_1 \\ x_3 &= x_1 \text{ AND } a_2 & \qquad b_2 &= x_0 \text{ XOR } x_3 \end{aligned}$$

Bitslicing is notably useful when performing several instances of the implementation in a parallel manner is needed. Indeed, a m-bit register can be filled with m one-bit variables arised from m different instances of the implementation.

Bitsliced implementations are by nature resistant to timing attacks. Timing attacks aim to extract the key (or a secret variable) from a cryptographic primitive by exploiting the variations of the execution time depending on this primitive inputs [BB05]. Bitsliced implementations are composed of the bitwise

operations mentioned hereinabove, and those operations are by design constant-time, hence the timing attack resistance.

There exists different ways to bitslice an implementation. Some papers like [MPC00] or [BLL15] describe bitsliced implementations specifically designed for certain cryptographic primitives. Many methods destined to bitslice any cryptographic primitive focus on the bitslicing of SBoxes, that are the usual non-linear operations of those primitives ([GR16], [SKP23]). To be able to fully bitslice any primitive, [Mer20] introduces the Usuba language and its corresponding compiler Usubac. Usubac takes the description of a symmetric cryptographic primitive in this language and produces a bitsliced implementation of the primitive in C.

4.5 Error-Correcting Codes

For the new masking scheme described in section 10, we firstly aim to use an error-correcting code with an easy management of codewords parities, hence the choice of cyclic codes. Furthermore, since the masking scheme is designed to only be constituted of Boolean operations, we choose BCH codes as they can be decoded in constant time via the Peterson-Gorenstein-Zierler algorithm ([Pet60]).

To that end, we first remind basic properties of error-correcting codes and subsequently BCH error-correcting codes.

4.5.1 Basic Properties

We first remind basic properties of error-correcting codes needed for the remainder of the dissertation.

Definition 5. Let A be a finite alphabet and $n \in \mathbb{N}^\star$. An *error-correcting code* over A of length n is a non-empty set of A^n.

Definition 6 (Minimal Distance)**.** Let $\mathcal{C}$ be an error-correcting code of length n. The minimal Hamming distance d of $\mathcal{C}$ verify

$$d = \min_{x,y \in \mathcal{C}} d_{HW}(x, y)$$

Definition 7 (Correction Capacity)**.** Let $\mathcal{C}$ be an error-correcting code of length n and minimal Hamming distance d. The correction capacity t of the code $\mathcal{C}$ verifies

$$t = \lfloor \frac{d-1}{2} \rfloor.$$

Definition 8 (Cyclic codes [ABO09])**.** Let $n \in \mathbb{N}^\star$. A linear code $\mathcal{C}$ of length n is said to be a **cyclic code** if all cyclic permutations of codewords belong to the code as well, i.e.

$$(c_0, c_1, \ldots, c_{n-1}) \in \mathcal{C} \Longrightarrow (c_{n-1}, c_0, c_1, \ldots, c_{n-2}) \in \mathcal{C}$$

Property 5 (Polynomial Representation of Codewords)**.** *For a cyclic code $\mathcal{C}$ of length n over a finite field $\mathbb{F}_q$,*

- *we can identify each codeword $(c_0, c_1, \ldots, c_{n-1}) \in \mathbb{F}_q^n$ to its polynomial representation $c_0 + c_1X + c_2X^2 + \ldots + c_{n-1}X^{n-1} \in \mathbb{F}_q[X]/(X^n - 1)$ and reciprocally.*
- *there exists a polynomial $g(X) \in \mathbb{F}_q[X]$ such that, for each codeword $(c_0, c_1, \ldots, c_{n-1})$, $g(X)|c_0 + c_1X + c_2X^2 + \ldots + c_{n-1}X^{n-1}$. This polynomial $g(X)$ is noted **generator polynomial** of the cyclic code $\mathcal{C}$.*

4.5.2 BCH Error-Correcting Codes

BCH error-correcting codes are based on the following mathematical notions and properties. Thereafter, we will assume q to be a prime power.

Definition 9. Let $n \in \mathbb{N}^\star$ such that $n \wedge q = 1$, m the multiplicative order of q modulo n. Let $s \in \{0, \ldots, n-1\}$. We note $C(s)$ the *q-cyclotomic class of s modulo n* :

$$C(s) = \{s, s * q, \ldots, s * q^{m_s - 1}\},$$

with m_s being the smallest non-zero integer verifying $s = s * q^{m_s} \pmod{n}$.

Definition 10. Let $n \in \mathbb{N}^\star$ such that $n \wedge q = 1$, m the multiplicative order of q modulo n and α an element of $\mathbb{F}_{q^m}$ of multiplicative order n. Let $s \in \{0, \ldots, n-1\}$. We note the polynomial $M_{\alpha^s}(X)$ to be

$$M_{\alpha^s}(X) = \prod_{i \in C(s)} (X - \alpha^i) = \prod_{i=0}^{m_s - 1} (X - \alpha^{sq^i}).$$

Subsequently, the BCH (Bose Chaudhuri Hocquenghem) error-correcting codes can be defined as follows :

Definition 11 (BCH Code, [BRC60a], [BRC60b])**.** Let $n \in \mathbb{N}^\star$, m the multiplicative order of q modulo n, α an element of $\mathbb{F}_{q^m}$ of multiplicative order n and $b, \delta \in \mathbb{N}$ such that $\delta \geqslant 3$. A cyclic code of length n over $\mathbb{F}_q$ is said to be a **BCH code** of minimum Hamming distance at least δ if its generator polynomial $g(X)$ verifies

$$g(X) = lcm(M_{\alpha^b}(X), M_{\alpha^{b+1}}(X), \ldots, M_{\alpha^{b+\delta-2}}(X))$$

Property 6 (Correction Capacity of a BCH Code)**.** *The correction capacity t of a BCH code of minimum Hamming distance at least δ verifies*

$$t \geqslant \lfloor \frac{\delta - 1}{2} \rfloor.$$

4.5.3 Additional Properties

For the selection of the parameters of the BCH code we use in the scheme that is detailed in subsection 9.1, we consider the following additional properties about the 2-cyclotomic classes and M_{α^i} polynomials.

Property 7. *Let odd $n \in \mathbb{N}^\star$ such that $n \geqslant 3$ and $s \in \{0, \ldots, n-1\}$. The 2-cyclotomic class of s modulo n is a singleton if and only if $s = 0$.*

Proof. On the one hand, by definition, $C(0) = \{0\}$. On the other hand, suppose that there exists $s \in \{0, \ldots, n-1\}$ such that $C(s) = \{s\}$. This implies that m_s, defined as the smallest non-zero integer such that $s = s * 2^{m_s} \pmod{n}$, also verifies that $s * 2^{m_s - 1} = s$.

Therefore, as naturally $m_s - 1 < m_s$, the hypothesis on m_s implies that $m_s - 1 = 0$, and thus $m_s = 1$. Therefore, s verifies $2 * s = s \pmod{n}$, i.e. $s = 0 \pmod{n}$. Since $s \in \{0, \ldots, n-1\}$, this implies that $s = 0$. As a conclusion, if the 2-cyclotomic class of s modulo n is a singleton, then $s = 0$.

□

Property 8. $M_{\alpha^s} \in \mathbb{F}_q[X]$.

Proof. First of all, $(M_{\alpha^s}(X))^q = (\prod_{i=0}^{m_s-1}(X - \alpha^{sq^i}))^q = \prod_{i=0}^{m_s-1}(X - \alpha^{sq^i})^q = \prod_{i=0}^{m_s-1} X^q - \alpha^{sq^{i+1}}$, therefore $(M_{\alpha^s}(X))^q = \prod_{i=0}^{m_s-1} X^q - \alpha^{sq^{i+1}} = \prod_{i=0}^{m_s-1} X^q - \alpha^{sq^i} = M_{\alpha^s}(X^q)$.

Subsequently, we note $M_{\alpha^s}(X) = \sum_{j=0}^{m_s-1} a_j X^j$, with $a_j \in \mathbb{F}_{q^m}$ for $0 \leqslant j \leqslant m_s - 1$. Then,

- $(M_{\alpha^s}(X))^q = (\sum_{j=0}^{m_s-1} a_j X^j)^q = \sum_{j=0}^{m_s-1} a_j^q X^{qj}$
- $M_{\alpha^s}(X^q) = \sum_{j=0}^{m_s-1} a_j (X^q)^j = \sum_{j=0}^{m_s-1} a_j X^{qj}$

Thus, $M_{\alpha^s}(X)^q = M_{\alpha^s}(X^q) \implies \sum_{j=0}^{m_s-1} a_j^q X^{qj} = \sum_{j=0}^{m_s-1} a_j X^{qj}$.

Consequently, since these polynomials are equal, it implies that for all $0 \leqslant j \leqslant m_s - 1$, $a_j = a_j^q$, hence $a_j \in \mathbb{F}_q$. In conclusion, $M_{\alpha^s}(X) = \sum_{j=0}^{m_s-1} a_j X^j \in \mathbb{F}_q[X]$. □

Property 9. *M_{α^s} is irreducible over $\mathbb{F}_q$.*

Proof. By definition, $\alpha^s \in \mathbb{F}_{q^m}$ is a root of M_{α^s}. If we suppose that M_{α^s} is not irreducible over $\mathbb{F}_q[X]$, then there exists a factor $P \in \mathbb{F}_q[X]$ admitting α^s as a root, i.e. $P(\alpha^s) = 0$. As $P \in \mathbb{F}_q[X]$, $P(\alpha^{sq}) = P(\alpha^s)^q = 0$ and by extension

$P(\alpha^{sq^i}) = 0$ for all $0 \leqslant i \leqslant m_s - 1$. Consequently, $M_{\alpha^s}(X) = \prod_{i=0}^{m_s-1} (X - \alpha^{sq^i})$ divides P.

Moreover, $P | M_{\alpha^s}$ by hypothesis, therefore $P = M_{\alpha^s}$. In conclusion, M_{α^s} is irreducible over $\mathbb{F}_q[X]$ (and is the minimal polynomial of α^s over $\mathbb{F}_q[X]$). □

Property 10. *Let $\{i_0, i_1, \ldots, i_{r-1}\} \subset \{0, \ldots, n-1\}$. The least common multiple of $M_{\alpha^{i_0}}, M_{\alpha^{i_1}}, \ldots, M_{\alpha^{i_{r-1}}}$ is a product of some of these M_{α^j} polynomials.*

Proof. First of all, we note $\{j_0, \ldots, j_{t-1}\}$ the representatives of the different cyclotomic classes of $i_0, i_1, \ldots, i_{r-1}$, in such a way that each different polynomial $M_{\alpha^{i_v}}$ only appears once in the set $\{M_{\alpha^{j_0}}, \ldots, M_{\alpha^{j_{t-1}}}\}$. Therefore,

$$lcm(M_{\alpha^{i_0}}, M_{\alpha^{i_1}}, \ldots, M_{\alpha^{i_{r-1}}}) = lcm(M_{\alpha^{j_0}}, \ldots, M_{\alpha^{j_{t-1}}}),$$

with the polynomials $M_{\alpha^{j_0}}, \ldots, M_{\alpha^{j_{t-1}}}$ being two by two distincts. As these polynomials are irreducible (see Property 9) and two by two distincts, they are coprime. Therefore,

$$\begin{aligned} lcm(M_{\alpha^{i_0}}, M_{\alpha^{i_1}}, \ldots, M_{\alpha^{i_{r-1}}}) &= lcm(M_{\alpha^{j_0}}, \ldots, M_{\alpha^{j_{t-1}}}) \\ &= \frac{M_{\alpha^{j_0}} * M_{\alpha^{j_1}} * \ldots * M_{\alpha^{j_{t-1}}}}{gcd(M_{\alpha^{j_0}}, M_{\alpha^{j_1}}, \ldots, M_{\alpha^{j_{t-1}}})} \\ &= M_{\alpha^{j_0}} * M_{\alpha^{j_1}} * \ldots * M_{\alpha^{j_{t-1}}} \end{aligned}$$

Thus, the least common multiple of $M_{\alpha^{i_0}}, M_{\alpha^{i_1}}, \ldots, M_{\alpha^{i_{r-1}}}$ is a product of some of these polynomials.

□

5 State of the Art

5.1 State of the Art of White-Boxing Contests

Since the development of the white-box cryptography, a cat-and-mouse challenge has been running between designers and attackers. For a long time, the security of most solutions deployed in the industry relied on secrecy rather than on academic-proved designs. To that end, the ECRYPT-CSA consortium organized for the first time WhibOx, a white-box cryptography competition during the 2017 edition of CHES [Whia]. The purpose of this Catch The Flag challenge was to allow white-box design researchers to confront their implementations to all attackers of the white-box cryptography community.

Participant designers were invited to submit their implementations of AES-128 with freely-chosen fixed keys, written with only generic C instructions. Each submission needed to comply with different requirements, for example a source code of at most 50 MB, a compilation time of less than 100 seconds, an executable being 20 MB in size or less and using 20 MB of RAM or less, and a function call time of less than one second on average. The results of this contest illustrated the difficulty of white-box secure cryptography designing, as none of the 94 submitted challenges had remained unbroken at the end. Precisely, among those 94 submissions :

- 88 were broken in less than one week (94%), including
- 81 that were broken in less than a day (86%), including
- 55 that were broken in less than an hour (59%), including
- 38 that were broken in less than 30 minutes (40%), including
- 20 that were broken in less than 10 minutes (21%).

More precisely, the three most successful challenges resisted for respectively 11, 12 and 27 days ([GPRW20]).

Subsequently, a second edition of WhibOx was organized in 2019 by CryptoExperts and CyberCrypt, that was still concerning AES-128 implementations in C [Whib]. Most of the requirements imposed during the first edition of WhibOx (including the ones mentioned above) were renewed. Among the 27 submissions of this edition :

- the 24 submissions that were broken (89%) were broken in less than a month, including
- 23 that were broken in less than two weeks (85%), including
- 18 that were broken in less than a week (67%), including
- 6 that were broken in less than a day (22%), including
- none that were broken in less than an hour.

Three challenges had remained unbroken by the end of the contest (respectively 21 and 24 days after their submissions), but were later broken with grey-box attacks in [GRW20].

Finally, a third edition of the WhibOx contest was organized as the CHES 2021 Catch the Flag Challenge [Whic]. Nevertheless, the focus of this edition was public-key white-box cryptography, and more specifically the ECDSA signature (on NIST P256 curve) under a freely chosen secret key. Therefore the submissions were not in the symmetric white-box cryptography scope of this thesis.

5.2 Masking Offers Resistance to Side-Channel Attacks

The analysis of the security of the candidates of the WhibOx 2017 contest detailed in [AT20] illustrates that side-channel attacks are a major threat to white-box implementations.

Masking constitutes a countermeasure to side-channel attacks. Indeed, as detailed in subsection 3.2, masking a variable $x \in \mathbb{K}$ with n_{in} shares x_0, ..., $x_{n_{in}-1}$ in $\mathbb{K}$ consists in replacing in an implementation the variable x by its shares $(x_i)_{0 \leqslant i < n_{in}}$. By definition of sharings, each tuple of $n_{in} - 1$ shares x_i is independent from x. Therefore, as the side-channel leakage of each share is independent from the side-channel leakage of other shares, recovering the variable x requires the retrieval of all n_{in} shares.

Consequently, as detailed notably in [Cas22], if each share x_i can be recovered with a probability p_w then, since an attacker will be able to recover the original variable x only by retrieving all n_{in} shares x_i, he will be able to retrieve the value of x with a probability being $\mathcal{O}(p_w^{n_{in}})$, hence the security enhancement.

5.3 Different Strategies to Mask an Implementation

Different types of masking have been developed in the literature. A branch has been notably devoted to the design of masking dedicated to specific cryptographic primitives. In these cases, a different masking scheme $(F_i)_{0 \leqslant i < n_{out}}$ is designed for each component function f of the considered primitive.

The most studied primitive to be masked is AES. For example, in 2012, [NSGD12] expounded RSM, a countermeasure masking each component function of AES-256, that was later improved in [BBD+14]. Many papers including [ZSM+08], [SBS16] and [GD17] focus on the masking of the SBox in AES, since it is its only non-linear operation, with high complexity. On the other hand, in 2021, [MZLZ21] aimed to improve the masking of the non-linear operations of AES, weaknesses of the then-existing constructions. Finally, in 2022, [ADN+22] focused on designing three different AES maskings that do not require fresh randomness.

The other major strategy consists in designing masking schemes operating on a lower level of the implementations of primitives, therefore allowing those schemes to not be specific to a cryptographic primitive. To that end, the implementations on which those masking schemes are applied are usually bitsliced

implementations of any cryptographic primitive. Most masking schemes with this aim in mind are designed in finite fields of characteristic two (i.e. of the form $\mathbb{F}_{2^k}$), in particular $\mathbb{F}_2$. In this instance, the operations are mainly bitwise operations, namely XOR, AND, OR and NOT.

The bitwise operation f of a bitsliced implementation mostly targeted to be masked when following the second strategy is the AND operation. Indeed, among the four Boolean operations XOR, AND, NOT and OR,

- The XOR operation is associative with regards to input sharings. Therefore, performing a XOR between two variables using their sharings can be performed simply by XORing shares.
- The NOT operation can be performed on a sharing of a variable by flipping the value of one of its shares.
- The OR operation can be rewritten with three instances of the NOT operation and one instance of the AND operation, since $a \vee b = \neg(\neg a \wedge \neg b)$.

5.4 State of the Art of Masking Schemes Security Properties

Different models have been introduced in the literature to assess the security of masking schemes, and different properties of these masking schemes can be sufficient conditions to ensure their security in those models.

5.4.1 Noisy Leakage Model Implies d-Probing Security

In 1999, Chari et al. introduced in their paper [CJRR99] the noisy leakage model which aimed to be the more realistic leakage model, where the adversary can obtain leaked values that are sampled thanks to a Gaussian distribution centered on the real value of the sensitive variables. This model was later extended by Rivain and Prouff in [PR13] to general noise distributions. The noisy leakage model allows to simulate leakage in a rather precise manner, but is not very easy to use in practice. That is why, in [ISW03], Ishai et al. introduced the d-probing security model.

Property 11 (d-probing security)**.** *A circuit is d-probing secure if and only if every set of d intermediate variables is independent of any sensitive variable. For every set of d probes* $(p_0, \ldots, p_{d-1})$*, this amounts to*

$$I(p_0 \cup \ldots \cup p_{d-1}; x) = 0$$

This property is much easier to prove than security in the noisy leakage model, but was thought to be not sufficiently accurate to describe the leakage. However, in 2014, Duc et al. proved in [DDF14] that security in the d-probing model implies security in the noisy leakage model.

5.4.2 Correctness

The first property that all masking schemes must comply with is correctness ([Bil15]). This property does not participate in the security of the scheme by itself, but is essential for the natural purpose of maintaining the functionality of the function to be masked f. Indeed, for a masking $(F_i)_{0 \leqslant i \leqslant n_{out}-1}$ of a function f such that $Y = f(X_1, \ldots, X_n)$, the output shares functions $(F_i)_{0 \leqslant i \leqslant n_{out}-1}$ must verify that

$$\bigoplus_{i=0}^{n_{out}-1} F_i\left((X_{0,j})_{0 \leqslant j < n_{in}}, \ldots, (X_{n-1,j})_{0 \leqslant j < n_{in}}\right) = f\left(\bigoplus_{j=0}^{n_{in}-1} X_{0,j}, \ldots, \bigoplus_{j=0}^{n_{in}-1} X_{n-1,j}\right)$$

5.4.3 Non-Completeness

Property 12 (Non-Completeness, [NRR06], [Bil15])**.** *A masking scheme F verifies non-completeness if each of its component functions F_i is independent of at least one share of each of the input variables of the scheme.*

In other words, non-completeness is necessary when supposing that a probe on a combinational block (i.e. an operation) implies the leakage of all inputs of this combinational block ([RBN+15]). Consequently, for a masking scheme that does not satisfy non-completeness, a probe would imply the leakage of all shares of at least one input variable, and therefore the recovery of the value of this input.

5.4.4 Uniformity

Property 13 (Uniform Masking, [Bil15])**.** *A (n_{in}, n_{out})-masking scheme F of a function $f : \mathbb{K}^n \to \mathbb{K}$ with n inputs X_i and one output Y is said to be uniform if and only if*

$$\forall (x_0, \ldots, x_{n-1}) \in \mathbb{K}^n, \forall (y_0, \ldots, y_{n_{out}-1}) \in \mathbb{K}^{n_{out}},$$
$$P\left((Y_j)_{0 \leqslant j \leqslant n_{out}-1} = (y_j)_{0 \leqslant j \leqslant n_{out}-1} \middle| (X_i)_{0 \leqslant i \leqslant n-1} = (x_i)_{0 \leqslant i \leqslant n-1}\right) =$$
$$\begin{cases} \frac{1}{|\mathbb{K}|^{n_{out}-1}} & \text{if } f(x_0, \ldots, x_{n-1}) = \bigoplus_{j=0}^{n_{out}-1} y_j \\ 0 & \text{otherwise} \end{cases}$$

In other words, uniformity implies that, for each n-tuple of input values of f noted $(x_0, \ldots, x_{n-1})$, all n_{out}-sharings $(y_0, \ldots, y_{n_{out}-1})$ that are computed by F with a n_{in}-sharing of $(x_0, \ldots, x_{n-1})$ as input are equiprobable.

5.4.5 Non-Completeness and Uniformity Imply 1-Probing Security

In [NRR06], Nikova et al. demonstrate that the three properties of correctness, uniformity and non-completeness constitute, when combined, sufficient conditions for the security of implementations against first-order probing attacks.

Lemma 1 ([NRR06])**.** *Non-completeness and uniformity implies 1-glitch probing extended security.*

5.5 State of the Art of Boolean AND Masking Schemes

Our purpose is to design a masking scheme implementing the AND operation between two bits with only Boolean operations. Among Boolean operations, the masking of AND is by far the most developed since XOR and NOT are linear with regards to their input shares. Therefore, they can be performed on input sharings by respectively XORing shares or performing NOT on only one input share. Finally, the OR operation can be written as a combination of instances of AND and NOT.

We test the uniformity and non-completeness properties on four different examples of such masking schemes : ISW ([ISW03]) and the three different multiplication gadgets presented in [GJRS18], namely the BDF+ algorithm of [BDF+17], the BBP+ algorithm of [BBP+16] and the BCPZ algorithm of [BCPZ16]. The results are presented in Table 1.

	ISW			BDF+			BBP+			BCPZ		
Number of shares d	2	4	8	2	4	8	2	4	8	2	4	8
Uniformity	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
1^{st}-Order Non-Completeness	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗
2^{nd}-Order Non-Completeness	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Probing Security Order	1	3	7	1	3	7	1	1	3	1	3	7
Clock cycles	75	291	1155		77	146		344	1204	108	498	2106
Code Size (Bytes)	164	164	164		248	244		344	344	240	648	2324
Random Variables	1	6	28	1	1	2	1	5	19	1	10	68

Table 1: Properties of ISW, BDF+, BBP+ and BCPZ gadgets

The implementations results, taken from [GJRS18], consider the straight implementations of ISW, BDF+ and BBP+ with loops, and BCPZ with macros.

5.6 Fault-Resistant Masking Schemes

The study of the WhibOx 2017 candidates in [AT20] also illustrates that fault attacks constitute the other major threat to white-box implementations. Therefore, fault-resistant masking schemes have been developed in order to combine

resistance against the listed two major threats to white-box implementations, namely side-channel attacks and fault attacks. One of the solutions frequently used to do so consists in introducing error-correcting codes in the design of masking schemes.

The strategy most fault-resistant masking schemes use is detecting potential faults rather than correcting them. In the case of masking schemes based on error-correcting codes, it uses the fault detection property of the code to deteriorate the computation in the presence of a fault : this avoids the costly procedure of correction. For example, authors of [MAN+19] detail a countermeasure to combined side-channel and fault attacks, that associates masking with infective computation at the detection of a fault. Likewise, [RDB+18] describes CAPA (combined Countermeasure Against Physical Attacks) that is a side-channel countermeasure inherited from Multi-Party Computation. This methodology aborts the computation when detecting faults. Furthermore, [CCG+19] describes IPM-FD, a masking scheme derived from IPM (Inner Product Masking, [BFGV12]) and DSM (Direct Sum Masking, [CGM19]) that performs error detection as well.

Most of the masking schemes using error-correcting codes necessitate the construction of suitable new error-correcting codes, as for example [CG16], and operate with words of these codes. Particularly, [BCC+14] introduces Orthogonal Direct Sum Masking (ODSM). This masking uses a code $\mathcal{C}$ and its dual $\mathcal{D}$, then each element $c \in \mathcal{C}$ is masked by XORing an element $d \in \mathcal{D}$. Subsequently, a method is provided to adapt the computation of the different steps of a block cipher, using ODSM. Additionally, most masking schemes using error-correcting codes use Maximum Distance Separable Codes specially designed for the masking scheme, as they can be considered optimal. Nevertheless, [CRZ13] describe a masking scheme using different non-MDS codes, depending on the scheme order.

In conclusion, the reasoning between our approach is to build a masking scheme that not only detects faults but corrects them. The overall computation is thus not distorted or aborted in the presence of a fault, but always return results, results that are correct. To do so, we aim to use an error-correcting code well studied in the literature, since it allows to select a code with an optimized correction process being compatible with bitslicing.

Part I
White-Boxing and NIST Lightweight Standardization Finalists

6 Study of the White-Boxability of NIST Lightweight Finalists

Lightweight cryptography is the cryptography field that aims to develop cryptographic primitives suitable to devices with constrained capacities. To fulfill the increasing need of cryptography in this type of devices, the NIST launched in 2015 the NIST Lightweight Cryptography Standardization Contest, a process to "solicit, evaluate, and standardize" cryptographic algorithms fulfilling these requirements.

Many of those devices with constrained capacities, as for example IoT ones, can be used in a context where they will be vulnerable to white-box attackers. Hence, we study the white-boxability of lightweight cryptographic primitives, and focus on the finalists of the NIST Lightweight Cryptography Standardization Contest.

6.1 Overview of NIST Lightweight Cryptography Standardization Contest

The NIST Lightweight Cryptography Standardization Contest was launched in August 2018 when the NIST published a call for algorithms to be standardized as "lightweight cryptographic standards with authenticated encryption with associated data (AEAD) and optional hashing functionalities" ([SMC+21]). Among the 57 submissions, 56 were confirmed as Round 1 candidates and announced in April 2019. After a first selection process, the 32 Round 2 candidates were revealed in August 2019.

Finally, the 10 finalists were announced in March 2021 :

- ASCON [DEMS21]
- Elephant [KCM20]
- GIFT-COFB [BPP+17b],[BCI+20]
- Grain-128AED [HJM+20]
- ISAP [DEM+20]
- PHOTON-Beetle [BCD+20]
- Romulus [GIK+20]
- SPARKLE [BBdS+20]
- TinyJambu [SSS+20]
- Xoodyak [DHP+20]

In February 2023, NIST announced ASCON to be the standardized lightweight cryptography primitive, as it "meets the needs of most use cases where lightweight cryptography is required".

The different criteria used to decide between candidates were side-channel and fault attacks resistances, costs and performances, third-party analyses and suitability for hardware and software implementations [NIS]. As white-boxability was not part of those criteria, we proceed to review the white-boxability of the 10 finalists of the contest.

6.2 A First Selection on the Ten NIST Lightweight Candidates

To select one of the ten NIST lightweight cryptography finalist candidates, Alex Charlès and I made a first sorting following the arguments described below.

Firstly, *an algorithm must not be broken with the disclosure of a state*. Indeed, in this case, retrieving the key would require to break only the encoding of a state rather than the encoding scheme of the whole algorithm, thus weakening the security of the said encoding scheme. In the NIST status report of the second round candidates ([SMC+21], §3.3.4), PHOTON-Beetle [BCD+20], Sparkle [BBdS+20] and Xoodyak [DHP+20] are not in accordance with this argument.

Secondly, *key information should be spread throughout an algorithm*, as it will oblige a white-box attacker to attack more parts of this algorithm ([Wys09], §3.2).

Finally, the Kerckhoffs's axiom imposes that it is supposed that the attacker knows the algorithm and its design except for the key. Therefore, retrieving a state of an algorithm allows an attacker to compute all operations that are not key-dependent. Thus, Isap [DEM+20], Ascon [DEMS21] and Grain128-AEAD [HJM+20] were eliminated.

6.3 The Choice of GIFT-COFB

6.3.1 Ruling Out TinyJAMBU and Romulus

TinyJAMBU [SSS+20] has an LFSR-based permutation that fills a 128-bit LFSR with its current state to clock it. As the feedback is computed using 5 bits of the LFSR content and one bit of the key, and recovering the bits used for the computation and the resulting feedback implies retrieving the key bits involved, the LFSR and feedback bits need to be encoded. However, each of the LFSR bits used in this computation can be reused up to 5 times during the following clocks, and will need to be decoded at each usage. That constitutes a drawback from the perspective of designing a white-box implementation of this algorithm, as decoding repetitively the same pieces of information could facilitate the attacker's access to them, and potentially lead to the break of the applied encoding scheme.

The 128-bit Romulus [GIK+20] state can be regarded as a matrix of 4x4 bytes. The MixColumns step of Romulus consists in XORing some state bytes with each other, as for TinyJAMBU. Thus, similarly as in TinyJAMBU, it will be necessary to decode some elements multiple times to achieve this MixColumns step, leading to the same potential weakness as the TinyJAMBU one.

6.3.2 The Elephant Case

Elephant [KCM20] uses a function $mask(K, a, b) = mask_K^{a,b}$ that extends a key K depending on $a, b \in \mathbb{N}$. Those parameters a and b depend on the block indexes of the message and associated data. Thus $mask_K^{a,b}$ depends only on constant

inputs, so we would want to precompute it in a white-box implementation to reduce the manipulation of the key. However, the message length and the nonce length are potentially infinite, forcing to restrict their lengths. Though, if Elephant is using the Spongent-π permutation (Dumbo and Jumbo instances), a similar solution to the one we are proposing might be able to encode it, after spreading the XOR of $mask_K^{a,b}$ through the permutation.

6.3.3 Choosing GIFT-COFB

Even though algorithmic white-box implementations may exist for the last three algorithms (TinyJAMBU, Romulus and Elephant), we finally choose GIFT-COFB [BCI+20] for a white-box implementation.

GIFT-COFB uses GIFT [BPP+17b] to perform its cipher. There exists two versions of GIFT : GIFT-64 and GIFT-128. For the rest of our study, we considered GIFT to be GIFT-128. This block cipher GIFT-128 performs a 128-bit keyed encryption of a 128-bit plaintext, whereas COFB allows to cipher arbitrary-long inputs for an Associated Encryption with Associated Data, using GIFT.

We assume that a white-box implementation of GIFT-COFB would be the easier to realise amongst the other algorithms, as the key is only used in GIFT calls. Furthermore, GIFT XORs some key bits near the SubCells step, which allows us to develop an implementation similar to the one proposed in Chow *et al.* AES [CEJvO02].

Finally, we focus on realising a white-box implementation of simply GIFT rather than the whole GIFT-COFB primitive, as the key is not used in GIFT-COFB outside of GIFT occurences. Indeed, GIFT being white-box secure would theorically imply GIFT-COFB being also secure to a white-box attacker.

7 Our White-Box Implementation of GIFT

7.1 Overview of GIFT

As explained previously, we choose to concentrate our efforts into designing a white-box implementation of GIFT. The GIFT primitive consists in forty rounds, and uses three different operations in each of its forty rounds : *SubCells*, *PermBits* and *AddRoundKey* ([BPP+17b]).

Algorithm 1: GIFT(S,RK)

Input : S, the 128-bit state and RK the round keys
Output: S

```
for i = 0 to 39 do
    S ← SubCells(S)
    S ← PermBits(S)
    S ← AddRoundKey(S, RK[i])
end for
return S
```

- *SubCells* applies the 4-bit GIFT SBox GS to all 4-bit nibbles of the 128 bit state.

$$GS = [1, 10, 4, 12, 6, 15, 3, 9, 2, 13, 11, 7, 5, 0, 8, 14]$$

- *PermBits* applies a bitwise permutation PB to the state.
- *AddRoundKey* XORs two bits of the current round key and one bit of a round constant to each 4-bit nibble of the state. To simplify the notations, we consider the round keys RK to include those round constants.

7.2 Presentation of Our Solution

We aim to build a white-box table-based implementation of GIFT. To that end, we first need to make a modification in the layout of GIFT, while retaining its correctness. Indeed, our solution first requires to merge AddRoundKey with SubCells. To do so, as AddRoundKey and SubCells are not consecutive in a same round, PermBits and AddRoundKey will be swapped. We use that, for S the state and E the key and round constants array,

$$PB(S) \oplus E = PB(S \oplus PB^{-1}(E))$$

Therefore, GIFT can be rewritten as described in Algorithm 2, in a way that facilitates the design of a table-based implementation of this primitive.

The SubCells and AddRoundKey operations can then be merged together in a single lookup table whose outputs will be encoded. Subsequently, to keep

Algorithm 2: $\widetilde{GIFT}(S,\widetilde{RK})$

Input : S, the 128-bit state and $\widetilde{RK} = PB^{-1}(RK)$
Output: S

```
for i = 0 to 39 do
    S ← SubCells(S)
    S ← AddRoundKey(S, RK̃[i])
    S ← PermBits(S)
end for
return S
```

the correctness of GIFT, the inverses of the output encodings of the previous round will be applied to the table input. We name the resulting look-up table a TBox : Figure 1 represents T_0, the right-most TBox of a round r.

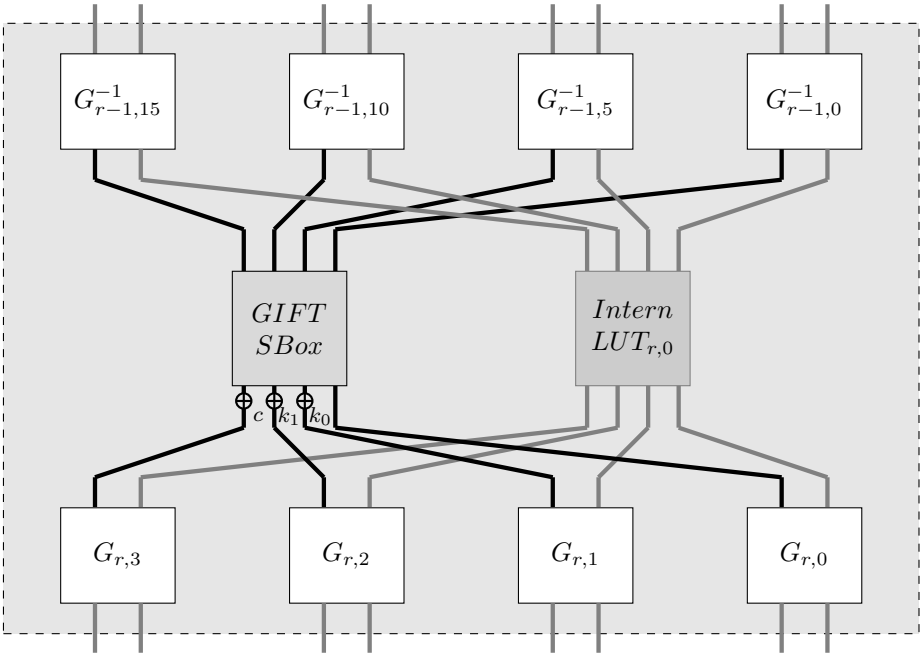


Figure 1: T_0, the right-most TBox of round r

The eight output bits of each TBox are split into four 2-bit blocks, each one of them being encoded by a randomly-chosen 2-bit permutation $G_{r,i}$. As a result, each 2-bit input block is decoded by the inverse $G^{-1}_{r-1,j}$ of the corresponding output encoding of the previous round.

In each 2-bit block of the input, once decoded, the left bit corresponds to a bit of the GIFT state, while the right one goes as one of the four input bits of

a randomly-chosen 4-bit bijective SBox $ILUT$.

Thus, on the one hand, the four decoded input blocks left bits go through the GIFT SBox GS (SubCells stage) and are then XORed to the key bits (k_1, k_0) and round constant bit c (AddRoundKey stage). On the other hand, the four decoded input blocks right bits go through the intern SBox $ILUT$. Then, similarly as for the input blocks, the left bit of each 2-bit output block is an output bit of GS and corresponds to a state bit of the actual GIFT computation, while the right bit is an output bit of $ILUT$ and can be considered as a pseudo-random bit. As a conclusion, each of these 2-bit output blocks can be considered as pseudo-random as it corresponds to the output of a pseudo-random encoding computation, with half of its input being pseudo-random as well.

Finally, the PermBits step remains the same as in the original implementation of GIFT, except for the fact that the permutation is now applied to the two-bit output blocks instead of single bits.

7.3 Design Rationale

7.3.1 The Output Bits of the GIFT SBox Cannot Be Encoded Altogether

During an execution of GIFT, we cannot encode if only two out of the four output bits of an instance of the GIFT SBox GS within the same function, as they will map to different instances of GS in the following round due to the GIFT permutation.

Furthermore, if we apply a 4-bit encoding to the 4-bit output of the GIFT SBox GS in a round r, it would force to decode those whole four bits before each GS instance of the next round $r + 1$ that take as input a bit out of these four. Indeed, the GIFT permutation PB was designed to ensure that the four output bits of the SBox are linked to four different instances of the SBox in the following round, and the 4-bit encoding applied implies that to recover one of these bits it is mandatory to recover the whole four. Therefore, decoding and recovering the four needed input bits of an instance of GS implies to decode four blocks of four bits independently. Moreover, each 4-bit encoded output of an instance of GS would need to be transmitted to each of the four next round instances of GS taking one of its bits as input. Consequently, the resulting TBoxes would have 16 input bits. The weight of a n-bit to m-bit look-up table being $2^n * m$ bits, it would imply a TBox weight of (at very least) 2^{16} bits, which is already too heavy regardless of the output size of the TBox.

In the case where we would split the 4 output bits of the SBox GS into two blocks of two bits and apply 2-bit encodings to each of these blocks, this encoding repartition would provide a differential attacker with information on the SBox GS inputs. Indeed, in this case, the attacker will be able to observe if modifying an input block to the SBox GS will impact the left half, the right half or the whole encoded output. As the value of GS is known to the attacker, he will easily be able to break the encodings.

7.3.2 Each Output Bit of the GIFT SBox Is 2-Bit Encoded

As stated above (subsubsection 7.3.1), it is not feasible to encode the output bits of an instance of the GIFT SBox altogether. For this reason, we decide to encode each of these four output bits separately, firstly by 2-bit encodings.

To ensure correctness throughout the different rounds, these 2-bit output encodings need to be inversed at the beginning of corresponding TBoxes of the following round. To that end, we use 2-bit input to 2-bit output encodings, and to obtain the extra pseudo-random input bit of each of those 2-bit encodings, we introduce the 4-bit pseudo-random SBox $ILUT$, that is bijective and non-linear. Thus, each TBox consists of 8 input bits and 8 output bits.

Once the 8-bit TBox input is decoded, each left bit of the four 2-bit blocks represents the state of GIFT, while the four remaining bits (the right bits of the 2-bit blocks) are pseudo-random bits. These four pseudo-random bits are mixed together thanks to the randomly-chosen bijective SBox $ILUT$ to complexify differential attacks. Indeed, a single bit modification of the 8-bit input of a TBox will have an impact on the 2-bit output of the decoding of the corresponding 2-bit input block. Thus, at least one bit of the state or one pseudo-random bit will be modified. If a state bit is modified, then the 4-bit output of the GIFT SBox GS will be modified and if a pseudo-random bit is modified then the 4-bit output of $ILUT$ will be modified. Hence, the overall 8-bit output of the TBox will be modified, and not only the 2-bit output block with the same index as the modified 2-bit input block. However, we will demonstrate in subsection 8.2 that the knowledge of the GIFT SBox GS leads to an differential 2-round attack.

Moreover, the randomly-chosen intern lookup table complicates a brute force attack. Indeed, there exists $(2^n)!$ different n-bit lookup tables, so the 4-bit intern lookup table brings $(2^4)! \approx 2^{44}$ possibilities. A brute force attack would also require to go through all the possible 2-bit lookup tables for each of the four input decodings and four output encodings, as well as for the four possible key values, and this, for each 256 inputs. Overall, this attack has to enumerate around 2^{90} possibilities.

We have chosen the 2-bit encoding design as it was the lightest, but heavier encodings are also an option to enhance the brute-force attack resistance. Replacing the two middle 2-bit encodings and decodings by a 3-bit version also allows to have a 6-bit intern lookup table (around 2^{296} possibilities), raising the total cost of a brute force attack to around 2^{385} possibilities.

7.4 The Problem of the First and Last Round

Our white-box implementation implements GIFT with encoded inputs and outputs. To keep correctness with raw inputs, its first round input decodings and last round output encodings must be removed. At this point, in each TBox of the last round, the attacker would have access to the two output bits of the GIFT SBox GS that are not to be XORed with (unknown) round key bits. Indeed, we consider the round constant XORed to the left output bit of GS to be known, we can then recover the plain value of this bit.

Thus, by brute-force, the attacker could recover the two other GS output bits, thus breaking all the TBoxes of the last round. This comprises the recovery of the input encodings of these TBoxes, that are inverses of the output encodings of the TBoxes of the penultimate round. These output encodings could hence be recovered and we could apply the same method to break the TBoxes of the penultimate round. Therefore, we could break the TBoxes round by round from the last round up until the first, thus leading to the break of the encoding scheme.

To avoid this kind of attacks, we suppose that GIFT is operating with encoded inputs and outputs. As a matter of fact, this is a use case common for instance in the DRM field.

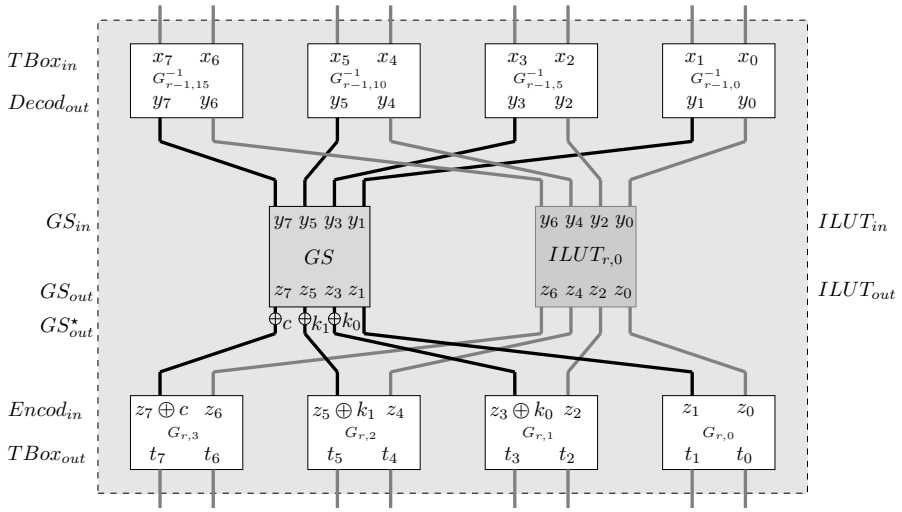


Figure 2: Intermediate notations of T_0, the right-most TBox of round r

8 Differential Attack and Evolution Perspectives

In this section, we first detail how, even in the case where $ILUT$ is non linear, the attacker knowledge of the GIFT SBox GS results in the possibility of a differential attack targeting two consecutive rounds of TBoxes. Then, we demonstrate that this attack can be simplified if $ILUT$ is linear. Finally, we look for the existence of a SBox complying with the GIFT SBox properties while preventing this 2-round differential attack. To that end, we use the notations detailed in subsection 8.1.

8.1 Notations

To investigate the differential properties of a TBox, we use the following notations, illustrated in Figure 2 with T_0, the right-most TBox of a round r. Additionally, we note that the reasoning developed in this section is valid for any TBox of any round but the last, as TBoxes of the following round will need to be considered.

The intermediate values of the TBox can be detailed as :

- $TBox_{in} = (x_7x_6\ x_5x_4\ x_3x_2\ x_1x_0)$ the input of the TBox.
- $Decod_{out} = (y_7y_6\ y_5y_4\ y_3y_2\ y_1y_0) = (G^{-1}_{r-1,15}(x_7x_6)\ G^{-1}_{r-1,10}(x_5x_4)\ G^{-1}_{r-1,5}(x_3x_2)\ G^{-1}_{r-1,0}(x_1x_0))$ the output of the input decodings.
- $GS_{in} = (y_7y_5y_3y_1)$ the input of the SBox GS.
- $GS_{out} = (z_7z_5z_3z_1) = GS[y_7y_5y_3y_1]$ the output of the SBox GS.

- $GS_{out}{}^* = ((z_7 \oplus c)(z_5 \oplus k_1)(z_3 \oplus k_0)z_1)$ the output of the SBox GS, XORed with c the round constant bit and (k_1, k_0) the key bits.
- $ILUT_{in} = (y_6 y_4 y_2 y_0)$ the input of the intern SBox $ILUT$.
- $ILUT_{out} = (z_6 z_4 z_2 z_0) = ILUT[y_6 y_4 y_2 y_0]$ the output of the intern SBox $ILUT$.
- $Encod_{in} = ((z_7 \oplus c)z_6\ (z_5 \oplus k_1)z_4\ (z_3 \oplus k_0)z_2\ z_1 z_0)$ the input of the output encodings.
- $TBox_{out} = (t_7 t_6\ t_5 t_4\ t_3 t_2\ t_1 t_0) = (G_{r,3}((z_7 \oplus c)z_6)\ G_{r,2}((z_5 \oplus k_1)z_4)\ G_{r,1}((z_3 \oplus k_0)z_2)\ G_{r,0}(z_1 z_0))$ the output of the TBox.

8.2 Differential Attack

The 2-round differential attack can be decomposed into three major steps : first of all, we show that the left bit of each input encoding of the TBox T_0 can be recovered, i.e. for each input encoding $G^{-1}_{r-1,j}$ and each $(x_l, x_m) \in \mathbb{F}_2^2$, we recover $y_l \in \mathbb{F}_2$ such that $G^{-1}_{r-1,j}(x_l x_m) = y_l \_$. The second step applies the same principle to the TBoxes of the following round to recover the left bits of their input encodings, and especially to the left bits of the input encodings that are inverses of the output encodings of T_0. This implies that, for each output encoding $G_{r,k}$ of T_0 and each $(t_l, t_m) \in \mathbb{F}_2^2$, we can recover $z_l^\star \in \mathbb{F}_2$ such that $G_{r,k}(z_1^\star \_) = t_l t_m$. Finally, the last step uses this knowledge about the encodings of T_0 and the knowledge of the GIFT SBox GS to recover k_0 and k_1, the key bits embedded in T_0.

8.2.1 First Step Of The Attack

As detailed hereinabove, to perform our attack, we first choose $(x_7, x_6, x_5, x_4, x_3, x_2) \in \mathbb{F}_2^6$ and note that

- $G^{-1}_{r-1,15}(x_7 x_6) = y_7 y_6$
- $G^{-1}_{r-1,10}(x_5 x_4) = y_5 y_4$
- $G^{-1}_{r-1,5}(x_3 x_2) = y_3 y_2$

As $G^{-1}_{r-1,0}$ is a 2-bit encoding, it can be described as a bijective endomorphism of $\mathbb{F}_2^2$. Thus, for every $(y_1, y_0) \in \mathbb{F}_2^2$, there exists a unique input $(x_1^{(y_1,y_0)}, x_0^{(y_1,y_0)}) \in \mathbb{F}_2^2$ such that $G^{-1}_{r-1,0}(x_1^{(y_1,y_0)} x_0^{(y_1,y_0)}) = (y_1, y_0)$.

For example, there exists $(x_1^{(0,0)}, x_0^{(0,0)}) \in \mathbb{F}_2^2$ such that the computation of $T_0[x_7 x_6\ x_5 x_4\ x_3 x_2\ x_1^{(0,0)} x_0^{(0,0)}]$ entails $Decod_{out} = (y_7 y_6 y_5 y_4 y_3 y_2 00)$ and more broadly the intermediate values listed in Figure 3.

Likewise, there exists $(x_1^{(0,1)}, x_0^{(0,1)}) \in \mathbb{F}_2^2$ such that the output of the input decodings is $Decod_{out} = (y_7 y_6\ y_5 y_4\ y_3 y_2\ 01)$. It then implies that the GIFT SBox

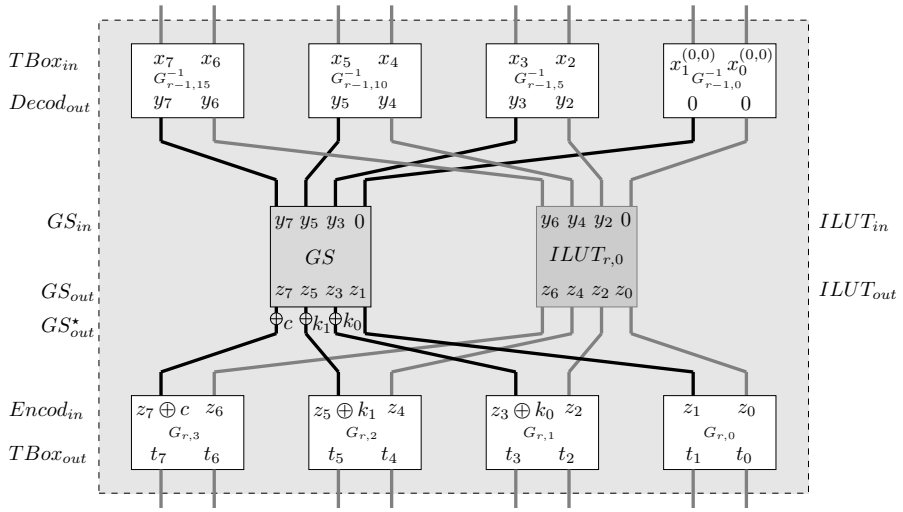


Figure 3: Example of intermediate values of T_0

input GS_{in} is $(y_7y_5y_30)$ and the $ILUT$ input $ILUT_{in}$ is $(y_6y_4y_21)$. Furthermore, we note the $ILUT$ output $ILUT_{out} = ILUT[y_6y_4y_21]$ as $(z'_6z'_4z'_2z'_0)$. This entails that the inputs of the output encodings $Encod_{in}$ are $((z_7 \oplus c)z'_6\ (z_5 \oplus k_1)z'_4\ (z_3 \oplus k_0)z'_2\ z_1z'_0)$, as summed up in Figure 4.

More broadly, on the one hand, when we compute $T_0[x_7x_6x_5x_4x_3x_2x_1^{(y_1,y_0)}x_0^{(y_1,y_0)}]$ for all $(y_1, y_0) \in \mathbb{F}_2^2$, the intermediate values $Decod_{out}$ obtained after the input decodings are :

- $(y_7y_6\ y_5y_4\ y_3y_2\ 00)$
- $(y_7y_6\ y_5y_4\ y_3y_2\ 10)$
- $(y_7y_6\ y_5y_4\ y_3y_2\ 01)$
- $(y_7y_6\ y_5y_4\ y_3y_2\ 11)$

Furthermore, we note

- $GS[y_7y_5y_30] = (z_7z_5z_3z_1)$
- $ILUT[y_6y_4y_20] = (z_6z_4z_2z_0)$
- $GS[y_7y_5y_31] = (z'_7z'_5z'_3z'_1)$
- $ILUT[y_6y_4y_21] = (z'_6z'_4z'_2z'_0)$

The intermediate values $Encod_{in}$, inputs of output encodings for the computations of the $T_0[x_7x_6x_5x_4x_3x_2x_1^{(y_1,y_0)}x_0^{(y_1,y_0)}]$ are thus

- $((z_7 \oplus c)z_6\ (z_5 \oplus k_1)z_4\ (z_3 \oplus k_0)z_2\ z_1z_0)$ for $(y_1, y_0) = (0, 0)$
- $((z_7 \oplus c)z'_6\ (z_5 \oplus k_1)z'_4\ (z_3 \oplus k_0)z'_2\ z_1z'_0)$ for $(y_1, y_0) = (0, 1)$
- $((z'_7 \oplus c)z_6\ (z'_5 \oplus k_1)z_4\ (z'_3 \oplus k_0)z_2\ z'_1z_0)$ for $(y_1, y_0) = (1, 0)$
- $((z'_7 \oplus c)z'_6\ (z'_5 \oplus k_1)z'_4\ (z'_3 \oplus k_0)z'_2\ z'_1z'_0)$ for $(y_1, y_0) = (1, 1)$

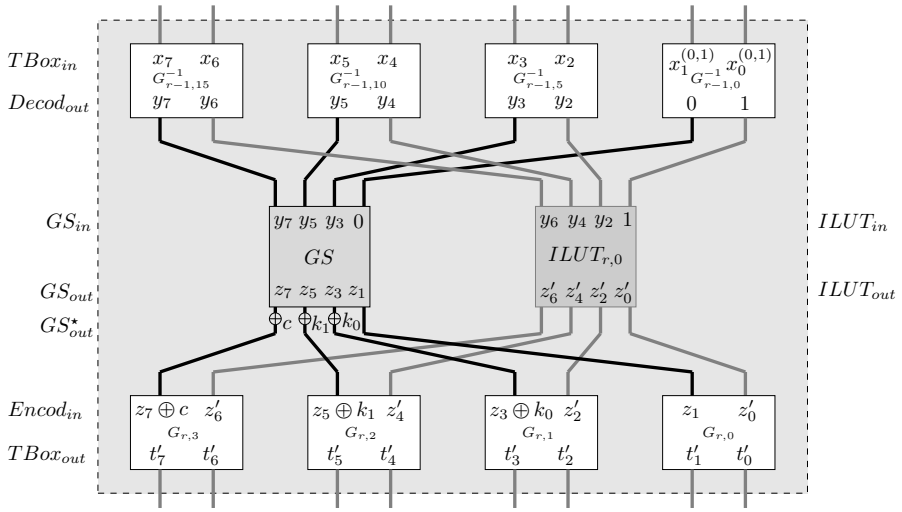


Figure 4: Second example of intermediate values of T_0

On the other hand, in the first place, we can compute the values of $\mathcal{T} = \{T_0[x_7x_6\ x_5x_4\ x_3x_2\ x_1x_0] \mid (x_1, x_0) \in \mathbb{F}_2^2\}$. These values can also be written as

- $(G_{r,3}((z_7 \oplus c)z_6)\, G_{r,2}((z_5 \oplus k_1)z_4)G_{r,1}((z_3 \oplus k_0)z_2)\, G_{r,0}(z_1z_0))$
- $(G_{r,3}((z_7 \oplus c)z'_6)\, G_{r,2}((z_5 \oplus k_1)z'_4)G_{r,1}((z_3 \oplus k_0)z'_2)\, G_{r,0}(z_1z'_0))$
- $(G_{r,3}((z'_7 \oplus c)z_6)\, G_{r,2}((z'_5 \oplus k_1)z_4)G_{r,1}((z'_3 \oplus k_0)z_2)\, G_{r,0}(z'_1z_0))$
- $(G_{r,3}((z'_7 \oplus c)z'_6)\, G_{r,2}((z'_5 \oplus k_1)z'_4)G_{r,1}((z'_3 \oplus k_0)z'_2)\, G_{r,0}(z'_1z'_0))$

For each of the elements listed above, we cannot determine the corresponding (x_1, x_0). For example, we cannot determine $(x_1, x_0) \in \mathbb{F}_2^2$ such that $T_0[x_7x_6\, x_5x_4\, x_3x_2\, x_1x_0] = (G_{r,3}((z_7 \oplus c)z'_6)\, G_{r,2}((z_5 \oplus k_1)z'_4)G_{r,1}((z_3 \oplus k_0)z'_2)\, G_{r,0}(z_1z'_0))$, we only know that the value of $(G_{r,3}((z_7 \oplus c)z'_6)\, G_{r,2}((z_5 \oplus k_1)z'_4)G_{r,1}((z_3 \oplus k_0)z'_2)\, G_{r,0}(z_1z'_0))$ figures in the set $\mathcal{T}$. Nevertheless, we demonstrate that information about the encodings can still be deduced from the values of $\mathcal{T}$.

Indeed, as $G_{r,3}$ is a bijection,

- if $G_{r,3}((z_7 \oplus c)z_6) = G_{r,3}((z_7 \oplus c)z'_6) = G_{r,3}((z'_7 \oplus c)z_6) = G_{r,3}((z'_7 \oplus c)z'_6)$, then $(z_7 \oplus c)z_6 = (z_7 \oplus c)z'_6 = (z'_7 \oplus c)z_6 = (z'_7 \oplus c)z'_6$, i.e. $z_7 = z'_7$ and $z_6 = z'_6$.
- if $G_{r,3}((z_7 \oplus c)z_6) \neq G_{r,3}((z_7 \oplus c)z'_6) \neq G_{r,3}((z'_7 \oplus c)z_6) \neq G_{r,3}((z'_7 \oplus c)z'_6)$, then $(z_7 \oplus c)z_6 \neq (z_7 \oplus c)z'_6 \neq (z'_7 \oplus c)z_6 \neq (z'_7 \oplus c)z'_6$, i.e. $z_7 \neq z'_7$ and $z_6 \neq z'_6$.
- if $\#\{G_{r,3}((z_7 \oplus c)z_6), G_{r,3}((z_7 \oplus c)z'_6), G_{r,3}((z'_7 \oplus c)z_6), G_{r,3}((z'_7 \oplus c)z'_6)\} = 2$, then $\#\{((z_7 \oplus c)z_6), ((z_7 \oplus c)z'_6), ((z'_7 \oplus c)z_6), ((z'_7 \oplus c)z'_6\} = 2$. Consequently, either $z_7 = z'_7$ and $z_6 \neq z_6$ or $z_7 \neq z'_7$ and $z_6 = z_6$.

The numerical value of the GIFT SBox GS is known (see subsection 7.1), and by notation $GS[y_7y_5y_30] = z_7z_5z_3z_1$ and $GS[y_7y_5y_31] = z'_7z'_5z'_3z'_1$. Moreover, if for example $z_7 = z'_7$, then $GS[y_7y_5y_31]$ can be noted as $z_7z'_5z'_3z'_1$. Hence, in this case :

$$\begin{cases} GS[y_7y_5y_30] = z_7z_5z_3z_1 \\ GS[y_7y_5y_31] = z_7z'_5z'_3z'_1 \end{cases} \Rightarrow (y_7, y_5, y_3) = (1,1,0) \text{ or } (1,1,1)$$

$$\Rightarrow (y_7, y_5) = (1,1)$$

$$\Rightarrow \begin{cases} G^{-1}_{r-1,15}(x_7x_6) = 1\_ \\ G^{-1}_{r-1,10}(x_5x_4) = 1\_ \end{cases}$$

Note that we can also set the values of $(x_7, x_6, x_5, x_4, x_1, x_0)$ and vary (x_3, x_2) to obtain information about $G^{-1}_{r-1,15}, G^{-1}_{r-1,10}$ and $G^{-1}_{r-1,0}$, and keep the same principle when varying (x_7, x_6) or (x_5, x_4).

To determine the value of the two key bits of this TBox T_0 of round r, the previous procedure needs to be applied in order to obtain the left bit of each of its four input encodings.

8.2.2 Second Step of the Attack : Consider TBoxes of Following Round

This first step can also be applied on the corresponding TBoxes of following round $r+1$, that admit $G^{-1}_{r,3}$, $G^{-1}_{r,2}$, $G^{-1}_{r,1}$ or $G^{-1}_{r,0}$ as one of their input decodings. Hence, for all $(a,b) \in \mathbb{F}_2^2$, we can recover $c_3, c_2, c_1, c_0 \in \mathbb{F}_2$ such that

- $G^{-1}_{r,3}(ab) = c_3\_$
- $G^{-1}_{r,2}(ab) = c_2\_$
- $G^{-1}_{r,1}(ab) = c_1\_$
- $G^{-1}_{r,0}(ab) = c_0\_$

Therefore, regarding the inverses of these encodings $G_{r,3}, G_{r,2}, G_{r,1}, G_{r,0}$ that are the output encodings of the TBox T_0, we can for all $(t,t') \in \mathbb{F}_2^2$ recover $\tilde{z}_7, \tilde{z}_5, \tilde{z}_3, \tilde{z}_1 \in \mathbb{F}_2$ such that $G_{r,3}(\tilde{z}_7\_) = tt', G_{r,2}(\tilde{z}_5\_) = tt', G_{r,1}(\tilde{z}_3\_) = tt'$ and $G_{r,0}(\tilde{z}_1\_) = tt'$.

8.2.3 Final Step Of The Attack

As the left bits of the TBox T_0 input encodings $G^{-1}_{r-1,15}, G^{-1}_{r-1,10}, G^{-1}_{r-1,5}$ and $G^{-1}_{r-1,0}$ are known, for any possible TBox input $TBox_{in} = (x_7, x_6, x_5, x_4, x_3, x_2, x_1, x_0) \in \mathbb{F}_2^6$ the corresponding intermediate GIFT SBox input $GS_{in} = (y_7y_5y_3y_1)$ and thus GIFT SBox output $GS_{out} = GS[y_7y_5y_3y_1] = (z_7z_5z_3z_1)$ can be deduced.

On the other hand, we can compute $TBox_{out} = (t_7t_6\, t_5t_4\, t_3t_2\, t_1t_0) = T_0[x_7x_6\, x_5x_4\, x_3x_2\, x_1x_0]$. Since the left bits of $G^{-1}_{r,3}$, $G^{-1}_{r,2}$, $G^{-1}_{r,1}$ and $G^{-1}_{r,0}$ are known as detailed in subsubsection 8.2.2, the values of $(z_7 \oplus c), (z_5 \oplus k_1), (z_3 \oplus k_0)$ and $z_1 \in \mathbb{F}_2$ can be determined such that

$$\begin{cases} G_{r,3}^{-1}(t_7t_6) = (z_7 \oplus c)\_ \\ G_{r,2}^{-1}(t_5t_4) = (z_5 \oplus k_1)\_ \\ G_{r,1}^{-1}(t_3t_2) = (z_3 \oplus k_0)\_ \\ G_{r,0}^{-1}(t_1t_0) = z_1\_ \end{cases}, \text{ i.e. } \begin{cases} G_{r,3}((z_7 \oplus c)\_) = t_7t_6 \\ G_{r,2}((z_5 \oplus k_1)\_) = t_5t_4 \\ G_{r,1}((z_3 \oplus k_0)\_) = t_3t_2 \\ G_{r,0}(z_1\_) = t_1t_0 \end{cases}$$

Hence, $GS_{out}{}^* = ((z_7 \oplus c)(z_5 \oplus k_1)(z_3 \oplus k_0)z_1)$ is also known. In conclusion, for any TBox input $TBox_{in} = (x_7, x_6, x_5, x_4, x_3, x_2, x_1, x_0) \in \mathbb{F}_2^6$ we can determine both $GS_{out} = (z_7z_5z_3z_1)$ and $GS_{out}{}^* = ((z_7 \oplus c)(z_5 \oplus k_1)(z_3 \oplus k_0)z_1)$. Thus, the value of key bits (k_1, k_0) can be deduced from the values of $(z_5 \oplus k_1), (z_3 \oplus k_0), z_5$ and z_3.

A numerical example of the major steps of the attack can be found in Appendix A.

8.3 Intern SBox ILUT and Differential Properties

8.3.1 Using a Linear Intern SBox ILUT Leads to its Recovery

In this subsection we suppose that we use in the TBox T_0 an intern linear SBox $ILUT$. Thus, as an example, when noting $(v_6, v_4, v_2, v_0) = ILUT[0001]$ then for all $(a, b, c, d) \in \mathbb{F}_2^4$ verifying $ILUT[abcd] = efgh \in \mathbb{F}_2^4$,

$$ILUT[abc\overline{d}] = ILUT[abcd] \oplus ILUT[0001] = efgh \oplus v_6v_4v_2v_0.$$

Subsequently, let $(x_7, x_6, x_5, x_4, x_3, x_2) \in \mathbb{F}_2^6$. We note

$$\begin{cases} G_{r-1,15}^{-1}(x_7x_6) = y_7y_6 \\ G_{r-1,10}^{-1}(x_5x_4) = y_5y_4 \\ G_{r-1,5}^{-1}(x_3x_2) = y_3y_2 \end{cases}$$

We then compute successively $T_0[x_7x_6x_5x_4x_3x_200]$, $T_0[x_7x_6x_5x_4x_3x_201]$, $T_0[x_7x_6x_5x_4x_3x_210]$ and $T_0[x_7x_6x_5x_4x_3x_211]$. As detailed above in the general case of the attack, this implies that among these four computations the output bits $(y_1, y_0) \in \mathbb{F}_2^2$ of the right-most input encoding $G_{r-1,0}^{-1}$ take alternatively (but not necessarily respectively) the four values $00, 01, 10$ and 11 of $\mathbb{F}_2^2$. Therefore, the inputs and outputs of the GIFT SBox GS and the intern SBox $ILUT$ are :

- When $G_{r-1,0}^{-1}(x_1x_0) = 00$,

$$(z_7z_5z_3z_1) = GS[y_7y_5y_30] \text{ and } (z_6z_4z_2z_0) = ILUT[y_6y_4y_20]$$

- When $G_{r-1,0}^{-1}(x_1x_0) = 01$,

$$z_7z_5z_3z_1 = GS[y_7y_5y_30] \text{ and } z'_6z'_4z'_2z'_0 = ILUT[y_6y_4y_21] \\ = ILUT[y_6y_4y_20] \oplus ILUT[0001] \\ = z_6z_4z_2z_0 \oplus v_6v_4v_2v_0 \\ = (z_6 \oplus v_6)(z_4 \oplus v_4)(z_2 \oplus v_2)(z_0 \oplus v_0)$$

- When $G_{r-1,0}^{-1}(x_1x_0) = 10$,

$$z'_7z'_5z'_3z'_1 = GS[y_7y_5y_31] \text{ and } z_6z_4z_2z_0 = ILUT[y_6y_4y_20]$$

- When $G_{r-1,0}^{-1}(x_1x_0) = 11$,

$$z'_7z'_5z'_3z'_1 = GS[y_7y_5y_31] \text{ and } z'_6z'_4z'_2z'_0 = ILUT[y_6y_4y_21] \\ = ILUT[y_6y_4y_20] \oplus ILUT[0001] \\ = z_6z_4z_2z_0 \oplus v_6v_4v_2v_0 \\ = (z_6 \oplus v_6)(z_4 \oplus v_4)(z_2 \oplus v_2)(z_0 \oplus v_0)$$

Thus, the different outputs of the TBox are

- $G_{r,3}(z_7z_6)G_{r,2}(z_5z_4)G_{r,1}(z_3z_2)G_{r,0}(z_1z_0)$
- $G_{r,3}(z_7(z_6 \oplus v_6))G_{r,2}(z_5(z_4 \oplus v_4))G_{r,1}(z_3(z_2 \oplus v_2))G_{r,0}(z_1(z_0 \oplus v_0))$
- $G_{r,3}(z'_7z_6)G_{r,2}(z'_5z_4)G_{r,1}(z'_3z_2)G_{r,0}(z'_1z_0)$
- $G_{r,3}(z'_7(z_6 \oplus v_6))G_{r,2}(z'_5(z_4 \oplus v_4))G_{r,1}(z'_3(z_2 \oplus v_2))G_{r,0}(z'_1(z_0 \oplus v_0))$

By computing sets of values $\{T_0[x_7x_6x_5x_4x_3x_200], T_0[x_7x_6x_5x_4x_3x_201], T_0[x_7x_6x_5x_4x_3x_210], T_0[x_7x_6x_5x_4x_3x_211]\}$ for $(x_7, x_6, x_5, x_4, x_3, x_2) \in \mathbb{F}_2^6$, it is possible to determine the value $v_6v_4v_2v_0 = ILUT[0001]$. First of all, regarding the value of v_6 :

- If $v_6 = 0$ then $\{G_{r,3}(z_7z_6), G_{r,3}(z_7(z_6 \oplus v_6)), G_{r,3}(z'_7z_6), G_{r,3}(z'_7(z_6 \oplus v_6))\} = \{G_{r,3}(z_7z_6), G_{r,3}(z_7z_6), G_{r,3}(z'_7z_6), G_{r,3}(z'_7z_6)\}$. Thus,
  - if $z'_7 = z_7$ then $\#\{G_{r,3}(z_7z_6), G_{r,3}(z_7z_6), G_{r,3}(z_7z_6), G_{r,3}(z_7z_6)\} = 1$
  - if $z'_7 = \overline{z_7}$ then $\#\{G_{r,3}(z_7z_6), G_{r,3}(z_7z_6), G_{r,3}(\overline{z_7}z_6), G_{r,3}(\overline{z_7}z_6)\} = 2$
- If $v_6 = 1$ then $\{G_{r,3}(z_7z_6), G_{r,3}(z_7(z_6 \oplus v_6)), G_{r,3}(z'_7z_6), G_{r,3}(z'_7(z_6 \oplus v_6))\} = \{G_{r,3}(z_7z_6), G_{r,3}(z_7\overline{z_6}), G_{r,3}(z'_7z_6), G_{r,3}(z'_7\overline{z_6})\}$. Thus,
  - if $z'_7 = z_7$ then $\#\{G_{r,3}(z_7z_6), G_{r,3}(z_7\overline{z_6}), G_{r,3}(z_7z_6), G_{r,3}(z_7\overline{z_6})\} = 2$
  - if $z'_7 = \overline{z_7}$ then $\#\{G_{r,3}(z_7z_6), G_{r,3}(z_7\overline{z_6}), G_{r,3}(\overline{z_7}z_6), G_{r,3}(\overline{z_7z_6})\} = 4$

Therefore, if there exists $(x_7, x_6, x_4, x_3, x_2) \in \mathbb{F}_2^6$ such that the left-most 2-bit output block $G_{r,3}(z_7z_6)$ takes the same value in $\mathbb{F}_2^2$ for the four computations of $T_0[x_7x_6x_5x_4x_3x_200]$, $T_0[x_7x_6x_5x_4x_3x_201]$, $T_0[x_7x_6x_5x_4x_3x_210]$ and $T_0[x_7x_6x_5x_4x_3x_211]$, then $v_6 = 0$. Likewise, if this 2-bit output block takes four different values, then $v_6 = 1$. Thus, as v_6 is known to the attacker, he can for a given $(x_7, x_6, x_5, x_4, x_3, x_2) \in \mathbb{F}_2^6$ determine whether $z'_7 = z_7$ or $z'_7 = \overline{z_7}$, i.e., for $y_7y_6 = G^{-1}_{r-1,15}(x_7x_6)$, $y_5y_4 = G^{-1}_{r-1,10}(x_5x_4)$, $y_3y_2 = G^{-1}_{r-1,5}(x_3x_2)$ and $GS[y_7y_5y_30] = z_7z_5z_3z_1$, whether

$$GS[y_7y_5y_31] = z_7z'_5z'_3z'_1 \text{ or } GS[y_7y_5y_31] = \overline{z_7}z'_5z'_3z'_1$$

In the same manner, v_4 can be recovered considering the set of values of the second left-most 2-bit output block $\{G_{r,2}((z_5 \oplus k_0)z_4), G_{r,2}((z_5 \oplus k_0)(z_4 \oplus v_4)), G_{r,2}((z'_5 \oplus k_0)z_4), G_{r,2}((z'_5 \oplus k_0)(z_4 \oplus v_4))\}$. This can be applied to the recovery of v_4, v_2 and v_0 such that it is possible for an attacker to determine $v_6v_4v_2v_0 = ILUT[0001]$.

Likewise, it is possible for an attacker to determine the values of $ILUT[0010]$, $ILUT[0100]$ and $ILUT[1000]$, and subsequently all values of $ILUT$, as for all $(y_6, y_4, y_2, y_0) \in \mathbb{F}_2^4$, $ILUT[y_6y_4y_2y_0] = (y_6 * ILUT[1000]) \oplus (y_4 * ILUT[0100]) \oplus (y_2 * ILUT[0010]) \oplus (y_0 * ILUT[0001])$.

8.3.2 The Knowledge of the Intern SBox ILUT Simplies the 2-Round Differential Attack

We demonstrate that the knowledge of the intern SBox $ILUT$ simplifies the recovery of the left bit of the input encodings of the TBox T_0. For that purpose, we still suppose $(x_7, x_6, x_5, x_4, x_3, x_2) \in \mathbb{F}_2^6$ with $G^{-1}_{r-1,3}(x_7x_6) = y_7y_6$, $G^{-1}_{r-1,2}(x_5x_4) = y_5y_4$ and $G^{-1}_{r-1,1}(x_3x_2) = y_3y_2$, and more broadly retain the notations developed in subsubsection 8.2.1.

In the generic setting of the attack described in subsection 8.2, when computing $T_0[x_7x_6x_5x_4x_3x_200]$, $T_0[x_7x_6x_5x_4x_3x_201]$, $T_0[x_7x_6x_5x_4x_3x_210]$ and $T_0[x_7x_6x_5x_4x_3x_211]$, the different input arrays of output encodings $Encod_{in}$ are noted :

- $((z_7 \oplus c)z_6\ (z_5 \oplus k_1)z_4\ (z_3 \oplus k_0)z_2\ z_1z_0)$ for $(y_1, y_0) = (0, 0)$
- $((z_7 \oplus c)z'_6\ (z_5 \oplus k_1)z'_4\ (z_3 \oplus k_0)z'_2\ z_1z'_0)$ for $(y_1, y_0) = (0, 1)$
- $((z'_7 \oplus c)z_6\ (z'_5 \oplus k_1)z_4\ (z'_3 \oplus k_0)z_2\ z'_1z_0)$ for $(y_1, y_0) = (1, 0)$
- $((z'_7 \oplus c)z'_6\ (z'_5 \oplus k_1)z'_4\ (z'_3 \oplus k_0)z'_2\ z'_1z'_0)$ for $(y_1, y_0) = (1, 1)$

with

$$\begin{cases} GS[y_7y_5y_30] = z_7z_5z_3z_1 \\ GS[y_7y_5y_31] = z'_7z'_5z'_3z'_1 \end{cases} \text{ and } \begin{cases} ILUT[y_6y_4y_20] = z_6z_4z_2z_0 \\ ILUT[y_6y_4y_21] = z'_6z'_4z'_2z'_0 \end{cases}$$

Thus, regarding the value of the first 2-bit output block in the four computations,

- If $\#\{G_{r-1,3}^{-1}((z_7 \oplus c)z_6), G_{r-1,3}^{-1}((z_7 \oplus c)z_6'), G_{r-1,3}^{-1}((z_7' \oplus c)z_6), G_{r-1,3}^{-1}(z_7' \oplus c)z_6'\} = 4$, then $\#\{(z_7 \oplus c)z_6, (z_7 \oplus c)z_6', (z_7' \oplus c)z_6, (z_7' \oplus c)z_6'\} = 4$, thus $z_7' = \overline{z_7}$ and $z_6' = \overline{z_6}$.
- If $\#\{G_{r-1,3}^{-1}((z_7 \oplus c)z_6), G_{r-1,3}^{-1}((z_7 \oplus c)z_6'), G_{r-1,3}^{-1}((z_7' \oplus c)z_6), G_{r-1,3}^{-1}(z_7' \oplus c)z_6'\} = 1$, then $\#\{(z_7 \oplus c)z_6, (z_7 \oplus c)z_6', (z_7' \oplus c)z_6, (z_7' \oplus c)z_6'\} = 1$, thus $z_7' = z_7$ and $z_6' = z_6$.
- If $\#\{G_{r-1,3}^{-1}((z_7 \oplus c)z_6), G_{r-1,3}^{-1}((z_7 \oplus c)z_6'), G_{r-1,3}^{-1}((z_7' \oplus c)z_6), G_{r-1,3}^{-1}(z_7' \oplus c)z_6'\} = 2$, it cannot be determined whether $z_7' = \overline{z_7}$ and $z_6' = z_6$ or $z_7' = z_7$ and $z_6' = \overline{z_6}$.

If $ILUT$ is known, noting $ILUT[0001] = v_6v_4v_2v_0$ implies that

$$\begin{cases} ILUT[y_6y_4y_20] = z_6z_4z_2z_0 \\ ILUT[y_6y_4y_21] = (z_6 \oplus v_6)(z_4 \oplus v_4)(z_2 \oplus v_2)(z_0 \oplus v_0) \end{cases}$$

Thus, the different input arrays of output encodings $Encod_{in}$ can be written as :

- $((z_7 \oplus c)z_6\ (z_5 \oplus k_1)z_4\ (z_3 \oplus k_0)z_2\ z_1z_0)$ for $(y_1, y_0) = (0, 0)$
- $((z_7 \oplus c)(z_6 \oplus v_6)\ (z_5 \oplus k_1)(z_4 \oplus v_4)\ (z_3 \oplus k_0)(z_2 \oplus v_2)\ z_1(z_0 \oplus v_0))$ for $(y_1, y_0) = (0, 1)$
- $((z_7' \oplus c)z_6\ (z_5' \oplus k_1)z_4\ (z_3' \oplus k_0)z_2\ z_1'z_0)$ for $(y_1, y_0) = (1, 0)$
- $((z_7' \oplus c)(z_6 \oplus v_6)\ (z_5' \oplus k_1)(z_4 \oplus v_4)\ (z_3' \oplus k_0)(z_2 \oplus v_2)\ z_1'(z_0 \oplus v_0))$ for $(y_1, y_0) = (1, 1)$

As a result, the third case where $\#\{G_{r-1,3}^{-1}((z_7 \oplus c)z_6), G_{r-1,3}^{-1}((z_7 \oplus c)z_6'), G_{r-1,3}^{-1}((z_7' \oplus c)z_6), G_{r-1,3}^{-1}(z_7' \oplus c)z_6'\} = 2$ can be rewritten as $\#\{G_{r-1,3}^{-1}((z_7 \oplus c)z_6), G_{r-1,3}^{-1}((z_7 \oplus c)(z_6 \oplus v_6)), G_{r-1,3}^{-1}((z_7' \oplus c)z_6), G_{r-1,3}^{-1}((z_7' \oplus c)(z_6 \oplus v_6))\} = 2$. As v_6 is known, it can be deduced whether $z_7' = z_7$ or $z_7' = \overline{z_7}$. At this point, the left bits of the input encodings of the TBox T_0 can be recovered by the 2-round attack detailed in subsection 8.2.

Nevertheless, we prove hereinbelow that the knowledge of the intern SBox $ILUT$ of the TBox T_0 (whether being deduced thanks to its linearity or not) does not enable an attacker to recover the key bits by a differential attack on T_0 only.

8.3.3 The Knowledge of the Intern SBox Does Not Lead to a Differential Attack on the TBox Itself

In this section, we suppose that in the set-up stated in subsection 8.2 the intern SBox $ILUT$ is known to the attacker. Therefore, the left bits of the input encodings of T_0 can be recovered as detailed in subsubsection 8.3.2.

Knowledge on the input left bits of output encodings of T_0 is essential for an attacker to determine with certainty its key bits, even if he has entirely broken the input encodings beforehand. Indeed, if we suppose that the attacker has effectively broken those input encodings, then when fixing the input of three 2-bit input blocks and computing the outputs of T_0 when varying the fourth 2-bit input block, he could effectively deduce the outputs of the GIFT SBox GS and the intern SBox $ILUT$.

Nevertheless, the key bits to be determined are added afterwards to the GS output bits. Given the four computed outputs of T_0, there exists for every possible key bits $(k_0, k_1) \in \mathbb{F}_2^2$ potential corresponding outputs encodings $G_{r,3}$, $G_{r,2}$, $G_{r,1}$ and $G_{r,0}$.

For example, we suppose $(x_5, x_4, x_3, x_2, x_1, x_0) \in \mathbb{F}_2^6$. As mentioned above, by hypothesis we assume that the attacker has entirely broken the input encodings (whether by brute force or not). Thus, the attacker knows the input encodings output values $G^{-1}_{r-1,10}(x_5x_4) = y_5y_4$, $G^{-1}_{r-1,5}(x_3x_2) = y_3y_2$ and $G^{-1}_{r-1,0}(x_1x_0) = y_1y_0$. Throughout the four computations of $T_0[00x_5x_4x_3x_2x_1x_0]$, $T_0[01x_5x_4x_3x_2x_1x_0]$, $T_0[10x_5x_4x_3x_2x_1x_0]$ and $T_0[11x_5x_4x_3x_2x_1x_0]$, the inputs and outputs of the GIFT SBox GS and the inputs and outputs of the intern SBox $ILUT$ are (not necessarily respectively)

- $GS[0y_5y_3y_1] = z_7z_5z_3z_1$ and $ILUT[0y_4y_2y_0] = z_6z_4z_2z_0$
- $GS[0y_5y_3y_1] = z_7z_5z_3z_1$ and $ILUT[1y_4y_2y_0] = z'_6z'_4z'_2z'_0$
- $GS[1y_5y_3y_1] = z'_7z'_5z'_3z'_1$ and $ILUT[0y_4y_2y_0] = z_6z_4z_2z_0$
- $GS[1y_5y_3y_1] = z'_7z'_5z'_3z'_1$ and $ILUT[1y_4y_2y_0] = z'_6z'_4z'_2z'_0$

As by hypothesis the values of SBoxes GS and $ILUT$ are known to the attacker, he can compute those four different values $GS[0y_5y_3y_1]$, $GS[1y_5y_3y_1]$, $ILUT[0y_4y_2y_0]$ and $ILUT[1y_4y_2y_0]$. The output arrays of output encodings corresponding to those pairs of GS and $ILUT$ outputs will then respectively be noted

- $G_{r,3}((z_7 \oplus c)z_6)G_{r,2}((z_5 \oplus k_0)z_4)G_{r,1}((z_3 \oplus k_1)z_2)G_{r,0}(z_1z_0)$
- $G_{r,3}((z_7 \oplus c)z'_6)G_{r,2}((z_5 \oplus k_0)z'_4)G_{r,1}((z_3 \oplus k_1)z'_2)G_{r,0}(z_1z'_0)$
- $G_{r,3}((z'_7 \oplus c)z_6)G_{r,2}((z'_5 \oplus k_0)z_4)G_{r,1}((z'_3 \oplus k_1)z_2)G_{r,0}(z'_1z_0)$
- $G_{r,3}((z'_7 \oplus c)z'_6)G_{r,2}((z'_5 \oplus k_0)z'_4)G_{r,1}((z'_3 \oplus k_1)z'_2)G_{r,0}(z'_1z'_0)$

The output bits values z_i and z'_i, the round constant bit c value as well as the outputs of the TBox output encodings $T_0[00x_5x_4x_3x_2x_1x_0]$, $T_0[01x_5x_4x_3x_2x_1x_0]$, $T_0[10x_5x_4x_3x_2x_1x_0]$ and $T_0[11x_5x_4x_3x_2x_1x_0]$ are known to the attacker, the key bits k_0 and k_1 are not. Even with this knowledge, the attacker does not have enough information on the output encodings to determine with certainty the key bits.

For example, we consider the second left-most output encoding $G_{r,2}$ and the corresponding key bit k_0. The attacker knows the values $z_4, z'_4, z_5, z'_5 \in \mathbb{F}_2$ and the set of values $\mathcal{G} = \{G_{r,2}((z_5 \oplus k_0)z_4), G_{r,2}((z_5 \oplus k_0)z'_4), G_{r,2}((z'_5 \oplus k_0)z_4), G_{r,2}((z'_5 \oplus k_0)z'_4)\}$. Thus, he can at most recover the right input bits of $G_{r,2}$. More precisely, he knows the right bit of the inputs of $G_{r,2}$ whose corresponding output belong to the set of values $\mathcal{G}$: for $g \in \mathcal{G}$, he knows $y \in \mathbb{F}_2$ such that $G_{r,2}(xy) = g$, with an undetermined $x \in \mathbb{F}_2$. Furthermore,

- If $z'_5 = z_5$ and $z'_4 = z_4$ then $\#\mathcal{G} = 1$. Twelve out of the twenty-four 2-bit encodings g effectively verify that the right bit of the input of g whose corresponding output is $G_{r,2}((z_5 \oplus k_0)z_4)$ equals to z_4: among those twelve encodings, six verify that $g(0z_4) = G_{r,2}((z_5 \oplus k_0)z_4)$ and the other six verify that $g(1z_4) = G_{r,2}((z_5 \oplus k_0)z_4)$.

- If $z'_5 = z_5$ and $z'_4 = \overline{z_4}$ then $\#\mathcal{G} = 2$. Four of the twelve 2-bit encodings g verifying that the right bit of the input $ab \in \mathbb{F}_2$ of g whose corresponding output is $G_{r,2}((z_5 \oplus k_0)z_4)$ equals to z_4 also verify that $g(a\overline{b}) = G_{r,2}((z_5 \oplus k_0)\overline{z_4})$. Among those four encodings, two verify that $g(0z_4) = G_{r,2}((z_5 \oplus k_0)z_4)$ and the other two verify that $g(1z_4) = G_{r,2}((z_5 \oplus k_0)z_4)$.

- If $z'_5 = \overline{z_5}$ and $z'_4 = z_4$ then $\#\mathcal{G} = 2$. Four of the twelve 2-bit encodings g verifying that the right bit of the input $ab \in \mathbb{F}_2$ of g whose corresponding output is $G_{r,2}((z_5 \oplus k_0)z_4)$ equals to z_4 also verify that $g(\overline{a}b) = G_{r,2}((\overline{z_5} \oplus k_0)z_4)$. Among those four encodings, two verify that $g(0z_4) = G_{r,2}((z_5 \oplus k_0)z_4)$ and the other two verify that $g(1z_4) = G_{r,2}((z_5 \oplus k_0)z_4)$.

- If $z'_5 = \overline{z_5}$ and $z'_4 = \overline{z_4}$ then $\#\mathcal{G} = 4$. Two of the twelve 2-bit encodings g verifying that the right bit of the input $ab \in \mathbb{F}_2$ of g whose corresponding output is $G_{r,2}((z_5 \oplus k_0)z_4)$ equals to z_4 also verify that $g(\overline{a}b) = G_{r,2}((\overline{z_5} \oplus k_0)z_4)$, $g(a\overline{b}) = G_{r,2}((z_5 \oplus k_0)\overline{z_4})$ and $g(\overline{ab}) = G_{r,2}((\overline{z_5} \oplus k_0)\overline{z_4})$. Among those two encodings, one verifies that $g(0z_4) = G_{r,2}((z_5 \oplus k_0)z_4)$ and the other one verifies that $g(1z_4) = G_{r,2}((z_5 \oplus k_0)z_4)$.

Therefore, in all cases there exists as many potential 2-bit encodings $G_{r,2}$ compatible with $k_0 = 0$ as potential 2-bit encodings $G_{r,2}$ compatible with $k_0 = 1$. Consequently, it is not possible for an attacker to determine key bits in a differential manner without further information on the output encodings of the TBox.

8.3.4 Conclusion

In this subsection we demonstrate that using a linear intern SBox $ILUT$ in the presented TBox construction leads to the recovery of this SBox, but that does not allow an attacker to determine with certainty the key bits embedded in a TBox T by a differential attack on T itself.

Nevertheless using such a linear $ILUT$ implies a slight improvement in the 2-round differential attack detailed in subsection 8.2. We however do not consider this eventuality to constitute a major additionnal threat to the construction as,

by Property 4, there exists $\prod_{i=0}^{3}(2^4-2^i) = 20160$ linear 4-bit SBoxes. Therefore, the probability to randomly pick an 4-bit SBox that is linear equals to $\frac{20160}{2^4!}$, i.e. around 2^{-30}, thus negligeable.

8.4 Another GIFT SBox

We search for the existence of an SBox complying with the GIFT SBox properties listed in subsubsection 8.4.1 while preventing the differential attack presented in subsection 8.2.

8.4.1 GIFT SBox Properties

The GIFT SBox GS has been chosen according to the following properties ([BPP+17b], §3.3):

- The implementation cost of GS should be of at most 17 units, with the operations NOT, NAND, NOR counting as 1 unit, and XOR and XNOR as 2 units.
- GS should have differential score and linear score of at least 4. (see subsection 4.3)
- There exists a common BOGI permutation for both differential and linear cases. In other words, there exists a permutation of $\{0001, 0010, 0100, 1000\}$ such that the Bad Outputs (BO) are mapped to Good Inputs (GI) in both of DDT and LAT cases.
- $\#\left\{(\Delta x, \Delta y) \in \mathbb{F}_2^4 \times \mathbb{F}_2^4 | \#\left\{x \in \mathbb{F}_2^4 | GS[x] \oplus GS[x \oplus \Delta x] = \Delta y\right\} > 4\right\} \leqslant 2.$
- $\#\left\{x \in \mathbb{F}_2^4 | GS[x] \oplus GS[x \oplus \Delta x] = \Delta y\right\} > 4 \implies HW(\Delta x) + HW(\Delta y) \geqslant 4.$

8.4.2 Attack Resistance Property

The attack presented in subsection 8.2 functions thanks to the differential properties of the GIFT SBox GS. To prevent this attack from happening, an SBox S must ensure that no differential SBox computation leads to exploitable information.

For instance, noting $S[y_7y_5y_30] = z_7z_5z_3z_1$ and $S[y_7y_5y_31] = z_7{}^*z_5{}^*z_3{}^*z_1{}^*$ with $z_i{}^* \in \{z_i, \overline{z_i}, z_i'\}$ and z_i' being either z_i or $\overline{z_i}$ (but not determined), no information about y_7, y_5 or y_3 should leak thanks to the knowledge of the values of S. In other words, we must have

$$\mathbb{P}\left(y_i = 0 \;\middle|\; \begin{array}{l} S[y_7y_5y_30] = z_7z_5z_3z_1 \\ S[y_7y_5y_31] = z_7{}^*z_5{}^*z_3{}^*z_1{}^* \end{array}\right) = \frac{1}{2}$$

for every $i \in \{3, 5, 7\}$.

For example, if there exists a tuple $(y_7, y_5, y_3) \in \mathbb{F}_2^3$ such that $S[y_7y_5y_30] = z_7z_5z_3z_1$ and $S[y_7y_5y_31] = z_7z_5'z_3'\overline{z_1}$, then we must have that for each $(vy_7, vy_5, vy_3) \in \mathbb{F}_2^3$ $S[vy_7vy_5vy_30] = vz_7vz_5vz_3vz_1$ implies that $S[vy_7vy_5vy_31] = vz_7vz_5'$ $vz_3'\overline{vz_1}$. Hence, no information on (y_7, y_5, y_3) would leak.

To avoid any point of attack, the previous property should stand for every pair

- $S[y_7y_5y_30]$ and $S[y_7y_5y_31]$
- $S[y_7y_50y_1]$ and $S[y_7y_51y_1]$
- $S[y_70y_3y_1]$ and $S[y_71y_3y_1]$
- $S[0y_5y_3y_1]$ and $S[1y_5y_3y_1]$

and for every $z_7{}^*z_5{}^*z_3{}^*z_1{}^*$, with $z_i{}^* \in \{z_i, \overline{z_i}, z_i'\}$.

Therefore, this implies that, to thwart this attack on the TBox, the underlying SBox S should obey to the following property :

Property 14. *There exists* $(s_3, s_2, s_1, s_0) \in \mathbb{F}_2^4$ *such that for all* $(y_7, y_5, y_3, y_1) \in \mathbb{F}_2^4, S[y_7y_5y_3y_1] = S[\overline{y_7y_5y_3y_1}] \oplus (s_3, s_2, s_1, s_0)$.

Consequently, there exists $16*15*14*12*10*8*6*4*2 = 30*2^7*8! \simeq 2^{27}$ 4-bit SBoxes satisfying this property. We aim to determine if any of these SBoxes could also satisfy the GIFT SBox properties.

8.4.3 A New SBox

After an heuristic research, we determine the following SBox GS' that complies with most of the GIFT SBox properties.

$$GS' = [5, 0, 3, 4, 15, 6, 1, 10, 2, 9, 14, 7, 12, 11, 8, 13]$$

Indeed, GS' presents the following 1-1 Difference Distribution Table (Table 2) and 1-1 Linear Approximation Table (Table 3) :

δ_i \ δ_o	0001	0010	0100	1000
0001	0	0	0	0
0010	0	0	4	0
0100	0	4	0	0
1000	0	0	0	0

Table 2: 1-1 DDT of GS'

δ_i \ δ_o	0001	0010	0100	1000
0001	0	0	0	0
0010	0	0	0	0
0100	0	0	0	4
1000	0	0	0	4

Table 3: 1-1 LAT of GS'

Thus, firstly, the repartition between Good Inputs and Bad Inputs on the one hand and Good Outputs and Bad Outputs on the other hand that are observed from the 1-1 DDT are :

- $GI_D = \{0001, 1000\}$
- $BI_D = \{0010, 0100\}$
- $GO_D = \{0001, 1000\}$
- $BO_D = \{0010, 0100\}$

Secondly, the repartition between Good Inputs and Bad Inputs on the one hand and Good Outputs and Bad Outputs on the other hand that are observed from the 1-1 LAT are :

- $GI_L = \{0001, 0010\}$
- $BI_L = \{0100, 1000\}$
- $GO_L = \{0001, 0010, 0100\}$
- $BO_L = \{1000\}$

Therefore, we can sum up the compliance of GS' to the GIFT SBox properties as following:

- GS' has for differential score $\#GI_D + \#GO_D = 4$ and linear score $\#GI_L + \#GI_D = 5$.
- There exists two BOGI permutations that are common to differential and linear case, i.e. permutations $\pi_i : \{0001, 0010, 0100, 1000\} \rightarrow \{0001, 0010, 0100, 1000\}$ such that

$$\begin{cases} x \in BO_D \implies \pi_D(x) \in GI_D \\ x \in BO_L \implies \pi_L(x) \in GI_L \end{cases}$$

These permutations verify

$$\begin{array}{lll} \pi_1 : & 0001 \rightarrow 0100 & \pi_2 : \quad 0001 \rightarrow 0100 \\ & 0010 \rightarrow 0001 & \quad\quad\; 0010 \rightarrow 1000 \\ & 0100 \rightarrow 1000 & \quad\quad\; 0100 \rightarrow 0001 \\ & 1000 \rightarrow 0010 & \quad\quad\; 1000 \rightarrow 0010 \end{array}$$

- $\left\{(\Delta x, \Delta y) \in \mathbb{F}_2^4 \times \mathbb{F}_2^4 | \#\left\{x \in \mathbb{F}_2^4 \mid GS'[x] \oplus GS'[x \oplus \Delta x] = \Delta y\right\} > 4\right\} = \{(0000, 0000), (1111, 1000)\}$.
- Consequently,

$$\begin{cases} \#\left\{x \in \mathbb{F}_2^4 | GS[x] \oplus GS[x \oplus \Delta x] = \Delta y\right\} > 4 \\ (\Delta x, \Delta y) \neq (0000, 0000) \end{cases}$$
$$\Leftrightarrow (\Delta x, \Delta y) = (1111, 1000)$$
$$\Rightarrow HW(\Delta x) + HW(\Delta y) \geqslant 4$$

Moreover, GS' complies with the differential attack property (i.e. Property 14) as for all $(y_7, y_5, y_3, y_1) \in \mathbb{F}_2^4$,

$$GS'[y_7 y_5 y_3 y_1] = GS'[\overline{y_7 y_5 y_3 y_1}] \oplus (1, 0, 0, 0).$$

Algorithm 3: GS' Bitsliced Implementation

Input : (MSB) $x[3], x[2], x[1], x[0]$ (LSB)
Output: (MSB) $y[3], y[2], y[1], y[0]$ (LSB)

1 $y[0] = x[0]$ $NXOR$ $x[3]$
2 $t_0 = x[0]$ XOR $x[1]$
3 $t_1 = x[3]$ $NAND$ t_0
4 $t_2 =$ $NOT(t_0)$
5 $y[3] = t_1$ XOR t_3
6 $t_4 = x[2]$ XOR $x[3]$
7 $t_5 =$ $NOT(y[0])$
8 $t_6 = t_4$ $NAND$ t_5
9 $y[1] = t_0$ XOR t_6
10 $t_7 = t_1$ XOR t_4
11 $t_8 =$ $NOT(x[1])$
12 $t_9 = x[0]$ NOR t_8
13 $y[1] = t_7$ XOR t_9
14 **return** *(y[3],y[2],y[1],y[0])*

Nevertheless, this SBox has an implementation cost of 20 units. Indeed, its bitsliced implementation (detailed in algorithm 3) consists of 6 XOR, 3 NOT, 2 NAND, 1 NXOR and 1 NOR, with respective costs of two, one, one, two and one unit.

We note nevertheless that the instructions of this bitsliced implementation do not comply with the types recommended in [BPP+17b], designed for the ease of the implementation of the SBox inverse.

Part II
A New Fault Resistant Masking Scheme

9 Masking Scheme Design Rationale

The second goal of this thesis is to design a Boolean masking scheme composed of Boolean operations that, complementary to its side-channel countermeasure nature by design, can correct at most two one-bit flipping faults committed during (or just before) its execution.

This section details the main design choices made during the development of this new masking scheme. First of all, it will be described how the scheme verifies the properties listed in subsection 5.4 that are sufficient conditions for first-order probing security. Subsequently, it will be explained how the correction of input shares is performed at the beginning of the scheme execution.

9.1 Masking Scheme Design Constraints

We aim to design our AND masking scheme so that it verifies the three implementation properties listed in Lemma 1 (correctness, uniformity and non-completeness) to ensure its first order probing security. The purpose of the scheme is to compute $a \wedge b$ with $a, b \in \mathbb{F}_2$, while supposing that all input shares A_i from input A and B_i from input B are words of an error-correcting code in order to introduce code correction at the beginning of the scheme. In other words, $HW(\bigoplus_i A_i) \bmod 2 = a$ and $HW(\bigoplus_i B_i) \bmod 2 = b$. To ensure compatibility between consecutive instances of the scheme, this implies that all the output shares will be codewords as well. Since the goal is to design a Boolean masking scheme, we aim to use an error-correcting code whose corresponding decoding process could be written with only Boolean operations, which is the case for BCH error-correcting codes, via the Peterson-Gorenstein-Zierler algorithm ([Pet60]). Furthermore, we also want to introduce randomness, with random polynomials R_i of respective parities r_i.

To achieve this, we design in the first place variables $s_i \in \mathbb{F}_2$ intented to be the parities of the codewords S_i, output shares of our scheme. These output shares parities depend on products between parities a_i and b_i of shares of different inputs, products $a_i r_j$ or $r_i b_j$ between parities of input shares and parities of random polynomials, and products $r_i r_j$ between parities of random polynomials. We aim to design those variables s_i so that they comply with both uniformity and non-completeness properties regarding the input shares and random polynomials parities a_i, b_i and r_i. Finally, by the correctness property, the sum of these output shares parities s_i will be equal to $a \wedge b$.

In the remainder of this subsection, we describe how these aimed design and masking scheme properties influence the construction of output shares parities s_i and the choice of the used BCH code.

9.1.1 Non-Completeness Implies a Condition on the Number of Shares

First of all, to be able to perform one instance of the scheme after another, the number of output shares needs to be equal to the number of shares of each input.

Furthermore, to comply with non-completeness the number of input shares of both a and b cannot be equal to 2. Indeed, in this case, we would suppose that

- a is represented by two codewords shares A_0 and A_1 of respective parities a_0 and a_1 such that $a_0 \oplus a_1 = a$.
- b is represented by two codewords shares B_0 and B_1 of respective parities b_0 and b_1 such that $b_0 \oplus b_1 = b$.
- There are two output codewords shares S_0 and S_1 of respective parities s_0 and s_1 depending on parities a_0, a_1, b_0 and b_1, such that their sum $s_0 \oplus s_1$ verifies $s_0 \oplus s_1 = a \wedge b$.

Therefore, the input shares parities a_i and b_i and output shares parities s_0 and s_1 would verify that

$$s_0 \oplus s_1 = a \wedge b = a_0 b_0 \oplus a_0 b_1 \oplus a_1 b_0 \oplus a_1 b_1$$

As a result, there would be four different products $a_i b_j$ of input shares parities to be distributed among the two output shares parities s_i, namely $a_0 b_0$, $a_0 b_1$, $a_1 b_0$ and $a_1 b_1$. Consequently, at least one variable s_i would be computed from two (or more) of those products. But, since there are only two shares a_i and two shares b_i, the sum of any two out of the four products $a_i b_j$ can not verify non-completeness.

In conclusion, the number n_{in} of input and output shares of the scheme needs to be at least 3.

9.1.2 Parity Requirement on the BCH Code Generator Polynomial

As we consider a_i and b_j the parities of codewords input shares and s_i the parities of codewords output shares, the generator polynomial of the BCH correcting code they belong to needs to be of odd parity so that variables a_i, b_j and s_i could take either values in $\mathbb{F}_2$. To that end, we suppose the generator polynomial of this BCH correcting code of length n to be $g(X) = lcm(M_{\alpha^b}(X), M_{\alpha^{b+1}}(X), \ldots, M_{\alpha^{b+\delta-2}}(X))$, such that m is the multiplicative order of 2 modulo n and α is an element of $\mathbb{F}_{q^m}$ of multiplicative order n. Based on Property 10, $g(X)$ can be written as a product of $M_{\alpha^i}(X)$ polynomials.

Furthermore, according to its definition, the degree of a polynomial $M_{\alpha^i}(X)$ is determined by the cardinality of $C(i)$, the corresponding 2-cyclotomic class of i. Therefore, on the one hand, Property 7 states that for any possible code length n the only 2-cyclotomic class modulo n of cardinality one is $C(0)$, the 2-cyclotomic class of 0. Consequently, the corresponding polynomial $M_{\alpha^0}(X)$ has even parity, since $M_{\alpha^0}(X) = X - \alpha^0 = X + 1$. On the other hand, this property also implies that all other 2-cyclotomic classes modulo n have a cardinality of at least two, so the polynomials $M_{\alpha^i}(X)$ with $0 < i < n$ have a degree greater than or equal to two.

In addition to this, all irreducible polynomials of $\mathbb{F}_2[X]$ with degree greater than or equal to two have odd parity, as otherwise they would admit 1 as a root,

and therefore could be factorized by $X+1$. Consequently, as they are irreducible polynomials of $\mathbb{F}_2[X]$ according to Property 9, all polynomials $M_{\alpha^i}(X)$ with $0 < i < n$ have odd parity.

As a conclusion, to ensure its odd parity, the generator polynomial $g(X)$ of the BCH code used in the scheme must not be a multiple of $M_{\alpha^0}(X) = X + 1$. In other words, this generator polynomial $g(X) = lcm(M_{\alpha^b}(X), M_{\alpha^{b+1}}(X), \ldots, M_{\alpha^{b+\delta-2}}(X))$ must verify that $0 \notin \{b \bmod n, b+1 \bmod n, \ldots, b+\delta-2 \bmod n\}$, i.e.

$$0 < b < n - (\delta - 2).$$

9.1.3 Maximizing the BCH Code Dimension

In subsubsection 9.2.5, it will be detailed why a BCH code of correction capacity at least two is needed to ensure the correctness of the scheme. Since the correction capacity t of a BCH code of minimum Hamming distance at least δ verifies $t \geqslant \lfloor \frac{\delta-1}{2} \rfloor$, imposing $t \geqslant 2$ necessitates that $\delta \geqslant 5$. Therefore, the generator polynomial $g(X)$ of the BCH code we aim to use can be rewritten in the form of

$$g(X) = lcm(M_{\alpha^b}(X), M_{\alpha^{b+1}}(X), M_{\alpha^{b+2}}(X), M_{\alpha^{b+3}}(X)),$$

with $0 < b < n - 3$.

Furthermore, we know that subsequently we will need codewords to act as masks to be applied to an array of cross-products of parities among input shares parities a_i and b_i and random polynomials parities r_i, and this in order to compute the parities s_i of the output shares. To ensure correctness, all cross-products involved of form $a_i r_j$, $r_i b_j$ or $r_i r_j$ must appear in an even number of variables s_i and all cross-products involved of form $a_i b_j$ must feature in an odd number of variables s_i, therefore those codewords need to have a certain number of exponents in common. To that end, we aim to maximize the search space for these codewords, and thus the dimension of the code they will belong to.

9.1.4 Randomness Requirement and its Impact on the BCH Code Choice

We note n_r the total number of random polynomials involved in the scheme. We suppose the number of random polynomials associated to each input to be identical, that is to say that there exists as many random polynomials parities r_j that are to be multiplied to input shares parities a_i than random polynomials parities r_i that are to be multiplied to input shares parities b_j. Consequently, n_r is even and the number of random polynomials associated to each input is equal to $\frac{n_r}{2}$.

To keep a reasonable randomness requirement of our scheme, we impose on ourselves that the number $\frac{n_r}{2}$ of random variables associated to each input is strictly less than the number of shares of each input. For instance, the number of

random polynomials parities r_j that are to be multiplied to the input parities a_i is defined to be strictly less than the number n_{in} of shares A_i, and reciprocally.

As the number n_{in} of input shares A_i is equal to the number of input shares B_i, it implies that

$$\frac{n_r}{2} < n_{in}.$$

Therefore, by definition of n_{in} and $\frac{n_r}{2}$, there exists at most $(n_{in} + \frac{n_r}{2})^2$ products of parities involved in the computation of output shares parities s_i, divided as follows:

- n_{in}^2 products $a_i b_j$ of input shares parities
- $\frac{n_{in} * n_r}{2}$ products $a_i r_j$
- $\frac{n_{in} * n_r}{2}$ products $r_i b_j$
- $\left(\frac{n_r}{2}\right)^2$ products $r_i r_j$

The total number of these products is considered to be an upper bound of the length of the BCH code used in the scheme, as codewords serving as masks will be applied to the array gathering them (see section 10). Therefore, applying a mask to an array that would be longer than this array would provoke a loss of efficiency. In the same manner, n_{in}^2 is a lower bound of the code length n since it corresponds to the number of products $a_i b_j$, that are imperatively involved in the computations of s_i variables to ensure correctness. To sum up, the code length n must verify that

$$n_{in}^2 \leqslant n \leqslant (n_{in} + \frac{n_r}{2})^2$$

First and foremost, supposing that the number of shares n_{in} is 3 and the number of random polynomials associated to each input $\frac{n_r}{2}$ is 2, these bounds imply that the code length n would verify that $9 \leqslant n \leqslant 25$. For every potential code length n complying with those bounds, the minimal degree of generator polynomials verifying the conditions detailed in subsubsection 9.1.3 is listed in Table 4 hereunder, together with the corresponding maximum BCH code dimension.

Code Length n	9	11	13	15	17	19	21	23	25
Minimal Generator Polynomial Degree	8	10	12	8	16	18	9	11	20
Corresponding Maximum Code Dimension	1	1	1	7	1	1	12	12	5

Table 4: Maximum code dimension depending on code length, with generator polynomial $g(X)$ of adequate form, for a number of shares $n_{in} = 3$

Therefore, for a code length n within the bounds $9 \leqslant n \leqslant 25$, the maximum potential dimension of a BCH code of correction capacity at least 2 is $k = 12$, reached for either $n = 21$ or $n = 23$. These cases correspond to BCH codes of cardinality 2^{12}.

Secondly, if the number of shares n_{in} is 4 and the number of random polynomials associated to each input $\frac{n_r}{2}$ is 3, then the code length n verifies $16 \leqslant n \leqslant 49$. Retaining the same BCH code properties, the maximum potential dimension value reaches $k = 29$ when $n = 45$, as listed in Table 5. In this case, the cardinality of the code equals to 2^{29}.

Code Length n	17	19	21	23	25	27	29	31	33
Minimal Generator Polynomial Degree	16	18	9	11	20	20	28	10	20
Corresponding Maximum Code Dimension	1	1	12	12	5	7	1	21	13

Code Length n	35	37	39	41	43	45	47	49
Minimal Generator Polynomial Degree	15	36	24	20	28	16	23	42
Corresponding Maximum Code Dimension	20	1	15	21	15	29	24	7

Table 5: Maximum code dimension depending on code length, with generator polynomial $g(X)$ of adequate form, for a number of shares $n_{in} = 4$

Since the code dimension $k = 29$ implies a rather large search space for potential codewords serving as masks, we select the corresponding number of shares $n_{in} = 4$. Subsequently, if the number of random polynomials associated to each input $\frac{n_r}{2}$ verifies $\frac{n_r}{2} \leqslant 2$, then it implies that $(n_{in} + \frac{n_r}{2})^2 < n = 45$. This brings a contradiction to the condition that $(n_{in} + \frac{n_r}{2})^2$ is an upper bound of the value of code length n, therefore we define $\frac{n_r}{2} = 3$.

Therefore, the BCH code used in the scheme has length $n = 45$, cardinality 2^{29} and generator polynomial

$$\begin{aligned} g(X) &= lcm(M_\alpha(X), M_{\alpha^2}(X), M_{\alpha^3}(X), M_{\alpha^4}(X)) \\ &= (X^{12} + X^3 + 1) * (X^4 + X + 1) \\ &= X^{16} + X^{13} + X^{12} + X^7 + X^3 + X + 1, \end{aligned}$$

with $m = 12$ is the multiplicative order of 2 modulo $n = 45$ and α is a 45^{th} primitive root of unity in $\mathbb{F}_{q^m} = \mathbb{F}_{2^{12}}$.

9.1.5 Ensuring Uniformity and Correctness

Since we choose the number of shares to be $n_{in} = 4$ and the number of random polynomials associated to each input to be $\frac{n_r}{2} = 3$, there exists at most 49 different products of parities involved in the computation of the output shares parities s_i. Those parities can be divided in the following way :

- 16 products of form $a_i b_j$ that must all be present in an odd number of s_i computations.
- 12 products of form $a_i r_j$, 12 products of form $r_i b_j$ and 9 products of form $r_i r_j$. Each of these products does not necessarily need to be present in the computation of any parity s_i, but if it does, it needs to be involved in an even number of s_i computations to guarantee correctness (so that the occurrences could cancel themselves).

It can be noticed that, to ensure uniformity, it is compulsory that each s_i includes at least one single parity, i.e. one variable of the form a_i, b_i or r_i. Indeed, as variables a_i, b_i and r_i are independent from one another and verify equiprobability in $\mathbb{F}_2$, all products of two of those variables admit $(\frac{3}{4}, \frac{1}{4})$ as probability vector in $\mathbb{F}_2$. Therefore, sums of those products cannot be equiprobable according to Property 15 hereinbelow.

Property 15. *Let $\mathcal{P}$ the set of products of form $a_i b_j$, $a_i r_j$, $r_i b_j$ and $r_i r_j$. Any sum s of elements in $\mathcal{P}$ complying with non-completeness cannot be equiprobable in $\mathbb{F}_2$.*

Proof. We will suppose for this proof that the sum s comprises addends of the four types. If otherwise, this does not change the core of the demonstration. Rearranging the addends according to their type, such sum s can be written in the form of $a_{j_0} b_{k_0} \oplus \ldots \oplus a_{j_{\Theta-1}} b_{k_{\Theta-1}} \oplus a_{l_0} r_{m_0} \oplus \ldots \oplus a_{l_{\Lambda-1}} r_{m_{\Lambda-1}} \oplus r_{n_0} b_{o_0} \oplus \ldots \oplus r_{n_{\chi-1}} b_{o_{\chi-1}} \oplus r_{p_0} r_{q_0} \oplus \ldots \oplus r_{p_{\psi-1}} r_{q_{\psi-1}}$, with $0 \leqslant \Theta \leqslant 16$, $0 \leqslant \Lambda \leqslant 12$, $0 \leqslant \chi \leqslant 12$ and $0 \leqslant \psi \leqslant 9$.

As s complies with non-completeness, it involves at most $n_{in} - 1 = 3$ different shares a_i. We will note them a_υ, a_δ and a_ξ. In potential cases where strictly less than three shares a_i would be involved in the computation of s, we can consider a_υ, a_δ or a_ξ to be null. Subsequently, s can be also be rewritten such that

$$s = a_\upsilon VB_\upsilon \oplus a_\delta VB_\delta \oplus a_\xi VB_\xi \oplus r_0 VB_0 \oplus r_1 VB_1 \oplus r_2 VB_2,$$

with each VB_i being a sum of at most $n_{in} - 1 = 3$ variables b_i and some variables r_i among r_3, r_4 and r_5. In the same way as for variables a_υ, a_δ and a_ξ, if r_0, r_1 or r_2 are not involved in s, we respectively consider VB_0, VB_1 or VB_2 to be null.

By definition of sharings (subsection 3.2), any tuple of at most $n_{in} - 1 = 3$ parities b_i of shares B_i constitute a set of variables independent from one another. Furthermore, random polynomials R_i are independent one from another and independent from shares B_i by design, and so are their respective parities b_i and r_i. Therefore, each sum VB_i verifies

$$\mathcal{P}(VB_i = 0) = \mathcal{P}(VB_i = 1) = \frac{1}{2}.$$

As mentioned above, all random polynomials R_i are independent from each other by design. Likewise, they are independent from shares A_i, and shares A_i are also independent from shares B_i, therefore so are their respective parities a_i, b_i and r_i. Hence all variables a_i, r_0, r_1 and r_2 are independent from sub-sums VB_j, and any sub-sum S of variables a_i, r_0, r_1 and r_2 verifies $\mathbb{P}(S = 0) = \mathbb{P}(S = 1) = \frac{1}{2}$. Furthermore, for $x = 0$, 1 or 2, we note $v_x = r_x$ and for $x = \upsilon$, δ or ξ we note $v_x = a_x$. Then, in the same manner that $\mathbb{P}(S = 0) = \mathbb{P}(S = 1) = \frac{1}{2}$, each product $v_i VB_i$ verifies $\mathbb{P}(v_i VB_i = 1) = \frac{1}{4}$.

Moreover, we suppose that there exists $\{x_0, \ldots, x_{p'-1}\} \subset \{0, 1, 2, \upsilon, \delta, \xi\}$ such that $VB_{x_0} = \ldots = VB_{x_{p'-1}}$. As demonstrated above, $\mathbb{P}(v_{x_0} \oplus \ldots \oplus v_{x_{p'-1}} = 1) = \frac{1}{2}$. Therefore,

$$\begin{aligned}
\mathbb{P}(v_{x_0} VB_{x_0} \oplus \ldots \oplus v_{x_{p'-1}} VB_{x_{p'-1}} = 1) &= \mathbb{P}((v_{x_0} \oplus \ldots \oplus v_{x_{p'-1}}) VB_{x_0} = 1) \\
&= \mathbb{P}(v_{x_0} \oplus \ldots \oplus v_{x_{p'-1}} = 1)\mathbb{P}(VB_{x_0} = 1) \\
&= \frac{1}{2} * \frac{1}{2} \\
&= \frac{1}{4}
\end{aligned}$$

Consequently, the sum s can be rewritten in the form of $VA_0VB_0 \oplus \ldots \oplus VA_{\Omega-1}VB_{\Omega-1}$, with

- VB_i distincts
- VA_i being either $v_x \in \{a_\upsilon, a_\delta, a_\xi, r_0, r_1, r_2\}$ or a sub-sum of variables in $\{a_\upsilon, a_\delta, a_\xi, r_0, r_1, r_2\}$.
- each v_x appearing only once among all the VA_i

Therefore, all the sums VA_i and VB_i are independent from one another and equiprobable in $\mathbb{F}_2$. Based on Property 1, this implies that

$$\mathbb{P}(s = 0) = \frac{10}{2^{\Omega+2}} - \frac{1}{2^{\Omega-1}} + \frac{1}{2}.$$

Furthermore, if we suppose s being equiprobable, then $\mathbb{P}(s = 0) = \frac{1}{2}$. It thus implies that $\frac{10}{2^{\Omega+2}} - \frac{1}{2^{\Omega-1}} + \frac{1}{2} = \frac{1}{2}$, then

$$\begin{aligned}
\frac{10}{2^{\Omega+2}} - \frac{1}{2^{\Omega-1}} = 0 &\iff \frac{10}{2^{\Omega+2}} = \frac{1}{2^{\Omega-1}} \\
&\iff 10 = 8,
\end{aligned}$$

hence the contradiction. Consequently, a sum $s = a_{j_0}b_{k_0} \oplus \ldots \oplus a_{j_{\Theta-1}}b_{k_{\Theta-1}} \oplus a_{l_0}r_{m_0} \oplus \ldots \oplus a_{l_{\Lambda-1}}r_{m_{\Lambda-1}} \oplus r_{n_0}b_{o_0} \oplus \ldots \oplus r_{n_{\chi-1}}b_{o_{\chi-1}} \oplus r_{p_0}r_{q_0} \oplus \ldots \oplus r_{p_{\psi-1}}r_{q_{\psi-1}} = VA_0VB_0 \oplus \ldots \oplus VA_{\Omega-1}VB_{\Omega-1}$ of elements of set $\mathcal{P}$ cannot be equiprobable. □

Therefore, to guarantee the uniformity of each s_i, at least one single parity among the variables a_i, b_i and r_i needs to figure in the computation of each s_i. Moreover, similarly as for products $a_i r_j$, $r_i b_j$ and $r_i r_j$, these single parities must be added in even numbers of s_i computations to ensure the correctness of the overall scheme.

Therefore, to build the output shares parities s_i, we first randomly split the 16 products of parities of form $a_i b_j$ among s_0, s_1, s_2 and s_3 so that each of them complies with non-completeness. At this point, each s_i variable cannot verify equiprobability (according to Property 15). Consequently, we add three single variables to pairs of s_i to ensure global equiprobability of each s_i among the values observed for all possible inputs, and then equiprobability conditioned by the value of $a \wedge b$ necessary for the uniformity property. Among all single variables, using r_0, a_2 and r_5 allows us to do so while verifying non-completeness.

Subsequently, we add products of form $a_i r_j$,$r_i b_j$ or $r_i r_j$ one after another to pairs of s_i variables, so that, when conditioned by either value of $a \wedge b$, the probability of each value of (s_0, s_1, s_2, s_3) verifying $s_0 \oplus s_1 \oplus s_2 \oplus s_3 = a \wedge b$ equals to $\left(\frac{1}{2}\right)^3$, while retaining the non-completeness property. We obtain the following formulas for the respective parities of output shares S_0, S_1, S_2 and S_3:

- $s_0 = r_1 b_1 \oplus r_2 b_1 \oplus r_0 \oplus a_1 r_3 \oplus a_2 r_5 \oplus a_2 b_2 \oplus a_3 r_3 \oplus a_3 b_2 \oplus r_1 b_0 \oplus a_3 b_1 \oplus r_1 r_5 \oplus r_2 r_5$
- $s_1 = r_1 b_1 \oplus r_0 \oplus a_2 b_1 \oplus a_0 b_2 \oplus a_2 r_5 \oplus a_0 b_1 \oplus a_0 r_3 \oplus a_3 b_0 \oplus r_2 b_2 \oplus a_0 b_0 \oplus r_2 r_5 \oplus a_2 \oplus r_1 r_5 \oplus r_0 r_3$
- $s_2 = a_3 b_3 \oplus a_1 r_5 \oplus r_5 \oplus a_3 r_3 \oplus r_2 b_2 \oplus a_1 b_2 \oplus r_1 b_0 \oplus r_2 r_5 \oplus r_2 b_0 \oplus a_2 \oplus r_0 r_3 \oplus a_2 r_4$
- $s_3 = a_2 b_3 \oplus r_2 b_1 \oplus a_1 r_5 \oplus a_1 b_1 \oplus r_5 \oplus a_1 r_3 \oplus a_0 r_3 \oplus a_1 b_0 \oplus a_0 b_3 \oplus r_2 r_5 \oplus r_2 b_0 \oplus a_1 b_3 \oplus a_2 b_0 \oplus a_2 r_4$

9.2 Input Shares Correction

To fulfill our purpose of developing the new masking scheme to be fault-resistant, we design the input shares correction part as described in subsubsection 9.2.2. Additionally, we specify in subsubsection 9.2.1 the fault attack model we place ourselves into.

Furthermore, for the purpose of detail, the effective functioning of the input shares correction part will be described for the different fault cases considered by our scheme in subsubsection 9.2.4 and subsubsection 9.2.5.

9.2.1 Fault Attack Model

For the design of this new masking scheme, we place ourselves in the one-bit flipping fault attack model, that is to say we consider an attacker to be able to randomly flip a bit in the implementation. Indeed, this is a fault attack model likely to be used by an attacker targeting a bitsliced implementation.

We demonstrate below the resistance of the masking scheme to the introduction of at most two one-bit faults in its input shares, as it is a reasonable fault resistance goal for each AND operation among all AND instances of the bitsliced implementation of a cryptographic primitive.

9.2.2 Input Shares Correction Design

The objective of the correcting design is to be able to correct potential faults in the masking scheme input shares, that is to say to correct faults committed at the end of the preceding instances or just before the beginning of the current instance of the scheme. That is why we impose that all input and output shares of the masking scheme will be codewords, and why the code correction is placed at the beginning of the scheme rather than at the end : as instances of the scheme are supposed to be used successively in a wider bitsliced implementation to mask different ANDs, this implies that faults committed in the last stages of one instance of the scheme or just after the end will be corrected at the beginning of the following instances.

To that end, to minimize the number of corrections needed we compute and correct sub-sums of input shares instead of correcting each input share separately. For non-completeness compliance, it is not possible to add all polynomial shares A_i of input A (or all polynomial shares B_i of input B) in a same sub-sum. Furthermore, we aim to design those sub-sums in such a way that, while using the smallest number of them, it is feasible in case of a fault to determine whether it was committed on A or B although they are sub-sums of shares coming from both inputs A and B. To do so, we determine the following repartition of input shares between sub-sums V_0, V_1 and V_2 and correct those sub-sums into respective codewords noted V_0', V_1' and V_2':

- $V_0 = A_0 \oplus A_1 \oplus B_0 \oplus B_1$
- $V_1 = A_2 \oplus A_3 \oplus B_2 \oplus B_3$
- $V_2 = A_2 \oplus A_3 \oplus B_0 \oplus B_1$

We can outline this repartition of polynomial shares A_i and B_i among sub-sums V_0, V_1 and V_2 as follows in Figure 5.

Subsequently, we add the sub-sums V_i and their corrected values V_i' to the polynomial input shares A_i and B_i to obtain the modified input shares A_i' and B_i'. These modified input shares will be used instead of the initial ones for the remainder of the masking scheme.

- $A_0' = A_0 \oplus V_0$
- $A_1' = A_1 \oplus V_1'$
- $A_2' = A_2 \oplus V_2$
- $A_3' = A_3 \oplus V_2'$
- $B_0' = B_0 \oplus V_1$
- $B_1' = B_1 \oplus V_0'$
- $B_2' = B_2 \oplus V_2$
- $B_3' = B_3 \oplus V_2'$

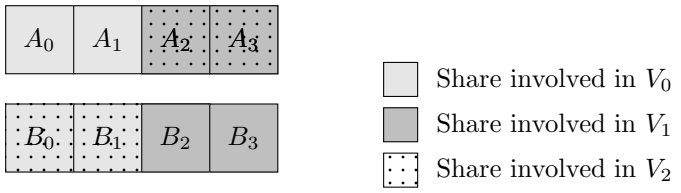


Figure 5: Diagram of the repartition of input shares A_i and B_i among sub-sums V_0, V_1 and V_2

In the case when no fault has been committed, all sub-sums V_i are then naturally not faulted. Therefore, by definition of sub-sums V_i', it implies that $V_i' = V_i$ for $i = 0, 1, 2$ and 3. In this manner, the modified input shares A_i' and B_i' stay coherent with the original input shares of the scheme, given that

- $A_0' \oplus A_1' \oplus A_2' \oplus A_3' = (A_0 \oplus V_0) \oplus (A_1 \oplus V_1') \oplus (A_2 \oplus V_2) \oplus (A_3 \oplus V_2') = A_0 \oplus (A_0 \oplus A_1 \oplus B_0 \oplus B_1) \oplus A_1 \oplus (A_2 \oplus A_3 \oplus B_2 \oplus B_3) \oplus A_2 \oplus A_3 \oplus (V_2 \oplus V_2') = B_0 \oplus B_1 \oplus B_2 \oplus B_3$
- $B_0' \oplus B_1' \oplus B_2' \oplus B_3' = (B_0 \oplus V_1) \oplus (B_1 \oplus V_0') \oplus (B_2 \oplus V_2) \oplus (B_3 \oplus V_2') = B_0 \oplus (A_2 \oplus A_3 \oplus B_2 \oplus B_3) \oplus B_1 \oplus (A_0 \oplus A_1 \oplus B_0 \oplus B_1) \oplus B_2 \oplus B_3 \oplus (V_2 \oplus V_2') = A_0 \oplus A_1 \oplus A_2 \oplus A_3$

9.2.3 The Correction Design Preserves Non-Completeness

In this subsection we verify that the correction design detailed above comply with non-completeness. First of all, we assume that no fault is committed. In this case, by definition, all sub-sums V_i and corresponding corrected values V_i' verify that $V_i' = V_i$.

Consequently,

- $A_0' = A_0 \oplus V_0 = A_0 \oplus (A_0 \oplus A_1 \oplus B_0 \oplus B_1) = A_1 \oplus B_0 \oplus B_1$
- $A_1' = A_1 \oplus V_1' = A_1 \oplus (A_2 \oplus A_3 \oplus B_2 \oplus B_3) = A_1 \oplus A_2 \oplus A_3 \oplus B_2 \oplus B_3$
- $A_2' = A_2 \oplus V_2 = A_2 \oplus (A_2 \oplus A_3 \oplus B_0 \oplus B_1) = A_3 \oplus B_0 \oplus B_1$
- $A_3' = A_3 \oplus V_2' = A_3 \oplus (A_2 \oplus A_3 \oplus B_0 \oplus B_1) = A_2 \oplus B_0 \oplus B_1$
- $B_0' = B_0 \oplus V_1 = B_0 \oplus (A_2 \oplus A_3 \oplus B_2 \oplus B_3) = A_2 \oplus A_3 \oplus B_0 \oplus B_2 \oplus B_3$
- $B_1' = B_1 \oplus V_0' = B_1 \oplus (A_0 \oplus A_1 \oplus B_0 \oplus B_1) = A_0 \oplus A_1 \oplus B_0$
- $B_2' = B_2 \oplus V_2 = B_2 \oplus (A_2 \oplus A_3 \oplus B_0 \oplus B_1) = A_2 \oplus A_3 \oplus B_0 \oplus B_1 \oplus B_2$
- $B_3' = B_3 \oplus V_2' = B_3 \oplus (A_2 \oplus A_3 \oplus B_0 \oplus B_1) = A_2 \oplus A_3 \oplus B_0 \oplus B_1 \oplus B_3$

Secondly, we notice that, in the case where a fault is committed, it does not change the involvment (or not) of shares A_i and B_i in the computations of modified input shares A'_i and B'_i.

As a result, those modified input shares A'_i and B'_i verify non-completeness with regards to the original input shares A_i and B_i.

9.2.4 One-bit Fault on One Input Share

Sub-sums V_0 and V_1 are designed to cover all possible locations for input shares faults, whereas, in case of a fault detected by one of those two sub-sums, V_2 is designed to determine whether this fault has initially been committed on a share A_i of A or on a share B_i of B.

In this section, we detail the impact on the parities of all modified input shares A'_i and B'_i of a one-bit fault on one input share A_i or B_i. With this aim in mind, from now onwards, we note a_i and b_i the parities of respective non-faulted input shares A_i and B_i. For example, if a fault is committed on A_2, it implies that

- $HW(A_2) \bmod 2 \ = a_2 \oplus 1$
- $HW(A_i) \bmod 2 \ = a_i$ for $i \in \{0, 1, 3\}$
- $HW(B_i) \bmod 2 \ = b_i$ for $i \in \{0, 1, 2, 3\}$

A_2 is involved in the computations of sub-sums V_1 and V_2, but not in the computation of V_0. Therefore, commit one fault on A_2 impacts the sub-sums V_1 and V_2, but not V_0. Consequently, the parities of the three sub-sums V_0, V_1 and V_2 in this case are :

- $HW(V_0) \bmod 2 = HW(V'_0) \bmod 2 = a_0 \oplus a_1 \oplus b_0 \oplus b_1$
- $HW(V_1) \bmod 2 = HW(V'_1) \bmod 2 \oplus 1 = a_2 \oplus a_3 \oplus b_2 \oplus b_3 \oplus 1$
- $HW(V_2) \bmod 2 = HW(V'_2) \bmod 2 \oplus 1 = a_2 \oplus a_3 \oplus b_0 \oplus b_1 \oplus 1$

Consequently, it implies that the sums of the parities of modified shares A'_i and B'_i are respectively equal to the sums of the parities of non-faulted shares B_i and A_i. As a matter of fact,

- $HW(A'_0 \oplus A'_1 \oplus A'_2 \oplus A'_3) \bmod 2 = HW(A_0 \oplus A_1 \oplus A_2 \oplus A_3 \oplus V_0 \oplus V'_1 \oplus V_2 \oplus V'_2) \bmod 2 \ = a_0 \oplus a_1 \oplus (a_2 \oplus 1) \oplus a_3 \oplus (a_0 \oplus a_1 \oplus b_0 \oplus b_1) \oplus (a_2 \oplus a_3 \oplus b_2 \oplus b_3) \oplus (a_2 \oplus a_3 \oplus b_0 \oplus b_1) \oplus (a_2 \oplus a_3 \oplus b_0 \oplus b_1 \oplus 1) = b_0 \oplus b_1 \oplus b_2 \oplus b_3$
- $HW(B'_0 \oplus B'_1 \oplus B'_2 \oplus B'_3) \bmod 2 = HW(B_0 \oplus B_1 \oplus B_2 \oplus B_3 \oplus V'_0 \oplus V_1 \oplus V_2 \oplus V'_2) \bmod 2 \ = b_0 \oplus b_1 \oplus b_2 \oplus b_3 \oplus (a_0 \oplus a_1 \oplus b_0 \oplus b_1) \oplus (a_2 \oplus a_3 \oplus b_2 \oplus b_3 \oplus 1) \oplus (a_2 \oplus a_3 \oplus b_0 \oplus b_1 \oplus 1) \oplus (a_2 \oplus a_3 \oplus b_0 \oplus b_1) = a_0 \oplus a_1 \oplus a_2 \oplus a_3$

We can notice that the formulas of sub-sums V_i imply that introducing a one-bit fault on share A_0 brings the same impact on the scheme as introducing a fault on A_1. In the same way, introducing a one-bit fault on A_2 brings the same impact as introducing a fault on A_3, as well as introducing a fault on B_0 or B_1, or on B_2 or B_3. Therefore, for the remainder of this dissertation, we will exclusively consider faults in A_0, A_2, B_0 or B_2 to simplify the notations.

Regarding the correctness of parities of sub-sums V_i depending on the location of the fault, Table 6 indicates if the parities of the sub-sums V_0, V_1 or V_2 are correct after the introduction of a fault on one input share (A_0, A_2, B_0 or B_2).

Fault Location \ V_i	V_0	V_1	V_2
A_0	✗	✓	✓
A_2	✓	✗	✗
B_0	✗	✓	✗
B_2	✓	✗	✓

Table 6: Correctness of the parity of sub-sums V_i depending on the location of the single one-bit fault

According to the definitions of the sub-sums V_i and regardless of the one-bit fault location, the parities of corresponding corrected sub-sums V_i' verify

- $HW(V_0') \bmod 2 = a_0 \oplus a_1 \oplus b_0 \oplus b_1$
- $HW(V_1') \bmod 2 = a_2 \oplus a_3 \oplus b_2 \oplus b_3$
- $HW(V_2') \bmod 2 = a_2 \oplus a_3 \oplus b_0 \oplus b_1$

Moreover, if the one-bit fault impacts a sub-sum V_i (see Table 6), then $HW(V_i) \bmod 2 = HW(V_i') \bmod 2 \oplus 1$.

Fault Location \ Parities of	$V_0 \oplus V_1' \oplus V_2 \oplus V_2'$	$V_0' \oplus V_1 \oplus V_2 \oplus V_2'$
A_0	$b_0 \oplus b_1 \oplus b_2 \oplus b_3 \oplus 1$	$a_0 \oplus a_1 \oplus a_2 \oplus a_3$
A_2	$b_0 \oplus b_1 \oplus b_2 \oplus b_3 \oplus 1$	$a_0 \oplus a_1 \oplus a_2 \oplus a_3$
B_0	$b_0 \oplus b_1 \oplus b_2 \oplus b_3$	$a_0 \oplus a_1 \oplus a_2 \oplus a_3 \oplus 1$
B_2	$b_0 \oplus b_1 \oplus b_2 \oplus b_3$	$a_0 \oplus a_1 \oplus a_2 \oplus a_3 \oplus 1$

Table 7: Parities of sub-sums V_i and V_i' allow to correct the impact of a one-bit fault on an input share.

Therefore, we can observe in Table 7 that the parity of $V_0 \oplus V_1' \oplus V_2 \oplus V_2'$ (i.e. the sum of sub-sums and corrected sub-sums added to shares A_i) behaves such that, regardless of the position of the one-bit fault, $HW(A_0' \oplus A_1' \oplus A_2' \oplus A_3') \bmod 2 = b_0 \oplus b_1 \oplus b_2 \oplus b_3$. Likewise, the parity of the sum of sub-sums and corrected sub-sums added to shares B_i ($V_0' \oplus V_1 \oplus V_2 \oplus V_2'$) behaves such that $HW(B_0' \oplus B_1' \oplus B_2' \oplus B_3') \bmod 2 = a_0 \oplus a_1 \oplus a_2 \oplus a_3$.

On an input share scale, Table 8 represents the correctness of parities of the modified input shares A_i' and B_i' (obtained after the XOR of variables V_i and V_i' to the initial input shares A_i and B_i). In other words, this table indicates if, after the XOR of sub-sums V_i and of the results of their respective corrections V_i', each modified input share A_i' or B_i' carries the same parity as would have carried the corresponding input share A_i or B_i in a non-faulted environment.

Fault Location \ Modified Input Share	A_0'	A_1'	A_2'	A_3'	B_0'	B_1'	B_2'	B_3'
A_0	✓	✓	✓	✓	✓	✓	✓	✓
A_2	✓	✓	✓	✓	✗	✓	✗	✓
B_0	✗	✓	✗	✓	✗	✓	✗	✓
B_2	✓	✓	✓	✓	✗	✓	✗	✓

Table 8: Correctness of the parity of input shares A_i' and B_i' after the XOR of variables V_i and V_i' in a single fault location case

Moreover, we can notice that modified input shares $A_0', A_1', A_2', A_3', B_0', B_1', B_2'$ and B_3' are not necessarily codewords in the case where a fault has been detected. At this point, the only requirement for the correctness of the following steps of the masking scheme is that the parities of $A_0' \oplus A_1' \oplus A_2' \oplus A_3'$ and $B_0' \oplus B_1' \oplus B_2' \oplus B_3'$ are (not necessarily respectively) $a_0 \oplus a_1 \oplus a_2 \oplus a_3$ and $b_0 \oplus b_1 \oplus b_2 \oplus b_3$.

Furthermore, on a wider scope, when applied on a bitsliced implementation of a cryptographic primitive this design does not correct the original faulted input share(s) A_i or B_i. This can be acceptable if the number of AND instances in the implementation is low, or if only very few faults are introduced in the overall implementation as these faults would be corrected at the beginning of each instance that involves the corresponding faulted share. Otherwise, there will rapidly exist instances of the masking scheme whose inputs would hold more faults that the correction design is able to handle.

To avoid this issue, all input shares A_i or B_i can be replaced at the end of the scheme by codewords of respective parities the parities of B_i' and A_i' (i.e. b_i or a_i). Indeed, as precedently detailed, $HW(A_0' \oplus A_1' \oplus A_2' \oplus A_3') \bmod 2 = b_0 \oplus b_1 \oplus b_2 \oplus b_3$ and $HW(B_0' \oplus B_1' \oplus B_2' \oplus B_3') \bmod 2 = a_0 \oplus a_1 \oplus a_2 \oplus a_3$.

These observations still stand in the case detailed in the subsubsection hereinafter, where two one-bit faults are committed on two different input shares.

9.2.5 One-bit Faults on Two Input Shares

We now consider the case where two one-bit faults are committed on two different input shares among the eight shares A_i and B_i. Table 9 represents the impact on the correctness of sub-sums V_0, V_1 and V_2 of committing a one-bit fault on each share of each couple of two different input shares among A_0, A_2, B_0 and B_2.

Fault Locations		V_0	V_1	V_2
A_0	A_2	✗	✗	✗
A_0	B_0	✓	✓	✗
A_0	B_2	✗	✗	✓
A_2	B_0	✗	✗	✓
A_2	B_2	✓	✓	✗
B_0	B_2	✗	✗	✗

Table 9: Correctness of the parity of variables V_i, depending on locations of two one-bit faults on two input shares

For example, commit a one-bit fault on both input shares A_2 and B_2 will impact both sub-sums V_0 and V_1, but not V_2 (hence $V_2 = V_2'$, i.e. $V_2 \oplus V_2' = 0$). Therefore,

- $HW(A_0' \oplus A_1' \oplus A_2' \oplus A_3') \bmod 2 = HW(A_0 \oplus A_1 \oplus A_2 \oplus A_3 \oplus V_0 \oplus V_1' \oplus V_2 \oplus V_2') \bmod 2 = a_0 \oplus a_1 \oplus (a_2 \oplus 1) \oplus a_3 \oplus (a_0 \oplus a_1 \oplus b_0 \oplus b_1 \oplus 1) \oplus (a_2 \oplus a_3 \oplus b_2 \oplus b_3) = b_0 \oplus b_1 \oplus b_2 \oplus b_3$
- $HW(B_0' \oplus B_1' \oplus B_2' \oplus B_3') \bmod 2 = HW(B_0 \oplus B_1 \oplus B_2 \oplus B_3 \oplus V_0' \oplus V_1 \oplus V_2 \oplus V_2') \bmod 2 = b_0 \oplus b_1 \oplus (b_2 \oplus 1) \oplus b_3 \oplus (a_0 \oplus a_1 \oplus b_0 \oplus b_1) \oplus (a_2 \oplus a_3 \oplus b_2 \oplus b_3 \oplus 1) = a_0 \oplus a_1 \oplus a_2 \oplus a_3$

Accordingly, in spite of these two faults, the parities of $A_0' \oplus A_1' \oplus A_2' \oplus A_3'$ and $B_0' \oplus B_1' \oplus B_2' \oplus B_3'$ carry respectively the correct values $b = b_0 \oplus b_1 \oplus b_2 \oplus b_3$ and $a = a_0 \oplus a_1 \oplus a_2 \oplus a_3$ of the parities of non-faulted inputs B and A.

Subsequently, in the case where the two faults are committed on shares of the same input (i.e. on A_0 and A_2 or on B_0 and B_2), the impacts of these faults will naturally cancel themselves regarding the parity of the sum of the modified shares of this input. Consequently, the parity of the sum of the modified shares of each input will be correct even without correction. Nevertheless, it can be

noted that the correction process still keeps the correctness of the parities of the sums of shares of each input. Indeed, if faults are committed for example on A_0 and A_2, it impacts V_0, V_1 and V_2. Therefore, in this case the parities of respective sums of modified shares A'_i and B'_i verify

- $HW(A'_0 \oplus A'_1 \oplus A'_2 \oplus A'_3) \bmod 2 = HW(A_0 \oplus A_1 \oplus A_2 \oplus A_3 \oplus V_0 \oplus V'_1 \oplus V_2 \oplus V'_2) \bmod 2 = (a_0 \oplus 1) \oplus a_1 \oplus (a_2 \oplus 1) \oplus a_3 \oplus (a_0 \oplus a_1 \oplus b_0 \oplus b_1 \oplus 1) \oplus (a_2 \oplus a_3 \oplus b_2 \oplus b_3) \oplus (a_2 \oplus a_3 \oplus b_0 \oplus b_1) \oplus (a_2 \oplus a_3 \oplus b_0 \oplus b_1 \oplus 1) = b_0 \oplus b_1 \oplus b_2 \oplus b_3$
- $HW(B'_0 \oplus B'_1 \oplus B'_2 \oplus B'_3) \bmod 2 = HW(B_0 \oplus B_1 \oplus B_2 \oplus B_3 \oplus V_0 \oplus V'_1 \oplus V_2 \oplus V'_2) \bmod 2 = b_0 \oplus b_1 \oplus b_2 \oplus b_3 \oplus (a_0 \oplus a_1 \oplus b_0 \oplus b_1) \oplus (a_2 \oplus a_3 \oplus b_2 \oplus b_3 \oplus 1) \oplus (a_2 \oplus a_3 \oplus b_0 \oplus b_1) \oplus (a_2 \oplus a_3 \oplus b_0 \oplus b_1 \oplus 1) = a_0 \oplus a_1 \oplus a_2 \oplus a_3$

Similarly as for Table 8, Table 10 indicates if each of the modified input shares A'_i and B'_i carries the same parity as would have carried the corresponding input share in a non-faulted environment in the two one-bit faults case, depending on the locations of these faults.

Fault Locations		Modified Input Share A'_0	A'_1	A'_2	A'_3	B'_0	B'_1	B'_2	B'_3
A_0	A_2	✓	✓	✓	✓	✗	✓	✗	✓
A_0	B_0	✗	✓	✗	✓	✗	✓	✗	✓
A_0	B_2	✓	✓	✓	✓	✗	✓	✗	✓
A_2	B_0	✗	✓	✗	✓	✓	✓	✓	✓
A_2	B_2	✓	✓	✓	✓	✓	✓	✓	✓
B_0	B_2	✗	✓	✗	✓	✓	✓	✓	✓

Table 10: Correctness of parities of modified input shares A'_i and B'_i after the XOR of variables V_i and V'_i in a two faults locations case

Moreover, it can be noticed that using only two variables V_i would not be enough to ensure both the correctness of parities of input shares and the non-completeness of those sub-sums V_i.

Indeed, to ensure non-completeness not all the shares A_i can figure in the computation of a same sub-sum V_i, and in the same manner not all the shares B_i can figure in the computation of a same sub-sum V_j. Nevertheless, to ensure that all potential input fault locations are covered, all input shares A_i and B_i must feature in at least one sub-sum V_i. Supposing that we employ only two sub-sums, these two conditions combined imply that both sub-sums would comprise at the same time some shares A_i from A and some shares B_i of B. Therefore, it is not possible to attribute with certainty a fault detected on one of these sub-sums to the input where it has been committed initially. That is why

the minimum number of sub-sums needed to cover all potential fault locations while complying with non-completeness is three, as deployed in our scheme.

Furthermore, in this two one-bit faults scenario, each sub-sum naturally carries at most two one-bit faults. In this case, this sub-sum, despite being faulted, represents the correct parity as parities are computed modulo 2. Nevertheless, we impose that the BCH code we use has a correction capacity of at least 2, to avoid the eventuality where a correction of two faults by a BCH code of correction capacity strictly less than two could modify the sub-sum in question and its then-correct parity.

10 A New Fault Resistant Masking Scheme

This section details the design of the masking scheme in three parts. The first subsection describes the input shares correction design, then the second subsection explains the core operations of the AND multiplication. Finally, the construction of the output shares is detailed in the third subsection.

10.1 Input Shares Correction

As explained in subsubsection 9.1.4, we choose to use the BCH code of length $n = 45$ with the maximum dimension. To that end, we consider $m = 12$ the multiplicative order of 2 modulo n, α a 45^{th} primitive root of unity in $\mathbb{F}_{q^m}$ and therefore determine the generator polynomial $g(X)$ of the BCH code we use to be

$$\begin{aligned} g(X) &= lcm(M_\alpha(X), M_{\alpha^2}(X), M_{\alpha^3}(X), M_{\alpha^4}(X)) \\ &= (X^{12} + X^3 + 1) * (X^4 + X + 1) \\ &= X^{16} + X^{13} + X^{12} + X^7 + X^3 + X + 1. \end{aligned}$$

To compute $a \wedge b$ with $a, b \in \mathbb{F}_2$, we represent a and b by codewords A and $B \in \mathbb{F}_2^{45}$ verifying $HW(A) \bmod 2 = a$ and $HW(B) \bmod 2 = b$. We then suppose each of those two input codewords A and B to be split between four shares. In other words, we randomly pick shares A_0, A_1, A_2, A_3, B_0, B_1, B_2 and B_3 that are codewords as well and verify $A = A_0 \oplus A_1 \oplus A_2 \oplus A_3$ and $B = B_0 \oplus B_1 \oplus B_2 \oplus B_3$.

Firstly, we perform code correction on the input shares to prevent any faults introduced at the end of preceding operations or just before the start of the current instance of the scheme. As explained in subsubsection 9.2.2, we must avoid the correction of each of the eight input shares for performance reasons. Consequently, we compute the following three intermediate sub-sums V_0, V_1 and V_2 and correct them into respective codewords V_0', V_1' and V_2':

- $V_0 = A_0 \oplus A_1 \oplus B_0 \oplus B_1$
- $V_1 = A_2 \oplus A_3 \oplus B_2 \oplus B_3$
- $V_2 = A_2 \oplus A_3 \oplus B_0 \oplus B_1$

Subsequently, thanks to these variables V_i and their corresponding corrected codewords V_i', we can correct the parities of shares A_i and B_i by computing modified shares A_i' and B_i' as follows :

- $A'_0 = A_0 \oplus V_0$
- $A'_1 = A_1 \oplus V'_1$
- $A'_2 = A_2 \oplus V_2$
- $A'_3 = A_3 \oplus V'_2$
- $B'_0 = B_0 \oplus V_1$
- $B'_1 = B_1 \oplus V'_0$
- $B'_2 = B_2 \oplus V_2$
- $B'_3 = B_3 \oplus V'_2$

10.2 Array of Subproducts

We consider $n_r = 6$ random polynomials $R_0, R_1, R_2, R_3, R_4, R_5 \in \mathbb{F}_2^{45}$. Moreover, we note $a_i \in \mathbb{F}_2$ the parities of respective input shares A'_i, $b_i \in \mathbb{F}_2$ the parities of respective input shares B'_i and $r_i \in \mathbb{F}_2$ the parities of respective random polynomials R_i. Subsequently, we compute the following array of products of parities, noted mCP :

$$mCP = \begin{bmatrix} r_2b_3 & a_3b_3 & r_1b_1 & a_2b_3 & r_2b_1 & a_3r_4 & r_0 & a_1r_5 & a_1b_1 \\ a_0r_4 & a_2b_1 & r_0b_2 & a_0b_2 & r_5 & r_1r_3 & a_1r_3 & a_1r_4 & a_2r_5 \\ a_2b_2 & a_3r_3 & a_0b_1 & a_0r_3 & a_3b_0 & r_0b_1 & a_1b_0 & r_0r_5 & r_2r_3 \\ r_1r_4 & a_3b_2 & r_2b_2 & a_1b_2 & a_0b_0 & r_1b_0 & r_0b_0 & a_3b_1 & a_0b_3 \\ a_0r_5 & r_2r_5 & r_2b_0 & a_2 & r_1r_5 & a_1b_3 & r_0r_3 & a_2b_0 & a_2r_4 \end{bmatrix}$$

The positioning of the products in the array enables to compute the output shares parities s_i by applying to mCP the following masks $maskS_0$, $maskS_1$, $maskS_2$, $maskS_3 \in \mathbb{F}_2^{45}$. The literal formulas of those output shares parities s_i depending on a_i, b_i and r_i are listed in subsubsection 9.1.5. In this manner, each output parity s_i can be computed the following way :

$$s_i = \sum_{j=0}^{44} (mCP[j] \ \& \ maskS_i[j])$$

These masks, whose values are listed hereinbelow, are also codewords. Therefore, they can be corrected a few times among all the masked AND occurrences of a global bitsliced implementation of a cryptographic primitive.

$$maskS_0 = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$maskS_1 = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \end{bmatrix}$$

$$maskS_2 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$

$$maskS_3 = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

These masks $maskS_i$ are respectively equivalent to the following codewords $ms_i(X)$:

- $ms_0(X) = X^{42} + X^{40} + X^{38} + X^{29} + X^{27} + X^{26} + X^{25} + X^{16} + X^{12} + X^{10} + X^7 + X^4$, with $ms_0 = (X^{26} + X^{24} + X^{23} + X^{21} + X^{19} + X^{18} + X^{16} + X^{15} + X^{13} + X^{12} + X^{10} + X^9 + X^7 + X^6 + X^5 + X^4) * g(X)$
- $ms_1(X) = X^{42} + X^{38} + X^{34} + X^{32} + X^{27} + X^{24} + X^{23} + X^{22} + X^{15} + X^{13} + X^7 + X^5 + X^4 + X^2$, with $ms_1 = (X^{26} + X^{23} + X^{20} + X^{19} + X^{18} + X^{16} + X^{12} + X^{11} + X^{10} + X^6 + X^3 + X^2) * g(X)$
- $ms_2(X) = X^{43}+X^{37}+X^{31}+X^{25}+X^{15}+X^{14}+X^{12}+X^7+X^6+X^5+X^2+1$, with $ms_2 = (X^{27} + X^{24} + X^{23} + X^{19} + X^{18} + X^{16} + X^{14} + X^{13} + X^{11} + X^8 + X^7 + X^6 + X^5 + X^3 + X + 1) * g(X)$
- $ms_3(X) = X^{41} + X^{40} + X^{37} + X^{36} + X^{31} + X^{29} + X^{23} + X^{20} + X^9 + X^7 + X^6 + X^3 + X + 1$, with $ms_3 = (X^{25} + X^{24} + X^{22} + X^{21} + X^{19} + X^{17} + X^{15} + X^{14} + X^{13} + X^{12} + X^{11} + X^9 + X^8 + X^7 + X^6 + 1) * g(X)$

10.3 Output Shares Computation

For $0 \leqslant i \leqslant 3$, we compute the variables $x_{i,0}, x_{i,1}, x_{i,2}, x_{i,3} \in \mathbb{F}_2$ depending on variables a_i, b_i and r_i, such that the following sets are four sets of independent and equiprobable variables.

- $s_0, s_1, s_2, x_{0,0}, x_{0,1}, x_{0,2}$ and $x_{0,3}$
- $s_1, s_0, s_3, x_{1,0}, x_{1,1}, x_{1,2}$ and $x_{1,3}$

- $s_2, s_0, s_3, x_{2,0}, x_{2,1}, x_{2,2}$ and $x_{2,3}$
- $s_3, s_1, s_2, x_{3,0}, x_{3,1}, x_{3,2}$ and $x_{3,3}$

Subsequently, we randomly choose four codewords $C_{0,0}, C_{1,0}, C_{2,0}, C_{3,0} \in \mathbb{F}_2^{45}$ of odd parity, and twenty-four codewords of even parity ($C_{0,j}, C_{1,j}, C_{2,j}, C_{3,j} \in \mathbb{F}_2^{45}$ for $1 \leqslant j \leqslant 6$). Finally, we use those codewords and these sets of variables in $\mathbb{F}_2$ to compute the output shares $S_0, S_1, S_2, S_3 \in \mathbb{F}_2^{45}$ such that

- $S_0 = s_0 * C_{0,0} + s_1 * C_{0,1} + s_2 * C_{0,2} + x_{0,0} * C_{0,3} + x_{0,1} * C_{0,4} + x_{0,2} * C_{0,5} + x_{0,3} * C_{0,6}$
- $S_1 = s_1 * C_{1,0} + s_0 * C_{1,1} + s_3 * C_{1,2} + x_{1,0} * C_{1,3} + x_{1,1} * C_{1,4} + x_{1,2} * C_{1,5} + x_{1,3} * C_{1,6}$
- $S_2 = s_2 * C_{2,0} + s_0 * C_{2,1} + s_3 * C_{2,2} + x_{2,0} * C_{2,3} + x_{2,1} * C_{2,4} + x_{2,2} * C_{2,5} + x_{2,3} * C_{2,6}$
- $S_3 = s_3 * C_{3,0} + s_1 * C_{3,1} + s_2 * C_{3,2} + x_{3,0} * C_{3,3} + x_{3,1} * C_{3,4} + x_{3,2} * C_{3,5} + x_{3,3} * C_{3,6}$

In this manner, each output share S_i verifies uniformity and can equiprobably take $2^7 = 128$ different values. With the codewords $C_{i,j}$ we choose, we obtain the following formulas for the output shares codewords S_i :

$$\begin{aligned} S_0 = (&s_0, s_1 \oplus x_{0,0}, s_2, s_0 \oplus x_{0,1}, s_0 \oplus x_{0,0}, x_{0,2} \oplus x_{0,0}, x_{0,2}, s_1, s_1 \oplus s_2 \oplus x_{0,2} \\ &\oplus x_{0,1}, s_2, x_{0,2}, s_1 \oplus x_{0,1}, s_2, s_1, s_1 \oplus s_2 \oplus x_{0,3} \oplus x_{0,1}, s_2, s_1, s_1 \oplus s_2 \\ &\oplus x_{0,2}, s_2, x_{0,3}, x_{0,3} \oplus x_{0,1}, x_{0,2}, s_1, s_2, s_0 \oplus x_{0,1}, s_0, s_1, s_2 \oplus x_{0,1}, x_{0,2}, \\ &s_0, x_{0,3} \oplus x_{0,1}, s_0 \oplus x_{0,0}, x_{0,3} \oplus x_{0,1}, s_0, x_{0,2}, x_{0,2} \oplus x_{0,0}, x_{0,2} \oplus x_{0,1} \\ &x_{0,3} \oplus x_{0,0}, s_0 \oplus x_{0,0}, x_{0,3}, s_0 \oplus x_{0,0}, s_0, x_{0,3}, x_{0,3} \oplus x_{0,0}, x_{0,3} \oplus x_{0,0}) \end{aligned}$$

$$\begin{aligned} S_1 = (&s_0, s_1, s_3 \oplus x_{1,0}, s_0, s_1, s_1, s_0, x_{1,3} \oplus x_{1,1}, s_3 \oplus x_{1,0}, s_3 \oplus x_{1,1}, x_{1,3}, x_{1,3} \\ &\oplus x_{1,0}, s_3, x_{1,3} \oplus x_{1,1}, s_3 \oplus x_{1,1}, s_3 \oplus s_0 \oplus x_{1,1} \oplus x_{1,1}, s_0 \oplus x_{1,0}, s_3 \\ &\oplus x_{1,1} \oplus x_{1,3}, s_3, x_{1,3}, x_{1,3} \oplus x_{1,0}, s_0, s_1 \oplus x_{1,1}, s_3 \oplus x_{1,1} \oplus x_{1,0}, s_0, s_0, \\ &s_1 \oplus x_{1,1}, s_3 \oplus s_0 \oplus x_{1,1}, s_0, s_1, x_{1,1}, s_1 \oplus x_{1,1}, s_1, x_{1,1}, s_1 \oplus x_{1,0}, x_{1,1}, \\ &x_{1,1} \oplus x_{1,1}, s_1, s_1 \oplus x_{1,0}, x_{1,1}, x_{1,3} \oplus x_{1,0}, x_{1,1} \oplus x_{1,0}, x_{1,1}, x_{1,3}, x_{1,3}) \end{aligned}$$

$$\begin{aligned} S_2 = (&s_3, x_{2,0} \oplus x_{2,2}, s_3, x_{2,3} \oplus x_{2,1}, x_{2,0} \oplus x_{2,1}, x_{2,0} \oplus x_{2,2}, s_2, s_2 \oplus x_{2,2}, s_2 \\ &\oplus x_{2,2}, x_{2,3}, s_0, s_3 \oplus x_{2,1}, s_3 \oplus s_0 \oplus x_{2,3}, s_0 \oplus x_{2,1} \oplus x_{2,2}, s_3, s_2, s_0 \oplus \\ &x_{2,1} \oplus x_{2,2}, x_{2,3} \oplus x_{2,2}, s_2, s_2, s_2 \oplus x_{2,1}, x_{2,3} \oplus x_{2,1}, s_3, s_0, x_{2,3}, s_2, s_2, \\ &s_3 \oplus x_{2,1}, s_2, s_2, s_0, s_3 \oplus s_0 \oplus x_{2,0} \oplus x_{2,2}, s_0 \oplus x_{2,1}, s_3, s_3 \oplus x_{2,2}, s_0 \oplus \\ &x_{2,3} \oplus x_{2,0} \oplus x_{2,1} \oplus x_{2,2}, s_0, x_{2,0}, x_{2,0}, x_{2,3}, x_{2,0}, x_{2,3}, x_{2,3}, x_{2,0}, x_{2,0}) \end{aligned}$$

$$S_3 = (x_{3,2}, s_2, s_1, x_{3,2}, x_{3,2}, s_2, x_{3,2} \oplus x_{3,1}, s_2, s_2 \oplus s_1, x_{3,2} \oplus x_{3,1}, x_{3,2} \oplus x_{3,1} \oplus x_{3,0}, s_1, s_3 \oplus x_{3,0}, s_2 \oplus x_{3,0}, x_{3,3}, s_3, s_2 \oplus s_1 \oplus x_{3,0}, s_2 \oplus x_{3,3}, x_{3,3} \oplus x_{3,1}, s_3, s_1 \oplus x_{3,3}, s_3 \oplus x_{3,1}, s_3, s_1 \oplus x_{3,0}, s_3, s_3, x_{3,3}, x_{3,3} \oplus x_{3,1}, s_3 \oplus x_{3,1}, x_{3,3}, x_{3,3} \oplus x_{3,0}, s_2 \oplus x_{3,0}, x_{3,3} \oplus x_{3,0}, s_3 \oplus x_{3,1}, s_2 \oplus s_1 \oplus x_{3,1}, s_2 \oplus x_{3,0}, x_{3,2} \oplus x_{3,0}, s_3, s_1, s_3, s_1 \oplus x_{3,2}, s_1, x_{3,2} \oplus x_{3,1}, x_{3,2}, x_{3,3})$$

11 Application to a Global Implementation

The AND masking scheme presented above requires that its input and output shares are represented by codewords. Therefore, to be able to apply it on a bitsliced implementation of any cryptographic primitive, the three other Boolean operations (OR, XOR and NOT) need to be addressed considering this requirement on the format of their inputs and outputs.

First of all, the XOR operation is associative with regards to input sharings, hence the input shares of a XOR can be just XORed to one another to compute output shares. In our AND masking scheme set-up where each input is split into four shares such that $HW(A_0 \oplus A_1 \oplus A_2 \oplus A_3) \mod 2 = a$ and $HW(B_0 \oplus B_1 \oplus B_2 \oplus B_3) \mod 2 = b$, the most straightforward manner to compute $a \oplus b$ depending on codewords input shares A_i and B_i is the following :

- $S_0 = A_0 \oplus B_0$
- $S_1 = A_1 \oplus B_1$
- $S_2 = A_2 \oplus B_2$
- $S_3 = A_3 \oplus B_3$

Therefore, the sum of parities of output shares S_i carries the correct value. Indeed,

$$\begin{aligned} HW(S_0 \oplus S_1 \oplus S_2 \oplus S_3) \mod 2 &= HW(A_0 \oplus B_0) + HW(A_1 \oplus B_1) \\ &\quad + HW(A_2 \oplus B_2) + HW(A_3 \oplus B_3) \mod 2 \\ &= HW(A_0 \oplus A_1 \oplus A_2 \oplus A_3) \\ &\quad + HW(B_0 \oplus B_1 \oplus B_2 \oplus B_3) \mod 2 \\ &= a \oplus b \end{aligned}$$

As a result, this XOR implementation is compatible with the new masking scheme, since its input shares are codewords and its output shares S_i are codewords as well, as sums of codewords.

Secondly, regarding the OR operation, it can be noticed that it can be written as a combination of an AND operation and three NOT operations. As a matter of fact,

$$a \vee b = \neg(\neg a \wedge \neg b)$$

Therefore, the OR operation can be implemented using the AND masking scheme developed in this thesis and the NOT implementation described hereinbelow. Thus, the only bitwise operation left to be implemented to be able to apply our AND masking scheme to any cryptographic algorithm is the NOT operation, detailed in the following subsection.

11.1 Implementation of the NOT Operation

As we consider input and output shares of the AND operation to be codewords, input and output shares of the NOT operation need to be codewords as well

to ensure compatibility. Hence, the NOT operation takes as input shares codewords A_0, A_1, A_2 and A_3 and returns codewords S_0, S_1, S_2 and S_3 such that $HW(A_0) + HW(A_1) + HW(A_2) + HW(A_3) \mod 2 = HW(S_0) + HW(S_1) + HW(S_2) + HW(S_3) + 1 \mod 2$.

The basic idea of the implementation is to add random codewords to the three first input shares A_0, A_1 and A_2, then add to A_3 a codeword of parity opposite to the parity of the sum of the random codewords XORed to A_0, A_1 and A_2.

The major drawback to this solution lies in the fact that computing random codewords can rapidly become costly. This can be done either by computing random messages and multiplying them by the generator polynomial of the code $g(X)$, or by testing the residue of random arrays $y \in \mathbb{F}_2^{45}$ modulo $g(X)$ until finding one verifying $y(X) = 0 \mod g(X)$. Moreover, those operations would have to be performed using only bitwise operations as well, so that the implementation of the cryptographic primitive would be exclusively made up of bitwise operations.

To limitate this extra cost, we aim to re-use codewords already involved in our AND masking scheme : the masks $ms_0(X)$, $ms_1(X)$, $ms_2(X)$ and $ms_3(X)$ (listed in subsection 10.2). To that end, we pick random variables $v_{i,j} \in \mathbb{F}_2$ for $0 \leqslant i \leqslant 2$ and $0 \leqslant j \leqslant 3$, and use them in the computation of output shares S_0, S_1, S_2 and S_3 such that :

- $S_0 = A_0 \oplus (v_{0,0} * ms_0(X)) \oplus (v_{0,1} * ms_1(X)) \oplus (v_{0,2} * ms_2(X)) \oplus (v_{0,3} * ms_3(X))$
- $S_1 = A_1 \oplus (v_{1,0} * ms_0(X)) \oplus (v_{1,1} * ms_1(X)) \oplus (v_{1,2} * ms_2(X)) \oplus (v_{1,3} * ms_3(X))$
- $S_2 = A_2 \oplus (v_{2,0} * ms_0(X)) \oplus (v_{2,1} * ms_1(X)) \oplus (v_{2,2} * ms_2(X)) \oplus (v_{2,3} * ms_3(X))$
- $S_3 = A_3 \oplus ((v_{0,0} \oplus v_{1,0} \oplus v_{2,0}) * ms_0(X)) \oplus ((v_{0,1} \oplus v_{1,1} \oplus v_{2,1}) * ms_1(X)) \oplus ((v_{0,2} \oplus v_{1,2} \oplus v_{2,2}) * ms_2(X)) \oplus ((v_{0,3} \oplus v_{1,3} \oplus v_{2,3}) * ms_3(X)) \oplus g(X)$

Thereby $A_0 \oplus A_1 \oplus A_2 \oplus A_3 \oplus S_0 \oplus S_1 \oplus S_2 \oplus S_3 = g(X)$. As detailed in subsubsection 9.1.2, the BCH code generator polynomial $g(X)$ has been chosen to have odd parity. Thus,

$$HW(S_0 \oplus S_1 \oplus S_2 \oplus S_3) \mod 2 = HW(A_0 \oplus A_1 \oplus A_2 \oplus A_3) + 1 \mod 2.$$

In conclusion, this implementation effectively performs the NOT operation with regards to its input sharing.

11.2 Tests

We test the new masking scheme presented in this thesis on a AES implementation with a randomly-chosen fixed key using the TBoxes of [CEJv03] bitsliced

with the Usuba tool ([Mer20]). This implementation is composed of 37586 AND gates, 2293 NOT gates and 66751 XOR gates and can be summarized as follows in algorithm 4 :

Algorithm 4: TBoxed AES With Fixed Key

Input : P, the 128-bit plaintext and RK the round keys
Output: C, the 128-bit ciphertext

for $r = 1$ **to** 9 **do**
  $P \leftarrow ShiftRows(P)$
  $P \leftarrow T_{r,0}[P_0]...T_{r,15}[P_{15}]$
  $P \leftarrow MixColumns(P)$
end for
$C \leftarrow T_{10,0}[P_0]...T_{10,15}[P_{15}]$
return C

- For the 8-bit TBoxes of rounds 1 to 9 ($1 \leqslant r \leqslant 9$), $T_{r,i}[x] = S[x \oplus ShiftRows(RK_{r-1})_i]$
- For the 8-bit TBoxes of round 10, $T_{10,i}[x] = S[x \oplus ShiftRows(RK_9)_i] \oplus RK_{10,i}$
- The AES ShiftRows transformation can be considered as a 128-bit bitwise permutation.
- The AES MixColumn transformation can also be be considered as a 128-bit bitwise operation.

We test with a processor Intel(R) Core(TM) i7-6600U CPU @ 2.60GHz three different versions of this implementation :

- The raw bitsliced implementation where the TBoxes are bitsliced and the ShiftRows and MixColumn operations are implemented with Boolean operations AND, OR, NOT and XOR as well.
- The bitsliced implementation where the NOT operation is performed by flipping a bit share, the XOR operation is implemented by XORing shares and the AND operation is masked by the ISW masking scheme ([ISW03]), that is the most known and studied Boolean AND masking scheme but does not offer fault-attack resistance.
- The bitsliced implementation where the NOT operation is performed as described in subsection 11.1, the XOR operation is implemented by XORing polynomial shares and the AND operation is masked by the new masking scheme presented in this paper.

We obtain the following values:

	Raw Bitsliced Implementation	Implementation Masked With ISW [ISW03]	Implementation Masked With Our Scheme
Time for 1000 executions	0.29 s	1103.12 s	2123.46 s
Binary Size	2.3 MB	3.6 MB	3.8 MB

These results show that masking a bitsliced implementation entails an increase of the binary size and particularly of the execution time, even when using a classic and well-studied masking scheme as ISW. It can also be noticed that, thanks to the use in the implementation of macros that are inherited from the bitslicing by Usuba, the binary size of the implementation masked with the new scheme remains almost equal to the binary size of the implementation masked with ISW. Likewise, the execution time of the implementation masked with the new scheme only doubles compared to the execution time of the implementation masked with ISW, although the new scheme offers the additional property of fault correction compared to ISW.

Conclusion

The first purpose of this PhD study is to develop enhancements and new tools for the white-box cryptographic needs of the Kudelski Group. In this dissertation, we first describe the study of the white-boxability of the NIST Lightweight Cryptography Standardization Contest finalists. Indeed, lightweight cryptography primitives are now widely deployed on devices with constrained capacities that are vulnerable to white-box attackers. Therefore, we aim to determine the finalist submission of the contest that is the most suitable to a white-box implementation : GIFT-COFB, and more precisely GIFT, its cryptographic core block.

Many white-box implementations are based on bitsliced implementations composed themselves of Boolean operations, including some of the most resistant challenges submitted to white-box cryptography contests WhibOx 2017 and WhibOx 2019. Masking schemes being by nature countermeasures to side-channel attacks, we then develop a new masking scheme composed of Boolean operations that is resistant to fault attacks. More precisely, this scheme can be applied on implementations of any primitive, and corrects potential faults without deteriorating or aborting the computation. These implementations will then always return results, results that are correct even if a fault has been introduced during computation. This constitutes a huge practical benefit in the field of fault-resistant masking schemes.

To that end, we use BCH error-correcting codes as BCH decoding can be performed with only Boolean operations as well. The design rationale behind the masking scheme together with the description of the scheme are detailed in the second part of the dissertation. Finally, the masking scheme is applied to a bitsliced implementation of AES and compared to the raw bitsliced implementation and the implementation using the masking scheme ISW.

Directions for Future Research

The different considerations for further research that this dissertation has led us to are the following :

Investigate the Possibility of Using Other Error-Correcting Codes

The error-correcting code used in the design of the masking scheme presented in this dissertation is the BCH error-correcting code, chosen for its easy management of codewords parity because of its cyclicity and for its decoding process that can be implemented using Boolean operations. Nevertheless, using such type error-correcting code while complying with the properties needed for the scheme imply a BCH error-correcting code of rather high length, here chosen to be 45.

It would be relevant to investigate if other types of error-correcting codes would be suitable to comply with the properties needed while using a smaller code length.

Experimentally and Theoretically Prove the Side-Channel Security of the Masking Scheme

The uniformity, non-completeness and correctness properties ensure the first-order probing security of the masking scheme, so we would need further analysis to determine the exact probing security order of the scheme.

In the same way, it would be relevant to experimentally confirm the side-channel resistance of an implementation of a cryptographic primitive using the presented AND masking scheme.

Balance Between Performance and Correction Efficiency

It can easily be noticed that the correction design is the most costly part of the masking schemes, whether in performances or memory. Therefore, we can ask ourselves if, throughout a whole bitsliced implementation, it is possible to remove the correction part at the beginning of some of the instances of the AND masking scheme to improve performances, while not impacting too much the fault correction capacity of this implementation.

A Attack Numerical Example

This appendix will detail the different steps of the attack on a TBox mentioned in subsection 8.2. To that end, we keep the notations of the intermediate values of a TBox detailed in subsection 8.1.

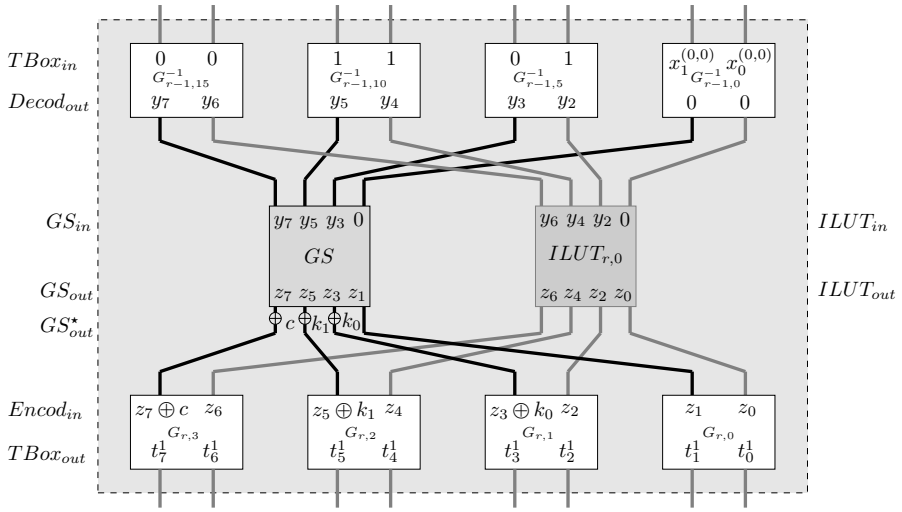
A.1 First Step of the Attack

The first step of the attack illustrates how to deduce the left half of the input encodings of a TBox.

We consider the right-most TBox T_0 of a round r, with key bits $(k_1, k_0) = (1, 1)$, round constant bit $c = 0$, the intern SBox $ILUT = [0110, 1100, 0011, 1111, 0010, 0001, 1010, 0111, 1011, 1101, 1000, 0101, 0000, 1001, 1110, 0100]$ and the input and output encodings

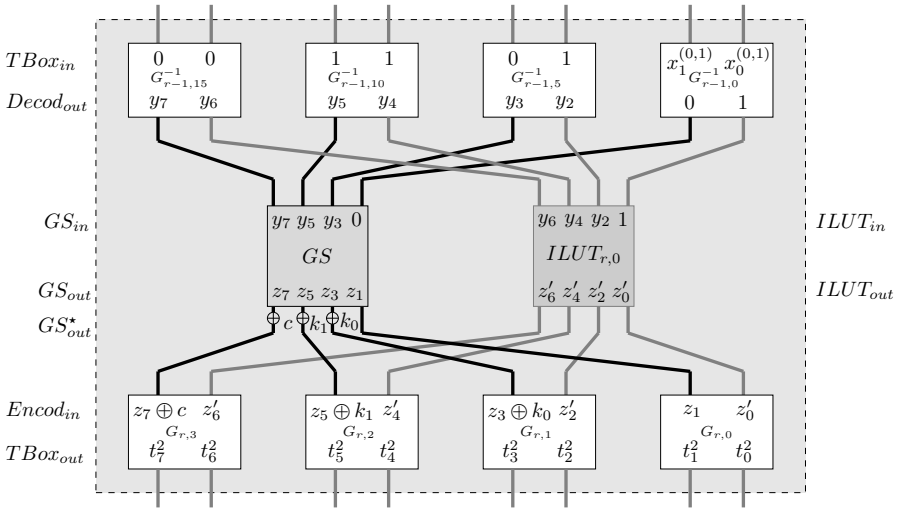
$$\begin{cases} G^{-1}_{r-1,15} = [01, 11, 00, 10] \\ G^{-1}_{r-1,10} = [00, 10, 01, 11] \\ G^{-1}_{r-1,5} = [00, 11, 10, 01] \\ G^{-1}_{r-1,0} = [10, 00, 11, 01] \end{cases}, \quad \begin{cases} G_{r,3} = [11, 10, 01, 00] \\ G_{r,2} = [10, 11, 00, 01] \\ G_{r,1} = [01, 10, 11, 00] \\ G_{r,0} = [10, 01, 11, 00] \end{cases}.$$

First of all, from an attacker viewpoint, we randomly consider the values of the six left-most input bits $(x_7, x_6, x_5, x_4, x_3, x_2)$ to be $(0, 0, 1, 1, 0, 1)$. By the bijectivity of encodings, there exists a unique value $(x_1^{(0,0)}, x_0^{(0,0)}) \in \mathbb{F}_2^2$ such that $G^{-1}_{r-1,0}(x_1^{(0,0)} x_0^{(0,0)}) = 00$. Thus, the computation of $T_0[00\,11\,01\,x_1^{(0,0)} x_0^{(0,0)}]$ presents the following intermediate values :

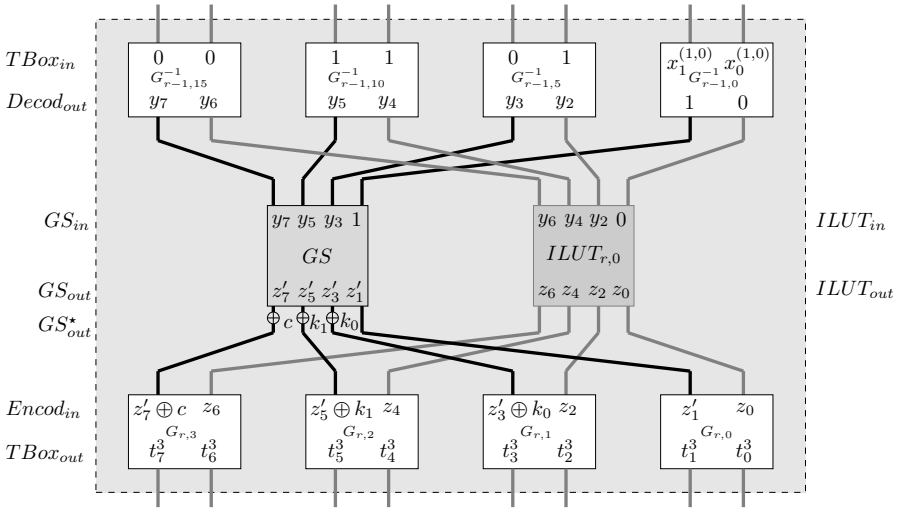


Likewise, there exists distinct input bits values $(x_1^{(0,1)}, x_0^{(0,1)})$, $(x_1^{(1,0)}, x_0^{(1,0)})$, $(x_1^{(1,1)}, x_0^{(1,1)}) \in \mathbb{F}_2^2$ such that $G_{r-1,0}^{-1}(x_1^{(y_1,y_0)} x_0^{(y_1,y_0)}) = y_1 y_0$ for all $(y_1, y_0) \in \mathbb{F}_2^2$. Therefore, the computations of $T_0[00\,11\,01\,x_1^{(0,1)}x_0^{(0,1)}]$, $T_0[00\,11\,01\,x_1^{(1,0)}x_0^{(1,0)}]$, $T_0[00\,11\,01\,x_1^{(1,1)}x_0^{(1,1)}]$ present the respective following intermediate values :

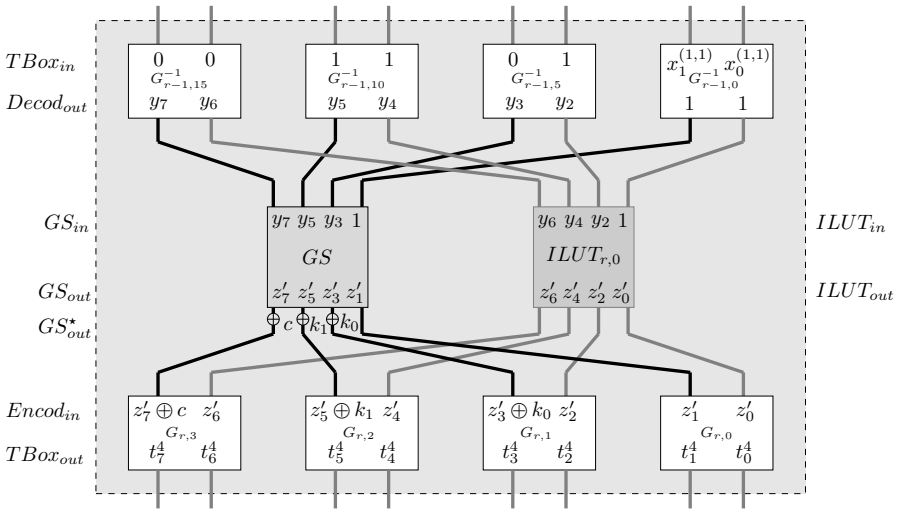
- For $T_0[00\,11\,01\,x_1^{(0,1)}x_0^{(0,1)}]$:



- For $T_0[00\,11\,01\,x_1^{(1,0)}x_0^{(1,0)}]$:



- For $T_0[00\,11\,01\,x_1^{(1,1)}x_0^{(1,1)}]$:



We can subsequently compute the values of $\mathcal{T} = \{\, T_0[00\,11\,01\,x_1^{(0,0)}x_0^{(0,0)}], T_0[00\,11\,01\,x_1^{(0,1)}x_0^{(0,1)}], T_0[00\,11\,01\,x_1^{(1,0)}x_0^{(1,0)}], T_0[00\,11\,01\,x_1^{(1,1)}x_0^{(1,1)}] \,\} = \{\, T_0[00\,11\,01\,00], T_0[00\,11\,01\,01], T_0[00\,11\,01\,10], T_0[00\,11\,01\,11] \,\}$. As a result,

$$
\begin{aligned}
&\{(00\ 01\ 00\ 11),\\
&\ (10\ 01\ 10\ 11),\\
&\ (01\ 01\ 11\ 11),\\
&\ (11\ 01\ 01\ 11)\}
\end{aligned}
=
\begin{aligned}
&\{(t_7^1t_6^1\ t_5^1t_4^1\ t_3^1t_2^1\ t_1^1t_0^1),\\
&\ (t_7^2t_6^2\ t_5^2t_4^2\ t_3^2t_2^2\ t_1^2t_0^2),\\
&\ (t_7^3t_6^3\ t_5^3t_4^3\ t_3^3t_2^3\ t_1^3t_0^3),\\
&\ (t_7^4t_6^4\ t_5^4t_4^4\ t_3^4t_2^4\ t_1^4\ t_0^4)\}
\end{aligned}
$$

$$
=
\begin{aligned}
&\{(G_{r,3}(\!(z_7{\oplus}c)z_6)G_{r,2}(\!(z_5{\oplus}k_1)z_4)G_{r,1}(\!(z_3{\oplus}k_0)z_2)G_{r,0}(z_1z_0)),\\
&\ (G_{r,3}(\!(z_7{\oplus}c)z_6')G_{r,2}(\!(z_5{\oplus}k_1)z_4')G_{r,1}(\!(z_3{\oplus}k_0)z_2')G_{r,0}(z_1z_0')),\\
&\ (G_{r,3}(\!(z_7'{\oplus}c)z_6)G_{r,2}(\!(z_5'{\oplus}k_1)z_4)G_{r,1}(\!(z_3'{\oplus}k_0)z_2)G_{r,0}(z_1'z_0')),\\
&\ (G_{r,3}(\!(z_7'{\oplus}c)z_6')G_{r,2}(\!(z_5'{\oplus}k_1)z_4')G_{r,1}(\!(z_3'{\oplus}k_0)z_2')G_{r,0}(z_1'z_0'))\}
\end{aligned}
$$

$$
=
\begin{aligned}
&\{T_0[00\,11\,01\,x_1^{(0,0)}x_0^{(0,0)}],\\
&\ T_0[00\,11\,01\,x_1^{(0,1)}x_0^{(0,1)}],\\
&\ T_0[00\,11\,01\,x_1^{(1,0)}x_0^{(1,0)}],\\
&\ T_0[00\,11\,01\,x_1^{(1,1)}x_0^{(1,1)}]\}
\end{aligned}
$$

$$
=
\begin{aligned}
&\{T_0[00\,11\,01\,00],\\
&\ T_0[00\,11\,01\,01],\\
&\ T_0[00\,11\,01\,10],\\
&\ T_0[00\,11\,01\,11]\}
\end{aligned}
$$

Although it is not possible to determine yet $i \in \{0,1,2,3\}$ such that (00 01 00 11) $= (t_7^i t_6^i\ t_5^i t_4^i\ t_3^i t_2^i\ t_1^i t_0^i)$, as $G_{r,3}((z_7 \oplus c)z_6) \neq G_{r,3}((z_7 \oplus c)z_6') \neq G_{r,3}((z_7' \oplus c)z_6) \neq G_{r,3}((z_7' \oplus c)z_6')$ it can be deduced by the bijectivity of $G_{r,3}$ that $z_7' \neq z_7$ (and $z_6' \neq z_6$), i.e. $z_7' = \overline{z_7}$.

Furthermore, $G_{r,2}((z_5 \oplus k_1)z_4) = G_{r,2}((z_5 \oplus k_1)z_4') = G_{r,2}((z_5' \oplus k_1)z_4) = G_{r,2}((z_5' \oplus k_1)z_4')$, hence $z_5' = z_5$ and $z_4' = z_4$. In the same manner, it can be determined that $z_3' = \overline{z_3}, z_2' = \overline{z_2}, z_1' = z_1$ and $z_0' = z_0$. Consequently, $GS[y_7y_5y_30] = z_7z_5z_3z_1$ implies that y_7, y_5 and y_3 verify that $GS[y_7y_5y_31] = \overline{z_7}z_5\overline{z_3}z_1$. According to the values of the SBox GS it can then be deduced that the unique possible value of (y_7, y_5, y_3) is $(0,1,1)$. Therefore $G_{r-1,15}^{-1}(00) = 0\_, G_{r-1,10}^{-1}(11) = 1\_$ and $G_{r-1,5}^{-1}(01) = 1\_$.

This method can be repeated while changing the six fixed input bits and potentially the varying 2-bit input block to determine the remaining left bit values of the four input encodings. For example, considering $(x_7, x_6, x_5, x_4, x_3, x_2) = (1,0,0,1,1,0)$, the output values of the TBox are

$$\begin{array}{ll} \{(t_7^1t_6^1\ t_5^1t_4^1\ t_3^1t_2^1\ t_1^1t_0^1), & \{(01\ 01\ 00\ 11), \\ (t_7^2t_6^2\ t_5^2t_4^2\ t_3^2t_2^2\ t_1^2t_0^2), & (11\ 01\ 10\ 11), \\ (t_7^3t_6^3\ t_5^3t_4^3\ t_3^3t_2^3\ t_1^3t_0^3), & (00\ 01\ 11\ 11), \\ (t_7^4t_6^4\ t_5^4t_4^4\ t_3^4t_2^4\ t_1^4\ t_0^4)\} & (10\ 01\ 01\ 11)\,\} \end{array} = \begin{array}{l} \{(01\ 01\ 00\ 11), \\ (11\ 01\ 10\ 11), \\ (00\ 01\ 11\ 11), \\ (10\ 01\ 01\ 11)\,\} \end{array}$$

Thus, noting $GS[y_7y_5y_30] = z_7z_5z_3z_1$ implies that $GS[y_7y_5y_31] = \overline{z_7}z_5\overline{z_3}z_1$. Therefore, according to the values of GS, it can be deduced that $G_{r-1,15}^{-1}(10) = 0\_, G_{r-1,10}^{-1}(01) = 1\_$ and $G_{r-1,5}^{-1}(10) = 1\_$.

Subsequently, considering $(x_7, x_6, x_5, x_4, x_1, x_0) = (0,0,1,0,0,0)$, the output values of the TBox are

$$\begin{array}{ll} \{(t_7^1t_6^1\ t_5^1t_4^1\ t_3^1t_2^1\ t_1^1t_0^1), & \{(01\ 00\ 01\ 10), \\ (t_7^2t_6^2\ t_5^2t_4^2\ t_3^2t_2^2\ t_1^2t_0^2), & (00\ 11\ 00\ 10), \\ (t_7^3t_6^3\ t_5^3t_4^3\ t_3^3t_2^3\ t_1^3t_0^3), & (01\ 10\ 11\ 10), \\ (t_7^4t_6^4\ t_5^4t_4^4\ t_3^4t_2^4\ t_1^4\ t_0^4)\} & (00\ 01\ 01\ 10)\,\} \end{array} = \begin{array}{l} \{(01\ 00\ 01\ 10), \\ (00\ 11\ 00\ 10), \\ (01\ 10\ 11\ 10), \\ (00\ 01\ 01\ 10)\,\} \end{array}$$

Therefore, when noting $GS[y_7y_5y_01y_1] = z_7z_5z_3z_1$, y_7, y_5 and y_1 verify that $GS[y_7y_5y_11y_1] = z_7'\overline{z_5z_3}z_1$. According to the values of GS, it can be deduced that (y_7, y_5, y_1) is either equal to $(0,0,1)$, $(0,1,1)$ or $(1,1,1)$. Thus $y_1 = 1$, i.e. $G_{r-1,0}^{-1}(00) = 1\_$.

Lastly, considering $(x_7, x_6, x_3, x_2, x_1, x_0) = (1,1,1,1,1,0)$, the output values of the TBox are

$$\begin{array}{ll} \{(t_7^1t_6^1\ t_5^1t_4^1\ t_3^1t_2^1\ t_1^1t_0^1), & \{(01\ 00\ 11\ 00), \\ (t_7^2t_6^2\ t_5^2t_4^2\ t_3^2t_2^2\ t_1^2t_0^2), & (11\ 11\ 11\ 01), \\ (t_7^3t_6^3\ t_5^3t_4^3\ t_3^3t_2^3\ t_1^3t_0^3), & (01\ 10\ 11\ 11), \\ (t_7^4t_6^4\ t_5^4t_4^4\ t_3^4t_2^4\ t_1^4\ t_0^4)\} & (11\ 01\ 11\ 10)\,\} \end{array} = \begin{array}{l} \{(01\ 00\ 11\ 00), \\ (11\ 11\ 11\ 01), \\ (01\ 10\ 11\ 11), \\ (11\ 01\ 11\ 10)\,\} \end{array}$$

Then, $GS[y_7 0 y_3 y_1] = z_7 z_5 z_3 z_1$ and $GS[y_7 0 y_3 y_1] = z'_7 z'_5 z_3 \overline{z_1}$, which implies $(y_7, y_3, y_1) = (0,0,1), (0,1,1), (1,0,1)$ or $(1,1,1)$. Therefore $y_1 = 1$, i.e. $G^{-1}_{r-1,0}(10) = 1\_$.

As a conclusion, all left bits of the four input encodings can be recovered by applying the same method with different input bits. Therefore, in the current exemple, the attacker can recover the following left bits of the four input encodings $G^{-1}_{r-1,15}$, $G^{-1}_{r-1,10}$, $G^{-1}_{r-1,5}$ and $G^{-1}_{r-1,0}$:

$G^{-1}_{r-1,15}$:	$G^{-1}_{r-1,10}$:	$G^{-1}_{r-1,5}$:	$G^{-1}_{r-1,0}$:
$00 \to 0\_$	$00 \to 0\_$	$00 \to 0\_$	$00 \to 1\_$
$01 \to 1\_$	$01 \to 1\_$	$01 \to 1\_$	$01 \to 0\_$
$10 \to 0\_$	$10 \to 0\_$	$10 \to 1\_$	$10 \to 1\_$
$11 \to 1\_$	$11 \to 1\_$	$11 \to 0\_$	$11 \to 0\_$

A.2 Second and Final Steps of the Attack

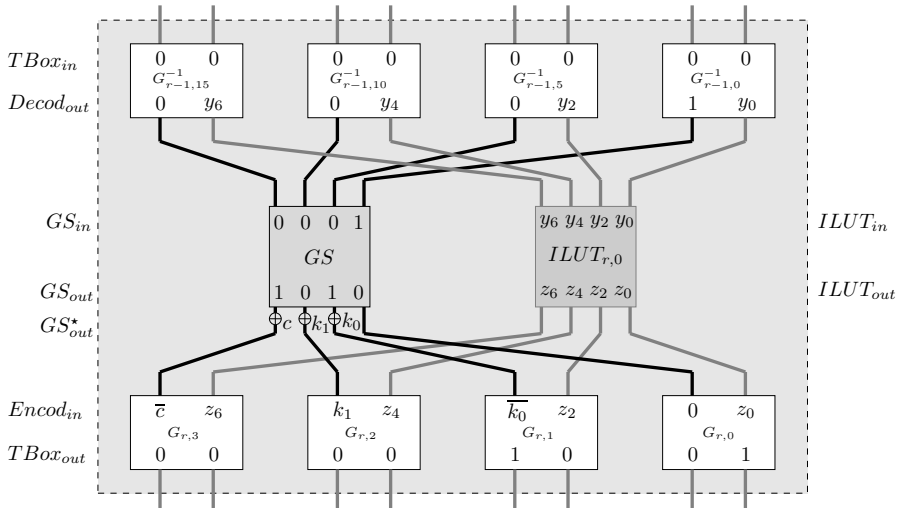
We now demonstrate how to deduce the two key bits of a TBox knowing the left half of the inverse of its output encodings. To that end, we still consider the TBox T_0 with the same notations, and suppose that T_0 does not belong to the last round of GIFT. We assume that, thanks to the procedure above, the attacker knows that the input encodings verify

$G^{-1}_{r-1,15}$:	$G^{-1}_{r-1,10}$:	$G^{-1}_{r-1,5}$:	$G^{-1}_{r-1,0}$:
$00 \to 0\_$	$00 \to 0\_$	$00 \to 0\_$	$00 \to 1\_$
$01 \to 1\_$	$01 \to 1\_$	$01 \to 1\_$	$01 \to 0\_$
$10 \to 0\_$	$10 \to 0\_$	$10 \to 1\_$	$10 \to 1\_$
$11 \to 1\_$	$11 \to 1\_$	$11 \to 0\_$	$11 \to 0\_$

Likewise, we suppose that the attacker knows the left halves of $G^{-1}_{r,2}$ and $G^{-1}_{r,1}$, the inverses of the middle output encodings $G_{r,2}$ and $G_{r,1}$ of T_0. They are obtained by applying the same procedure to the corresponding TBoxes of round $r+1$ that admit $G^{-1}_{r,2}$ and $G^{-1}_{r,1}$ as input encodings, and they provide information on the output encodings of T_0 themselves :

$G^{-1}_{r,2}$:		$G_{r,2}$:		$G^{-1}_{r,1}$:		$G_{r,1}$:
$00 \to 1\_$		$0\_ \to 10$		$00 \to 1\_$		$0\_ \to 01$
$01 \to 1\_$	$\Rightarrow$	$0\_ \to 11$	and	$01 \to 0\_$	$\Rightarrow$	$0\_ \to 10$
$10 \to 0\_$		$1\_ \to 00$		$10 \to 0\_$		$1\_ \to 00$
$11 \to 0\_$		$1\_ \to 01$		$11 \to 1\_$		$1\_ \to 11$

Finally, this knowledge on the input and output encodings is sufficient to conclude the attack. Indeed, it can be used to determine with certainty some of the intermediate values of the computation of T_0 for any randomly chosen input. For example, the computation of $T_0[00\,00\,00\,00]$ presents the following intermediate values :



Consequently, $G_{r,2}(k_1 z_4) = 00$ and $G_{r,1}(\overline{k_0} z_2) = 10$. Therefore, by the bijectivity of the output encodings $G_{r,2}$ and $G_{r,1}$, the values of key bits k_1 and k_0 verify $(k_1, \overline{k_0}) = (1, 0)$, i.e. $(k_1, k_0) = (1, 1)$.

B References

[ABO09] Daniel Augot, Emanuele Betti, and Emmanuela Orsini. An introduction to linear and cyclic codes. In Massimiliano Sala, Shojiro Sakata, Teo Mora, Carlo Traverso, and Ludovic Perret, editors, *Gröbner Bases, Coding, and Cryptography*, pages 47–68. Springer, 2009.

[ADN+22] Amund Askeland, Siemen Dhooghe, Svetla Nikova, Vincent Rijmen, and Zhenda Zhang. Guarding the first order: The rise of AES maskings. In Ileana Buhan and Tobias Schneider, editors, *Smart Card Research and Advanced Applications - 21st International Conference, CARDIS 2022, Birmingham, UK, November 7-9, 2022, Revised Selected Papers*, volume 13820 of *Lecture Notes in Computer Science*, pages 103–122. Springer, 2022.

[AT20] Estuardo Alpirez Bock and Alexander Treff. Security assessment of white-box design submissions of the CHES 2017 CTF challenge. In Guido Marco Bertoni and Francesco Regazzoni, editors, *COSADE 2020*, volume 12244 of *LNCS*, pages 123–146. Springer, Heidelberg, April 2020.

[BB05] David Brumley and Dan Boneh. Remote timing attacks are practical. *Comput. Networks*, 48(5):701–716, 2005.

[BBD+14] Shivam Bhasin, Nicolas Bruneau, Jean-Luc Danger, Sylvain Guilley, and Zakaria Najm. Analysis and improvements of the DPA contest v4 implementation. In Rajat Subhra Chakraborty, Vashek Matyas, and Patrick Schaumont, editors, *Security, Privacy, and Applied Cryptography Engineering - 4th International Conference, SPACE 2014, Pune, India, October 18-22, 2014. Proceedings*, volume 8804 of *Lecture Notes in Computer Science*, pages 201–218. Springer, 2014.

[BBdS+20] Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großschädl, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, and Qingju Wang. Lightweight AEAD and hashing using the sparkle permutation family. *IACR Trans. Symmetric Cryptol.*, 2020(S1):208–261, 2020.

[BBP+16] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. Randomness complexity of private circuits for multiplication. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 616–648. Springer, Heidelberg, May 2016.

[BCC+14] Julien Bringer, Claude Carlet, Hervé Chabanne, Sylvain Guilley, and Houssem Maghrebi. Orthogonal direct sum masking - A smart-

card friendly computation paradigm in a code, with builtin protection against side-channel and fault attacks. In David Naccache and Damien Sauveron, editors, *Information Security Theory and Practice. Securing the Internet of Things - 8th IFIP WG 11.2 International Workshop, WISTP 2014, Heraklion, Crete, Greece, June 30 - July 2, 2014. Proceedings*, volume 8501 of *Lecture Notes in Computer Science*, pages 40–56. Springer, 2014.

[BCD+20] Zhenzhen Bao, Avik Chakraborti, Nilanjan Datta, Jian Guo, Mridul Nandi, Thomas Peyrin, and Kan Yasuda. Photon-beetle authenticated encryption and hash family, 2020.

[BCI+20] Subhadeep Banik, Avik Chakraborti, Tetsu Iwata, Kazuhiko Minematsu, Mridul Nandi, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. GIFT-COFB. *IACR Cryptol. ePrint Arch.*, page 738, 2020.

[BCPZ16] Alberto Battistello, Jean-Sébastien Coron, Emmanuel Prouff, and Rina Zeitoun. Horizontal side-channel attacks and countermeasures on the ISW masking scheme. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *CHES 2016*, volume 9813 of *LNCS*, pages 23–39. Springer, Heidelberg, August 2016.

[BDF+17] Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. Parallel implementations of masking schemes and the bounded moment leakage model. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, pages 535–566. Springer, Heidelberg, April / May 2017.

[BFGV12] Josep Balasch, Sebastian Faust, Benedikt Gierlichs, and Ingrid Verbauwhede. Theory and practice of a leakage resilient masking scheme. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 758–775. Springer, Heidelberg, December 2012.

[BGEC04] Olivier Billet, Henri Gilbert, and Charaf Ech-Chatbi. Cryptanalysis of a white box AES implementation. In Helena Handschuh and Anwar Hasan, editors, *SAC 2004*, volume 3357 of *LNCS*, pages 227–240. Springer, Heidelberg, August 2004.

[Bih97] Eli Biham. A fast new DES implementation in software. In Eli Biham, editor, *FSE'97*, volume 1267 of *LNCS*, pages 260–272. Springer, Heidelberg, January 1997.

[Bil15] Begül Bilgin. *Threshold implementations : as countermeasure against higher-order differential power analysis*. PhD thesis, University of Twente, Enschede, Netherlands, 2015.

[BLL15] Zhenzhen Bao, Peng Luo, and Dongdai Lin. Bitsliced implementations of the PRINCE, LED and RECTANGLE block ciphers on AVR 8-bit microcontrollers. In Sihan Qing, Eiji Okamoto, Kwangjo Kim, and Dongmei Liu, editors, *ICICS 15*, volume 9543 of *LNCS*, pages 18–36. Springer, Heidelberg, December 2015.

[BPP+17a] Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. GIFT: A small present - towards reaching the limit of lightweight encryption. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 321–345. Springer, 2017.

[BPP+17b] Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. GIFT: A small present - towards reaching the limit of lightweight encryption. In Wieland Fischer and Naofumi Homma, editors, *CHES 2017*, volume 10529 of *LNCS*, pages 321–345. Springer, Heidelberg, September 2017.

[BRC60a] R.C. Bose and D.K. Ray-Chaudhuri. Further results on error correcting binary group codes. *Information and Control*, 3(3):279–290, 1960.

[BRC60b] R.C. Bose and D.K. Ray-Chaudhuri. On a class of error correcting binary group codes. *Information and Control*, 3(1):68–79, 1960.

[Cad05] Tom Caddy. Side-channel attacks. In Henk C. A. van Tilborg, editor, *Encyclopedia of Cryptography and Security*. Springer, 2005.

[Cas22] Gaëtan Cassiers. *Composable and efficient masking schemes for side-channel secure implementations*. PhD thesis, Catholic University of Louvain, Louvain-la-Neuve, Belgium, 2022.

[CCG+19] Wei Cheng, Claude Carlet, Kouassi Goli, Jean-Luc Danger, and Sylvain Guilley. Detecting faults in inner-product masking scheme - IPM-FD: IPM with fault detection. In Karine Heydemann, Ulrich Kühne, and Letitia Li, editors, *Proceedings of 8th International Workshop on Security Proofs for Embedded Systems, PROOFS 2019, colocated with CHES 2018, Atlanta, GA, USA, August 24, 2019*, volume 11 of *Kalpa Publications in Computing*, pages 17–32. EasyChair, 2019.

[CEJv03] Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot. White-box cryptography and an AES implementation. In Kaisa Nyberg and Howard M. Heys, editors, *SAC 2002*, volume 2595 of *LNCS*, pages 250–270. Springer, Heidelberg, August 2003.

[CEJvO02] Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot. White-box cryptography and an AES implementation. In Kaisa Nyberg and Howard M. Heys, editors, *Selected Areas in Cryptography, 9th Annual International Workshop, SAC 2002, St. John's, Newfoundland, Canada, August 15-16, 2002. Revised Papers*, volume 2595 of *Lecture Notes in Computer Science*, pages 250–270. Springer, 2002.

[CG16] Claude Carlet and Sylvain Guilley. Complementary dual codes for counter-measures to side-channel attacks. *Advances in Mathematics of Communications*, 10:131–150, 03 2016.

[CG22] Alex Charlès and Chloé Gravouil. Review of the white-box encodability of NIST lightweight finalists. *IACR Cryptol. ePrint Arch.*, page 804, 2022.

[CGM19] Claude Carlet, Sylvain Guilley, and Sihem Mesnager. Direct sum masking as a countermeasure to side-channel and fault injection attacks. In José Luis Hernández Ramos and Antonio F. Skarmeta, editors, *Security and Privacy in the Internet of Things: Challenges and Solutions*, volume 27 of *Ambient Intelligence and Smart Environments*, pages 148–166. IOS Press, 2019.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 398–412. Springer, Heidelberg, August 1999.

[CRZ13] Guilhem Castagnos, Soline Renner, and Gilles Zémor. High-order masking by using coding theory and its application to AES. In Martijn Stam, editor, *14th IMA International Conference on Cryptography and Coding*, volume 8308 of *LNCS*, pages 193–212. Springer, Heidelberg, December 2013.

[DDF14] Alexandre Duc, Stefan Dziembowski, and Sebastian Faust. Unifying leakage models: From probing attacks to noisy leakage. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 423–440. Springer, Heidelberg, May 2014.

[DEM+20] Christoph Dobraunig, Maria Eichlseder, Stefan Mangard, Florian Mendel, Bart Mennink, Robert Primas, and Thomas Unterluggauer. Isap v2.0. *IACR Trans. Symmetric Cryptol.*, 2020(S1):390–416, 2020.

[DEMS21] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2: Lightweight authenticated encryption and hashing. *J. Cryptol.*, 34(3):33, 2021.

[DHP+20] Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Xoodyak, a lightweight cryptographic scheme. *IACR Trans. Symmetric Cryptol.*, 2020(S1):60–87, 2020.

[GD17] Ashrujit Ghoshal and Thomas De Cnudde. Several masked implementations of the Boyar-Peralta AES S-box. In Arpita Patra and Nigel P. Smart, editors, *INDOCRYPT 2017*, volume 10698 of *LNCS*, pages 384–402. Springer, Heidelberg, December 2017.

[GIK+20] Chun Guo, Tetsu Iwata, Mustafa Khairallah, Kazuhiko Minematsu, and Thomas Peyrin. Romulus v1.3, 2020.

[GJRS18] Dahmun Goudarzi, Anthony Journault, Matthieu Rivain, and François-Xavier Standaert. Secure multiplication for bitslice higher-order masking: Optimisation and comparison. In Junfeng Fan and Benedikt Gierlichs, editors, *COSADE 2018*, volume 10815 of *LNCS*, pages 3–22. Springer, Heidelberg, April 2018.

[GPRW20] Louis Goubin, Pascal Paillier, Matthieu Rivain, and Junwei Wang. How to reveal the secrets of an obscure white-box implementation. *J. Cryptogr. Eng.*, 10(1):49–66, 2020.

[GR16] Dahmun Goudarzi and Matthieu Rivain. On the multiplicative complexity of Boolean functions and bitsliced higher-order masking. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *CHES 2016*, volume 9813 of *LNCS*, pages 457–478. Springer, Heidelberg, August 2016.

[Gra23] Chloé Gravouil. A new generic fault resistant masking scheme using error-correcting codes. *IACR Cryptol. ePrint Arch.*, page 118, 2023.

[GRW20] Louis Goubin, Matthieu Rivain, and Junwei Wang. Defeating state-of-the-art white-box countermeasures with advanced gray-box attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):454–482, 2020.

[GT04] Christophe Giraud and Hugues Thiebeauld. A survey on fault attacks. In Jean-Jacques Quisquater, Pierre Paradinas, Yves Deswarte, and Anas Abou El Kalam, editors, *Smart Card Research and Advanced Applications VI, IFIP 18th World Computer Congress, TC8/WG8.8 & TC11/WG11.2 Sixth International Conference on Smart Card Research and Advanced Applications (CARDIS), 22-27 August 2004, Toulouse, France*, volume 153 of *IFIP*, pages 159–176. Kluwer/Springer, 2004.

[HJM+20] Martin Hell, Thomas Johansson, Willi Meier, Jonathan S¨onnerup, and Hirotaka Yoshida. Grain-128aead, a lightweight aead stream cipher, 2020.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 463–481. Springer, Heidelberg, August 2003.

[KCM20] Tim Beyne KU, Yu Long Chen, and Christoph Dobraunig Bart Mennink. Elephant v2, 2020.

[Kwa00] Matthew Kwan. Reducing the gate count of bitslice DES. *IACR Cryptol. ePrint Arch.*, page 51, 2000.

[LR13] Tancrède Lepoint and Matthieu Rivain. Another nail in the coffin of white-box AES implementations. *IACR Cryptol. ePrint Arch.*, page 455, 2013.

[MAN+19] Lauren De Meyer, Victor Arribas, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. M&m: Masks and macs against physical attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(1):25–50, 2019.

[MDLM18] Darius Mercadier, Pierre-Évariste Dagand, Lionel Lacassagne, and Gilles Muller. Usuba: Optimizing & trustworthy bitslicing compiler. In Jan Eitzinger and James C. Brodman, editors, *Proceedings of the 4th Workshop on Programming Models for SIMD/Vector Processing, WPMVP@PPoPP 2018, Vienna, Austria, February 24, 2018*, pages 4:1–4:8. ACM, 2018.

[Mer20] Darius Mercadier. *Usuba, Optimizing Bitslicing Compiler. (Usuba, Compilateur Bitslicing Optimisant)*. PhD thesis, Sorbonne University, France, 2020.

[MPC00] Lauren May, Lyta Penna, and Andrew J. Clark. An implementation of bitsliced DES on the Pentium MMX$^{\text{TM}}$ processor. In Ed Dawson, Andrew Clark, and Colin Boyd, editors, *ACISP 00*, volume 1841 of *LNCS*, pages 112–122. Springer, Heidelberg, July 2000.

[MRP13] Yoni De Mulder, Peter Roelse, and Bart Preneel. Revisiting the BGE attack on a white-box AES implementation. *IACR Cryptol. ePrint Arch.*, page 450, 2013.

[MZLZ21] Jingdian Ming, Yongbin Zhou, Huizhong Li, and Qian Zhang. A secure and highly efficient first-order masking scheme for AES linear operations. *Cybersecur.*, 4(1):14, 2021.

[NIS] Submission requirements and evaluation criteria for the lightweight cryptography standardization process. Technical report, NIST.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In

Peng Ning, Sihan Qing, and Ninghui Li, editors, *ICICS 06*, volume 4307 of *LNCS*, pages 529–545. Springer, Heidelberg, December 2006.

[NSGD12] Maxime Nassar, Youssef Souissi, Sylvain Guilley, and Jean-Luc Danger. RSM: A small and fast countermeasure for aes, secure against 1st and 2nd-order zero-offset scas. In Wolfgang Rosenstiel and Lothar Thiele, editors, *2012 Design, Automation & Test in Europe Conference & Exhibition, DATE 2012, Dresden, Germany, March 12-16, 2012*, pages 1173–1178. IEEE, 2012.

[Ott05] Martin Otto. *Fault attacks and countermeasures*. PhD thesis, University of Paderborn, Germany, 2005.

[Pet60] W. Wesley Peterson. Encoding and error-correction procedures for the bose-chaudhuri codes. *IRE Trans. Inf. Theory*, 6(4):459–470, 1960.

[PR13] Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 142–159. Springer, Heidelberg, May 2013.

[RBN+15] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating masking schemes. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 764–783. Springer, Heidelberg, August 2015.

[RDB+18] Oscar Reparaz, Lauren De Meyer, Begül Bilgin, Victor Arribas, Svetla Nikova, Ventzislav Nikov, and Nigel P. Smart. CAPA: The spirit of beaver against physical attacks. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 121–151. Springer, Heidelberg, August 2018.

[SBS16] Fabrizio De Santis, Tobias Bauer, and Georg Sigl. Squeezing polynomial masking in tower fields - A higher-order masked AES s-box. In Kerstin Lemke-Rust and Michael Tunstall, editors, *Smart Card Research and Advanced Applications - 15th International Conference, CARDIS 2016, Cannes, France, November 7-9, 2016, Revised Selected Papers*, volume 10146 of *Lecture Notes in Computer Science*, pages 192–208. Springer, 2016.

[Sha49] Claude E. Shannon. Communication theory of secrecy systems. *Bell Syst. Tech. J.*, 28(4):656–715, 1949.

[SKP23] Yaroslav Sovyn, Volodymyr Khoma, and Michal Podpora. Bitsliced implementation of non-algebraic 8×8 cryptographic s-boxes using ×86-64 processor SIMD instructions. *IEEE Trans. Inf. Forensics Secur.*, 18:491–500, 2023.

[SMC+21] Meltem Sonmez, Kerry McKay, Donghoon Chang, Cagdas Calik, Lawrence, Jinkeon Kang, and John Libert. Status report on the second round of the nist lightweight cryptography standardization process. Technical report, NIST, 2021-07-20 04:07:00 2021.

[SSS+20] Dhiman Saha, Yu Sasaki, Danping Shi, Ferdinand Sibleyras, Siwei Sun, and Yingjie Zhang. On the security margin of tinyjambu with refined differential and linear cryptanalysis. *IACR Cryptol. ePrint Arch.*, page 1045, 2020.

[Whia] Whibox 2017. https://whibox.io/contests/2017/.

[Whib] Whibox 2019. https://whibox.io/contests/2019/.

[Whic] Whibox 2021. https://whibox.io/contests/2021/.

[Wys09] Brecht Wyseur. *White-Box Cryptography*. PhD thesis, Katholieke Universiteit Leuven, 2009.

[ZSM+08] Babak Zakeri, Mahmoud Salmasizadeh, Amir Moradi, Mahmoud Tabandeh, and Mohammad T. Manzuri Shalmani. Compact and secure design of masked AES S-box. In Sihan Qing, Hideki Imai, and Guilin Wang, editors, *ICICS 07*, volume 4861 of *LNCS*, pages 216–229. Springer, Heidelberg, December 2008.

COLLEGE DOCTORAL BRETAGNE
MATHS, TELECOMS
INFORMATIQUE, SIGNAL
SYSTEMES, ELECTRONIQUE

Université de Rennes

Titre : Masquage Booléen Résistant aux Attaques par Fautes et White-Boxabilité de Primitives Cryptographiques Légères

Mot clés : Cryptographie White-Box, Masquage Résistant aux Fautes, Cryptographie Légère

Résumé : La cryptographie white-box est dédiée aux implémentations sûres face à un attaquant ayant le contrôle total des dispositifs sur lesquels elles sont déployées. Un des enjeux majeurs auquel elle doit répondre est la résistance aux attaques side-channel. A cette fin, les concepteurs d'implémentations white-box ont pour but d'atténuer au maximum toute dépendance entre les variables de l'implémentation et ses données sensibles. Pour cela, l'une des contre-mesures classiques est l'utilisation de schémas de masquage, néanmoins vulnérables aux attaques par fautes.

La cryptographie white-box doit aussi considérer le compromis coûts-performances de ses implémentations : la question de la « white-boxabilité » des primitives légères, adaptées aux dispositifs aux capacités limitées se pose donc.

Dans cette thèse, nous discutons tout d'abord de la white-boxabilité des finalistes du processus de standardisation de primitives légères du NIST, et présentons une implémentation white-box de GIFT. Dans la seconde partie, nous décrivons notre schéma de masquage de l'opération AND résistant à l'introduction de fautes grâce à l'usage de codes correcteurs BCH et pouvant être implémenté avec uniquement des opérations bit-à-bit.

Title: Boolean Fault-Resistant Masking and White-Boxability of Lightweight Cryptography

Keywords: White-Box Cryptography, Fault-Resistant Masking, Lightweight Cryptography

Abstract: White-box cryptography is dedicated to the implementations of cryptographic primitives that are secure against an attacker being in total control of the devices they are deployed on. One of the main security challenges it needs to address is side-channel security. To that end, designers aim to eliminate the dependence between variables and sensitive data. Classical countermeasures to do so are masking schemes. However, implementations using masking schemes are still vulnerable to fault attacks.

Moreover, the classical cryptographic compromise between security, costs and performances remains in white-box cryptography. Lightweight cryptography is the field of cryptography designed for devices with constrained capacities, therefore the question of the white-boxability of lightweight cryptographic algorithms arises as well.

In the first part of the thesis, we discuss the suitability of the finalists of the NIST Lightweight Cryptography Standardization Contest to white-boxing. We then develop a white-box implementation of GIFT. In the second part of the thesis, we describe a new construction of a bitwise AND masking scheme correcting faults using BCH error-correcting codes and only composed of Boolean operations on bits.