Cryptanalysis of symmetric cipher using generic solvers

Arthur Gontier

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 : *Informatique*

Par
Arthur GONTIER

Utilisation de solveurs génériques pour la cryptanalyse de chiffrements symétriques

Thèse présentée et soutenue à Rennes, le 10 novembre 2023
Unité de recherche : IRISA, INRIA, Université de Rennes 1

Rapporteurs avant soutenance :

Christina BOURA Maîtresse de Conférences, Université de Versailles Saint-Quentin-en-Yvelines
Thomas PEYRIN Professeur, Nanyang Technological University of Singapore

Composition du Jury :

Présidente :	Christine SOLNON	Professeure, INSA Lyon
Examinateurs :	Christina BOURA	Maîtresse de Conférences, Université de Versailles Saint-Quentin-en-Yvelines
	Thomas PEYRIN	Professeur, Nanyang Technological University of Singapore
Dir. de thèse :	Stéphanie DELAUNE	Directrice de recherche CNRS, Rennes
Encadrant :	Patrick DERBEZ	Maître de Conférences, Université de Rennes 1
Encadrant :	Charles PRUD'HOMME	Maître de Conférences, IMT Atlantique, Nantes

Acknowledgement

J'ai choisi de commencer mes remerciements par ma directrice de thèse, Stéphanie, qui s'est aventurée dans la supervision d'une thèse à l'intersection de deux domaines avec succès ainsi que Patrick et Charles, mes encadrants. Le chemin de la recherche est parfois sinueux et c'était un plaisir de l'arpenter avec vous. Votre accompagnement sans failles m'a beaucoup aidé à progresser. Je remercie également Christina et Thomas pour avoir accepté de rapporter ma thèse ainsi que Christine pour la présidence de ce jury.

J'associe à mes remerciements tous les membres des équipes EMSEC, SPICY et CAPSULE pour leur soutien et leur bonne ambiance, les assistantes de ces équipes pour leur aide précieuse ainsi que mes nombreux collègues de bureau. J'ai aussi beaucoup apprécié la collaboration avec Loïc et les membres de l'ANR Decrypt. Enfin, je remercie Laure et Charlotte pour m'avoir initié au monde de la recherche.

Je remercie également mes amis pour leur soutien au quotidien, surtout ceux à qui j'ai infligé les premières relectures de cette thèse.

Pour finir, je remercie chaleureusement ma famille et en particulier mes frères et sœurs et mes parents. En ces temps incertains, la campagne en votre compagnie est un havre de paix délicieux.

Résumé

La cryptographie est omniprésente dans nos sociétés connectées. Cette science englobe toutes les méthodes pour concevoir, analyser et mettre en œuvre des algorithmes dits *cryptographiques* tels que les *chiffrements*, les *signatures* et les *fonctions de hachage*. Les algorithmes cryptographiques permettent, par exemple, la transmission d'une information tout en garantissant sa confidentialité, son intégrité et son authenticité. Un exemple classique d'utilisation de la cryptographie est le protocole HTTPS. Il permet à deux personnes (par exemple Alice et Bob) de communiquer un message alors qu'une personne mal intentionnée peut intercepter le message (Eve). Cependant, la propriété de confidentialité empêche Eve de comprendre le message, l'intégrité permet à Bob de savoir si le message a été modifié et l'authenticité permet à Bob de savoir que c'est bien Alice qui a envoyé le message. La propriété de confidentialité est historiquement assurée par des chiffrements *symétriques*, nécessitant le partage d'une même clef secrète entre Alice et Bob. En 1977, un algorithme de chiffrement *asymétrique* a été inventé pour la première fois. Ce mécanisme repose sur deux clefs distinctes, la clef publique utilisée pour le chiffrement, et la clef privée nécessaire au déchiffrement. Grâce à cet algorithme, tout le monde peut chiffrer un message avec la clef publique mais seule Alice peut les déchiffrer car cela requiert la clef privée. Dans de nombreux protocoles, on effectue un échange de clef reposant sur la cryptographie asymétrique, puis on utilise cette clef fraîchement générée et partagée pour poursuivre les échanges avec la cryptographie symétrique. En effet, même si la cryptographie asymétrique présente de nombreux avantages, elle est plus coûteuse que les chiffrements symétriques qui s'avèrent plus efficaces pour l'échange de grands volumes de données. La conception, l'étude et la mise en œuvre des chiffrements symétriques représentent donc une partie importante de la cryptographie.

La recherche opérationnelle (RO) est originellement définie par "la mise en œuvre de méthodes scientifiques, essentiellement mathématiques, en vue de prendre la meilleure décision possible" pour des problèmes comme l'ordonnancement ou la planification. Le problème le plus connu est probablement le voyageur de commerce où une personne doit visiter un grand nombre de villes en prenant le plus court chemin possible. En RO, les problèmes sont d'abord modélisés, c'est-à-dire qu'ils sont exprimés sous forme de *modèles*

qui peuvent être utilisés par les solveurs. Puis les solveurs résolvent le problème grâce à un ensemble de méthodes de résolution. Les solveurs ont la capacité de fournir la solution optimale, une solution valide, l'ensemble des solutions possibles ou encore de révéler l'absence de solution. Ces dernières années, les progrès réalisés dans le perfectionnement de ces solveurs les ont rendus particulièrement attrayants pour les enjeux complexes de la cryptanalyse. Cependant, la modélisation d'un problème n'est pas toujours immédiate et le choix du solveur peut grandement impacter l'efficacité de la résolution. Pour résoudre efficacement un problème cryptographique avec un solveur, il faut donc des connaissances en cryptographie et en modélisation.

Cryptanalyse des chiffrements symétriques

Les chiffrements symétriques sont des algorithmes qui prennent en entrée un texte et une clef secrète et qui renvoient un texte chiffré. Ce texte chiffré doit être indistinguable d'un texte aléatoire et il doit exister un algorithme capable de déchiffrer le texte chiffré avec cette même clef secrète. En 1949, il a été établi qu'un chiffrement sécurisé devait comporter deux propriétés : la *diffusion* et la *confusion*. La diffusion assure que chaque partie du texte chiffré dépende de tout le texte clair. De cette façon, si une seule lettre du texte clair change, tout le texte chiffré sera impacté. La diffusion est souvent assurée par des opérations comme des permutations, des décalages et d'autres opérations linéaires comme le XOR. La confusion est l'idée que le texte doit être chiffré de manière "complexe", ce qui se traduit par des opérations non-linéaires dont la plus utilisée est la Boite-S. La Boite-S est une table qui mélange une petite partie d'un texte de manière non-linéaire. Les chiffrements symétriques sont souvent des fonctions itérées, c'est-à-dire que l'on va appliquer plusieurs fois une même fonction, appelée *fonction de tour*. Une construction classique est le réseau de substitution-permutation (SPN) où la fonction de tour est composée d'une application de Boites-S suivie d'une permutation. L'exemple le plus connu de SPN est le standard AES mais il en existe beaucoup d'autres. Les chiffrements se distinguent en deux grandes catégories. Les *chiffrements par bloc* (comme les SPN et les réseaux de Feistel) qui découpent le message en plusieurs blocs, et les *chiffrements par flot* (comme Grain et Trivium) qui chiffrent un message au fur et à mesure.

L'analyse des chiffrements symétriques passe principalement par la recherche de *distingueurs*, des propriétés qui permettent de distinguer le chiffrement d'une permutation aléatoire. S'il existe un distingueur sur un chiffrement, alors on peut, en théorie, monter

une attaque et le casser, c'est-à-dire que l'on peut retrouver des messages sans la clef ou déduire une partie de la clef. Faire les attaques n'est pas toujours facile car cela dépend beaucoup de l'efficacité du distingueur, mais en pratique, un chiffrement sur lequel on trouve un distingueur est déprécié. Beaucoup de distingueurs de différents types ont été découverts dans l'histoire de la cryptanalyse (par exemple les distingueurs intégraux qui exploitent la forme algébrique des chiffrements ou les distingueurs différentiels qui étudient la propagation d'une différence à travers le chiffrement, ...). Pour être déclaré robuste, un chiffrement doit être analysé par rapport à chacun des distingueur connus.

Recherche opérationnelle

Les méthodes de recherche opérationnelle se divisent en plusieurs paradigmes. Les plus connus sont les suivants:

Mixed Integer Linear Programming (MILP). La programmation linéaire utilise une modélisation exclusivement sous forme d'équations linéaires. Elle utilise des variables réelles ou entières, ou parfois même les deux. Pour résoudre les problèmes, elle utilise des méthodes comme l'algorithme du simplex ou l'algorithme des points intérieurs pour optimiser un objectif lui aussi linéaire. Si on veut modéliser un problème non-linéaire en MILP, il existe des méthodes de linéarisation mais elles se font au prix de l'introduction de nouvelles variables et contraintes, ce qui peut affecter les performances du solveur. Dans cette thèse, nous utiliserons le solveur Gurobi pour résoudre nos modèles MILP.

Boolean Satisfiability (SAT). Le problème de satisfiabilité booléenne ne porte que sur des variables booléennes. Les modèles SAT utilisent des clauses booléennes sous la forme normale conjonctive, une conjonction de disjonctions. Pour résoudre les problèmes, les solveurs SAT récents utilisent un algorithme appelé le *conflict driven clause learning* (CDCL) pour déterminer s'il existe une assignation des variables qui rend la formule vraie. Nous utiliserons les solveurs picatSAT [ZK16], Z3 [dMB08], Glucose [AS09] et Cryptominisat [Soo16].

Constraint Programming (CP). La programmation par contraintes est plus expressive que les deux précédents paradigmes car elle repose sur une collection de différents types de contraintes qui définissent chacune une relation. Par exemple, la plus connue,

AllDifferent, assure que chaque variable a une valeur différente. Chaque type de contrainte possède un algorithme qui lui permet de faire en sorte que la relation qu'elle contraint soit respectée. Pour résoudre les problèmes, les solveurs de programmation par contraintes utilisent un algorithme de recherche en profondeur ainsi que les algorithmes de chaque contrainte pour trouver une ou plusieurs solutions s'il y en a. Il existe un grand nombre de contraintes (plusieurs centaines dans le catalogue des contraintes [BCDP07]) et il est possible d'en faire de nouvelles. Nous utiliserons principalement les solveurs Choco [PF22], Chuffed [Stu10] et Or-tools.

Contributions

Cette thèse porte sur l'amélioration des techniques de cryptanalyse et des solveurs. En particulier, nous nous intéressons à la recherche de distingueurs sur les chiffrements symétriques avec des solveurs de programmation par contraintes.

Distingueur dans Trivium. Trivium est un chiffrement avec une fonction de tour très simple mais qui comprend un grand nombre de tours (il faut 1152 tours pour initialiser Trivium). À chaque tour après l'initialisation, Trivium produit un nouveau booléen utilisé pour chiffrer un bit du message. Ce booléen est le résultat de l'état initial du chiffrement puis de l'application d'au moins 1152 fois la fonction de tour. Si on développait toutes les fonctions de tour depuis l'état initial, on obtiendrait un grand polynôme qui décrit exactement le booléen de sortie. Connaître la présence ou l'absence de certains monômes dans ce polynôme peut servir de distingueur et permet de monter des attaques dites intégrales [DS09]. Cependant, ce polynôme est beaucoup trop grand pour être retrouvé en temps raisonnable au-delà de 300 tours. On va donc rechercher une sous-partie du polynôme appelée le *superpoly*. Les méthodes existantes pour trouver le superpoly utilisent des modèles MILP à base de booléens et une stratégie diviser-pour-régner [HSWW20, HLM$^+$20]. Une des difficultés du problème intervient quand deux monômes identiques sont dans le superpoly. Dans ce cas ils s'annuleront car l'addition des booléens est le XOR. Il y a un grand nombre d'annulations de ce genre dans le superpoly des grandes instances de Trivium. Le problème est donc de trouver tous les monômes du superpoly tout en faisant attention à la parité des occurrences de chaque monôme.

Comme première contribution, nous proposons une modélisation sous forme de graphe de l'application des fonctions de tour de Trivium. Cette modélisation permet de dé-

tecter des sous-graphes qui donneront un nombre pair de fois le même monôme et de restreindre ces cas dans les modèles. Le modèle graphe permet aussi d'utiliser des formules d'approximation de la taille des monômes pour trouver ceux qui nous intéressent rapidement.

Diffusion dans les réseaux de Feistel généralisés. Le réseau de Feistel est une des premières structures de chiffrement standardisé. Un chiffrement de type Feistel sépare le texte en deux blocs, applique une fonction de tour, ajoute un des blocs avec l'autre puis inverse leur "positions" et recommence. La version généralisée des réseaux de Feistel (GFN) est une application en parallèle de plusieurs paires de Feistel [Nyb96]. De plus, une permutation mélange tous les blocs à chaque tour. Cette permutation doit apporter la diffusion dans le chiffrement, c'est-à-dire le fait que tous les blocs d'entrée impactent chaque bloc de sortie. Pour choisir la permutation qui diffuse le plus, toutes les permutations étaient énumérées et leur diffusion était testée [SM10]. Malheureusement, l'énumération exhaustive n'est plus possible pour des permutations de grande taille. Pendant l'énumération des petites permutations, il a été observé que l'ensemble des meilleures permutations contenait toujours une permutation avec une propriété particulière appelée *even-odd* (chaque bloc d'indice pair était envoyé sur un bloc d'indice impair et inversement). Des travaux plus récents ont permis l'énumération de permutations de plus grande taille (jusqu'à 36 blocs) avec cette propriété particulière [CGT19, DFLM19]. Cependant, il n'y a aucun résultat permettant de savoir si ces permutations sont les meilleures pour la diffusion.

Comme deuxième contribution, nous proposons une modélisation sous forme de graphe des permutations des GFN. Nous présentons un algorithme efficace qui exploite la structure du graphe pour chercher des permutations sans la propriété even-odd et concluons qu'elles ne peuvent pas diffuser strictement mieux que celles ayant la propriété even-odd, et ce, pour les permutations jusqu'à 32 blocs. Nous proposons aussi des variations de cet algorithme pour optimiser d'autres critères que la diffusion.

Génération de modèles. Les distingueurs différentiels sont connus depuis les années 1990 [BS93, BS90]. Ils consistent à trouver une différence entre deux messages qui permettent de retrouver avec une bonne probabilité une différence entre les deux textes chiffrés correspondants. Un composant important des chiffrements pour éviter les distingueurs différentiels est la Boite-S. La modélisation de recherche d'une différentielle se fait souvent en deux parties [BN10, FJP13, GLMS20]. La première partie modélise une

version tronquée du problème où les Boites-S sont approximées. La deuxième partie essaie d'instancier les solutions tronquées ce qui nécessite de modéliser les Boites-S. La programmation par contraintes a montré dans la littérature qu'elle est une solution adaptée pour modéliser la propagation des différences à travers les Boites-S. De plus, il existe déjà des outils génériques pour résoudre la version tronquée du problème [LDLS21].

Comme troisième contribution, nous proposons un outil pour générer des modèles CP pour résoudre l'instanciation de différentielles tronquées. Les modèles sont générés à partir d'une description simple du chiffrement sous forme d'un graphe. Pour modéliser ce problème, nous avons dû ajouter des contraintes dans le solveur Choco et nous avons étudié leur efficacité. Nous proposons aussi une simplification automatique des modèles et une résolution multicœurs.

Amélioration des solveurs CP. La difficulté des problèmes de cryptanalyse pousse les solveurs à leurs limites et ils ne sont pas toujours suffisants pour analyser les chiffrements complets (on étudie souvent des versions réduites des chiffrements en considérant un plus petit nombre de tours). Pour améliorer la recherche de distingueur, proposer des modèles plus astucieux n'est pas toujours suffisant. Une autre piste consiste donc à améliorer les solveurs eux-mêmes. Au début des années 2000, l'apprentissage de conflits (CDCL) a grandement amélioré l'efficacité des solveurs SAT. Pour améliorer les solveurs CP, une adaptation de l'algorithme CDCL a été développée [Stu10, VS10]. Cependant, les algorithmes CP-CDCL ont besoin de pouvoir expliquer le raisonnement de chaque contrainte et déterminer ces explications n'est pas toujours simple.

Comme quatrième contribution, nous proposons un algorithme capable de générer des explications de contraintes. Nous avons défini un ensemble de règles de réécriture qui peuvent déduire les explications d'une contrainte à partir d'un ensemble de contraintes plus petites qui définissent la même relation. Pour valider notre approche, nous avons montré l'efficacité des explications générées sur les différents problèmes des instances publiques du challenge MiniZinc.

Table of Contents

Chapter 1

Introduction

1.1 Context

Cryptography is the art and science of transforming communication into puzzles, guarding the digital realm's most vital treasures with the power of mathematical wizardry. Nowadays, our connected societies use cryptography everywhere. For example, the web protocol for secure connections (HTTPS) can be resumed by two entities (usually named Alice and Bob) who want to share a message with some properties:

— *Confidentiality*, be sure that only Bob can understand the message sent by Alice.
— *Integrity*, be sure that the message was not modified.
— *Authentication*, be sure that the message was sent by Alice.

Confidentiality is generally obtained with a *cipher*, an algorithm that transforms the message, called the *plaintext*, to a random-like message, called the *ciphertext*. This process is called encryption. The ciphertext can then be sent to Bob, who can decrypt the ciphertext back to the plaintext. The encryption algorithm originally uses one *secret key*, and this key is known only by Alice and Bob. Thus, they are the only ones able to encrypt and decrypt the messages. When Alice and Bob share a secret key, the encryption algorithm is called a *symmetric* cipher. A straightforward example of symmetric cipher is the Caesar code. Each letter of a message is replaced by another according to a shift. In this example the shift is the secret key. However, this cipher is weak because it can be brute-forced by trying the 26 possible shifts. Symmetric cipher methods were upgraded over the years alongside the development of machines and computers, but the question of how to share the secret key is not trivial.

In the late 1970s, an asymmetric cipher was invented [DH76, RSA78]. Asymmetric ciphers use two keys, a private key and a public key. To illustrate this, we can think of the public key as a box with a padlock and the secret key as the padlock's key. Alice gives the box with the open padlock to Bob and keeps the key. Bob puts his secret in the box and locks the padlock before sending it back to Alice. Alice can then open the

box and recover the secret. In RSA [RSA78], the secret key is two prime numbers, and the public key is the product of these two primes. The security of this cipher then relies on the hardness of recovering the two primes from their product. Asymmetric ciphers are very slow compared to symmetric ciphers. Therefore, they are mainly used to exchange a secret key. Alice and Bob then know the same secret key and can use a symmetric cipher to communicate. In 1994, a quantum algorithm was discovered to find prime factors, and it was much more efficient than classical algorithms *i.e.,* it has a theoretical complexity polynomial in the size of prime factors [Sho94]. Consequently, the National Institute of Standards and Technology (NIST) started a competition to standardize a new cipher relying on a different problem than factorization as a replacement. Since it should resist the quantum algorithm, these new ciphers will be called post-quantum algorithms, even if we do not know when a strong enough quantum computer will be built to break RSA. In the end, these new algorithms may replace RSA but the use of symmetric cipher is still mandatory.

Our starting example of Alice and Bob with the basic security properties requires a simple protocol. However, more complex configurations and security properties can be imagined. In our societies, we may need to have end-to-end auditable voting systems [RBH+09], blind signatures [Cha83], secure multiparty computation [CDN15], ... For these protocols, we need asymmetric and symmetric ciphers, and we need them to be secure.

Cryptography has adopted binary encoding with modern machines: plaintext, ciphertext, and keys are all bitstrings in $\mathbb{F}_2^*$. Therefore, security is often expressed as a comparison to the key length in the number of bits. A cipher is considered "broken" when an algorithm can find the secret key without trying all the possible keys. Of course, the key length is also significant to say that a cipher is safe. A non-broken cipher with a key whose length is too small can be brute-forced.

1.2 Symmetric encryption

The work of this thesis will focus on symmetric ciphers.

1.2.1 Some historical aspects of symmetric cryptography

The first cryptographic technique in human history is probably long forgotten but we have some records of substitution ciphers, like the Caesar code, in Egypt, Mesopotamia,

Greece, India, Roman empire, Arabic world, ... The invention of new means of communications, computers, and then World Wide Web gave rise to new needs of cryptography. The first applications were military communications. In 1883, it was stated in a French military journal that a good cipher should be secure but not secret [Ker83]. If the secret key is the only secret, then the enemy can discover the cipher without consequences. Note that ciphers were simple electromagnetic machines at this time. The most famous one is Enigma. Invented in 1918 in Germany, it was widely used by the Axis military forces during World War Two to secure radio communication. The number of possible keys was around 2^{67}, so the cipher was considered secure. However, in 1938, the Poland mathematician M.A.Rejewski and his colleagues made a machine that could decrypt Enigma ciphertexts [Rej81]. It exploited some redundancy in the plaintexts to guess the key efficiently. The decryption machine was shared with the Allied forces and was upgraded by A.Turing and his team. It is estimated that this cryptanalysis had a considerable impact on the outcome of the war. In 1973, the American National Bureau of Standards (later renamed NIST) called for candidates ciphers for a Data Encryption Standard (DES) [S$^+$99], a symmetric cipher for non-military uses. The winner was a cipher made by H. Feistel and his colleagues at IBM [Smi71] with the participation of the NSA. This cipher was replaced in another competition in 1997-2001 by the Advanced Encryption Standard (AES) [Ano97]. DES was replaced because the size of the key was too small (56 bits), whereas AES keys have 128 to 256 bits. Cipher competition for standards is an excellent way to stimulate research around new ciphers and allows everyone to have a better trust in their robustness. The two latest competitions are the post-quantum cryptography standardization (2016-2022) for asymmetric cryptography (a second round is planed), and the lightweight cryptography standardization process (2018-2023) for symmetric ciphers. Each competition is composed of several rounds where all the candidate ciphers are studied and attacked by the cryptographic community.

In symmetric cryptography, there is one cipher that cannot be cracked. It is called the One-Time Pad. The first version was invented for telegram bank communications in 1883 [Mil82]. However, it requires a key at least as long as the plaintext. The cipher is simple and relies on only one operator, usually the XOR. Each bit of the plaintext is XORed with a bit of the key. To decrypt the message, each bit of the ciphertext is XORed with the same key. With this method, if the key is "random" and used only once, the cipher provides absolute security. However, we cannot afford long, "random", and single-use keys in modern cryptography, so we need to find an alternative.

1.2.2 Stream ciphers

Stream ciphers are deterministic algorithms that can generate pseudorandom sequences of bits from a key. The output of a stream cipher is an infinite sequence of bits that can be used like the key of a One-Time Pad cipher. Since pseudorandom generators are deterministic, this sequence will loop in the end, so stream ciphers must be reset with a new key from time to time.

A widely used structure to build stream ciphers is *feedback shift registers* (FSR) [Mas69, Rue84]. They are composed of an internal state called a register of bits, and a clock. At each clock tick, a new bit is computed with a feedback function that depends on the bits of the register. This new bit is put at the beginning of the register, and each bit is shifted. The last bit of the register goes out and becomes the next bit of the sequence. The register bits are usually initialized with the secret key. The feedback function is often linear (LFSR). For example, in Figure 1.1, S_i is a bit of the register and c_i is a coefficient of the linear feedback function. The feedback function can also be non-linear (NLFSR) if at least one of the XOR of Figure 1.1 is replaced by an AND.

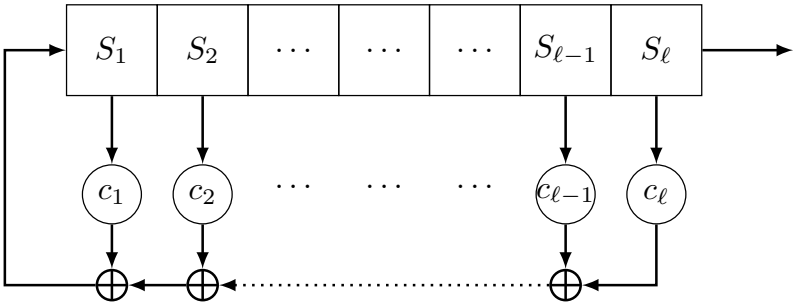


Figure 1.1 – Feedback shift register

A stream cipher can use one FSR or combine multiple FSR to compute a single output sequence bit (see Figure 1.2).

Early standards were mostly stream ciphers (A5/1 (1987), RC4 (1987), E0 (1999)), but they did not age well. Most of them were broken over the years, and some even had backdoors [BSW00, BBFL22].

Shift registers-based stream ciphers were widely used in early cryptography because they can be implemented efficiently. Indeed, shift registers can be directly implemented in a simple electronic circuit called a *flip-flop cascade*.

Block ciphers may have replaced stream ciphers in the new standards because we have a better understanding of block cipher security.

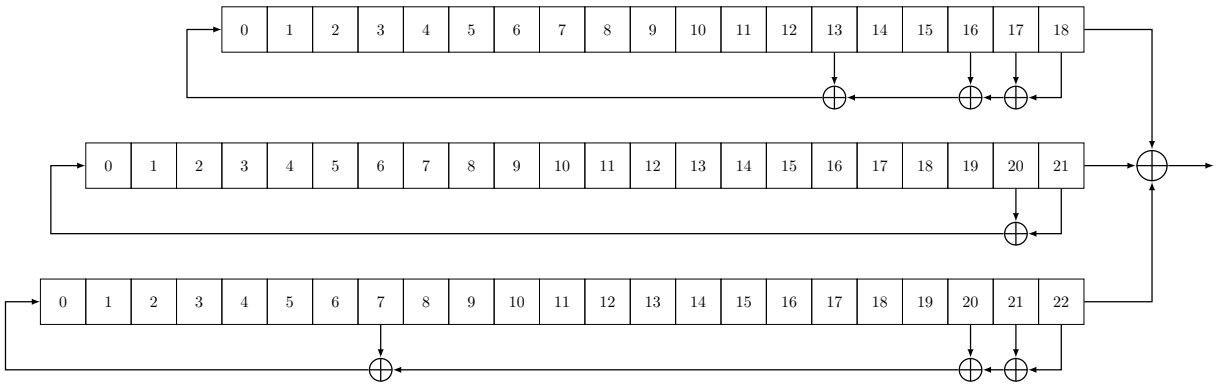


Figure 1.2 – Stream cipher based on three LFSR (A5/1)

In 2004, the eSTREAM competition introduced new resistant stream ciphers, in particular the NLFSR Trivium [CP08] that will be the target of Chapter 2.

1.2.3 Block ciphers

Unlike stream ciphers, block ciphers only operate on fixed length groups of bits called *blocks*. The use of blocks seems more specific than stream ciphers, but a single key can be used to encrypt multiple blocks thanks to algorithms called *modes of operations* [DSD01]. A poorly designed mode paired with an unbreakable cipher can be a weak encryption algorithm (for example the Electronic Codebook Block mode). Some modes can also provide interesting properties like authentication and integrity (for example the Galois Counter Mode).

In [Sha49], Shannon established two essential criteria to design a secure cipher, namely *Diffusion* and *Confusion*.

> **Diffusion** is the fact that each bit of the ciphertext must depend on all the bits of the plaintext. This is usually obtained with permutations and other linear operations like the XOR.
>
> **Confusion** is the idea that each bit must be encrypted in a "complex way". Intuitively, we can say that the cipher needs non-linear operators. For efficiency reasons, the most used non-linear operator is the S-Box, a non-linear function described by a small table (usually 4 or 8 bits). Bit products and finite field arithmetic operations like multiplications and additions can also be used.

The search for good S-Boxes is a complex topic that will not be studied in this thesis.

There are a lot of block ciphers, but they can be roughly put into the two following categories.

Substitution-permutation Network (SPN). SPN are ciphers that alternate two types of layers:

— The substitution layer is usually a parallel application of S-Boxes. Its role is to provide confusion.

— The permutation layer is a series of linear operators to provide diffusion.

AES is a good example of SPN [DR99]. The internal state of AES is a 4×4 matrix of bytes (of 8 bits each). Like most symmetric ciphers, AES is composed of one iterated function called the *round function*. The round function of AES is composed of four layers (see Figure 1.3).

— AddRoundKey (AR) adds the key to the state with a bit-wise XOR.

— SubBytes (SB) is a parallel application of an S-Box on each byte.

— ShiftRows (SR) ensures diffusion in the rows by shifting the bytes.

— MixColumns (MC) ensures diffusion in the columns with a matrix multiplication on each column.

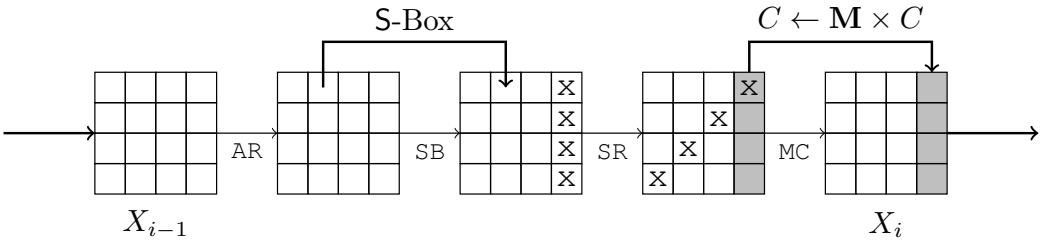


Figure 1.3 – AES round function

Feistel Networks. Feistel network ciphers are the second big category of block ciphers. The main advantage of Feistel networks is that they provide a structure that allows the usage of non-invertible functions. In contrast, each SPN operator must be invertible for the decryption algorithm. There are many Feistel network variations. Chapter 3 of this thesis will study one of them, the Generalized Feistel Network.

DES is an excellent example of a Feistel cipher [Smi71]. The plaintext is split into two blocks of equal length (X_0, X_1). A non-invertible function F is used to compute the blocks

of the next round $(X_1, X_0 \oplus F(X_1, K_r))$ with K_r a key. In this example of Feistel cipher, the F function is an SPN *i.e.,* it is composed of an S-Box layer and a permutation layer.

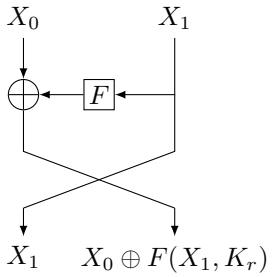


Figure 1.4 – DES round function

SPN and Feistel networks use round functions that may need a key at each round (K_r). For example the AddRoundKey operator in AES and the F function of DES. The round keys are generated from the secret key with a *key schedule* algorithm. If a key schedule is too simple, it may introduce a bias in the cipher so this component must be studied as well [KM04].

1.2.4 Lightweight encryption

The latest symmetric competition organized by the NIST (2018-2023) was about lightweight ciphers. The idea was to select symmetric ciphers more suited for constrained environments. A lightweight cipher can be designed for a specific environment or simply for global efficiency [BP17]. Since there are different use cases, these ciphers may not have the same efficiency goals. They are generally designed to have some of the following properties: fast throughput, low number of logical gates, or small power consumption. Moreover, lightweight ciphers have to be secure like all the other ciphers. The design phase of lightweight ciphers may also be analysed for attacks on the physical chip because the hardware implementation is central.

There is a wide variety of lightweight block and stream ciphers and most of them are referenced and compared in [BP15]. Block and stream ciphers both were proposed to the NIST Lightweight Cryptography competition and the two ideas can be combined in several ways since stream-inspired block ciphers [CDK09] have been proposed and conversely [EJMY19].

1.3 Attacks

To convince that a new symmetric cipher is secure, it must resist all known attacks. Therefore, the definition of a cipher is always followed by a security analysis to evaluate the resistance of the cipher against each known attack.

Brute-force. The brute-force attack is when the adversary tries all the keys to find the correct one. However, the complexity of the brute-force attack is 2^k (with k the number of bits of the key) so it is impracticable for ciphers with a reasonable number of security bits like 128 bits. In cryptanalysis, a cipher is considered "broken" if an algorithm can find the key faster than the brute-force attack.

1.3.1 Security

The security can be formalized as follows. Let F be a vectorial Boolean function taking as argument the key $K \in \{0,1\}^k$ and a message $x \in \{0,1\}^n$ *i.e.,*

$$F : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$$

For each key K, the function must be a permutation because we need its inverse to decrypt the messages. To be secure, this permutation must be a pseudorandom permutation *i.e.,* it must be indistinguishable from random noise. Because we cannot simply create a pseudorandom permutation, we will estimate its security by searching how to distinguish the function from a random noise.

To do so, we will define an adversary and an oracle. The oracle is an entity that will randomly provide either $F(K,x)$ or a random noise in $\{0,1\}^n$ to the adversary for each input x. The goal of the adversary is then to distinguish the cipher from the random noise. If the adversary has an algorithm that can find the right answer with a probability higher than 0.5, the cipher is considered as not secure. In practice, the adversary may not have full access to the cipher. To take this into account, we will use different attacker models such as:

— Ciphertext only: The adversary has only access to the ciphertexts.
— Known plaintext: The adversary knows some plaintext-ciphertext pairs.
— Chosen plaintext: The adversary can choose plaintexts and get their corresponding ciphertexts.

— Chosen ciphertext: The adversary can choose ciphertexts and get their corresponding plaintexts.
— Related key: The adversary can get ciphertexts of the same plaintext with other keys derived from the secret key. The other keys are derived in a known way.

Each model corresponds to a real world situation and some distinguishers may not be compatible with the most restrictive attack models.

The attack. Making the attack is possible if the distinguisher has a high enough probability and the attack model is possible in practice. The attacker searches for the distinguisher with plaintext, ciphertext or related-key attacks depending on the attack model. The attacker can then recover some messages or it can deduce the secret key. Practical attacks are rarely made, and a cipher is deprecated if a distinguisher with a high probability is found on it. We will illustrate in more detail a common distinguisher, the differential distinguisher [BS90].

1.3.2 Differential distinguisher

Differential analysis is a method to analyze the effect of particular differences in plaintext pairs on the differences of the resultant ciphertexts. Differential analysis was first introduced in [BS93, BS90] to study the cipher DES and some DES variants. It was then applied to many other ciphers [MPP09, NPSS10, SY11, BPW15, BGG$^+$23b, MPRS09, MRST09, GP10, Gil14, GR20].

A difference is the result of a substraction between two messages. In cryptanalysis, the bit-wise XOR operation is mainly used, and we will note it $+$ for clarity. For a cipher function F, a is an input difference if x and $x+a$ are both plaintexts or both keys. a would be an output difference if x and $x+a$ were ciphertexts. There is a differential distinguisher against F for a pair of differences (a, b) if the probability that $F(x) = F(x+a)+b$ is high *i.e.,* the input difference a has a good probability of ending up to the output difference b. This behaviour is not present in pseudorandom functions, so we do not want it in any cipher. Since we cannot search for all pair of differences a and b to find a distinguisher because there are too many of them, we need to find more efficient methods.

In differential analysis, we decompose the study of the propagation of the input difference a through all the cipher operators. The paths of the difference is called a differential trail. The linear operators of the cipher like the XOR will always propagate the difference. For example, if we have two variables x and y in a cipher that end up with the differences

a and b, then the difference in the variable $z = x + y$ is $a + b$. However, the confusion part of a cipher is composed of non-linear operators that will propagate a difference with a probability lower than one. For example, we can compute the probability of every input-output pair of differences of an S-Box. If there are more than one S-Box in the trail, their probabilities are multiplied assuming the Markov property of the cipher *i.e.,* the difference probability is independent of the plaintext [LMM91]. The problem of searching a differential trail is usually hard. However, we only need to be sure that the best trails have a low enough probability to say that a cipher is secure.

Truncated trails The truncated differential analysis simplifies the idea of a differential trail [Knu94]. Each difference will be replaced by a single Boolean variable. This variable encodes the existence of the difference without tracking its value. For the probability of propagation through the S-Boxes, we will take the best probability for the attacker. The best truncated trail then gives an upper bound on the probability of differential trails. If this probability is low enough, we can stop the differential analysis there. In cryptographic papers, the truncated trails results are often resumed by the lowest number of S-Boxes in the best trail (we say that there are n *active* S-Boxes). When the truncated trails have a promising probability, we must try to instantiate them with real differences. Because the truncated trails are a simplification of real trails, they may not lead to real differential trails. In general, the search for differential distinguishers is separated into two steps [BN10, FJP13, GLMS20]. The first step is the search for truncated trails. The second step tries to instantiate the truncated trails with real difference values. The second step of this problem will be studied in more details in Chapter 4.

1.3.3 Other distinguishers and their corresponding attacks

There are many distinguishers in cryptanalysis [SPQ03]. We give in this section a short overview of the most famous ones.

Boomerang attack. Boomerang attack is a differential-style attack in which the attacker does not try to cover the whole cipher with a single highly-probable differential pattern. Instead, the attacker tries to find two high-probability differentials that are not necessarily related to each other but together cover the whole cipher. In its basic version, it requires the ability to make chosen-plaintext and chosen-ciphertext queries [Wag99].

Variants of the boomerang attacks are called rectangle attack [BDK01] or sandwich attack [DKS10].

Impossible differential attacks. Impossible differential attacks are an other differential variant [BBS99]. They consist of finding differentials trails with a probability of 0. These trails can be used by the adversary to test less keys.

Higher order differential attacks. Higher order differential attacks extends the difference propagation to multi-difference propagation. The idea is to study the propagation of a sum of differences to a single output difference by exploiting a low algebraic degree of the cipher [Knu94].

Integral attacks. Integral attacks (also called square [DKR97] or saturation or multiset attacks) are chosen plaintext attacks studying the propagation of well-chosen sets of plaintexts through the cipher. It has the particularity that we get information only by considering specific sums of ciphertexts.

Linear attacks. Linear cryptanalysis is a known plaintext attack that exploits a linear relation between the inputs and outputs of a cipher [Mat93]. If a linear relation has a good probability, it can be used as a distinguisher to make an attack. This might happen if the S-Boxes and the other non linear operators of a cipher have a close linear approximation and if they are few.

Interpolation attacks. Interpolation cryptanalysis is a known or chosen plaintext attack applicable to ciphers for which the round function can be written as a reasonably simple algebraic expression [JK97]. It relies on the application of the Lagrange interpolation formula.

Meet-in-the-middle attack. Meet-in-the-middle attack is a known plaintext attack that relies on the idea to utilize both the encryption and decryption algorithms of a cipher and try to find pairs of keys that are compatible [DH77]. This attack has shown great results on 2DES and is the main reason why we use 3DES, a successive application of three DES ciphers.

Slide attacks. Slide attacks exploit the degree of repetitiveness of a block cipher and thus are applicable to iterative block ciphers with a periodic key schedule [BW99]. This attack is unrelated to the number of rounds.

1.4 Tools to find distinguishers

Until recently, the search of distinguishers was performed with dedicated algorithms. However, as we have seen in the previous section, there are a lot of distinguishers and a lot of ciphers. To ensure their robustness, we must search for each distinguisher on each cipher. In parallel, the operation research methods and solvers gained in efficiency and became attractive tools to solve these cryptographic problems. A solver is a toolbox of solving methods that are used to solve a *model*. The advantage of a solver is that you only need to declare the model and the solver will solve it, you do not need to implement the state of the art solving methods. Thus using a solver can save a lot of development time. However, solvers act as black boxes and this can become problematic if we want to control it precisely. Moreover, there are many ways to model the same problem and there is often one model that will be solved more quickly by the solver. Some early applications to cryptanalysis led to unsatisfactory results [Mas99, MZ06, Sta14]. Indeed, the model and solver choice is essential. For example, a non-linear problem can be linearized but a linear solver may struggle to solve it. In the solver efficiency comparison on truncated differentials on the cipher Skinny [DDH+21], the solvers have big efficiency differences and the fastest method was a handmade algorithm. These results show the trade-off between solvers that need a carefully chosen model and the dedicated methods that need to be built from scratch.

Operations research. The term "operations research" originates from using automatic analytic methods to estimate losses in military operations. These techniques became very useful with the development of computers. Indeed, they can be applied in a lot of industrial situations. Two prominent examples are transportation and scheduling problems. For example, the travelling salesperson problem (TSP) is a problem where a person or a vehicle needs to find the shortest path to visit all the cities of a country [ABCC11]. A complex example of scheduling problem is finding the optimal schedule for all students and teachers of a university. In practice, each situation is specific, and there may be a lot of variants. Transportation problems can involve multiple vehicles, timing constraints, ca-

pacity constraints, energy constraints, pollution constraints, ... Transportation problems also depend on the terrain, for example, rail, sea, road, air, space, pipeline, cable or more than one of these. Scheduling problems have even more variants because every industrial process can be scheduled and some cases may involve both problems. For example, the maintenance schedule of trains or planes impacts the transportation problem. For the largest problems, finding the optimal solution might take too much time. Therefore, the *exact* methods are sometime replaced by *incomplete* methods. Rather than giving the optimal solution, the incomplete methods find a "good" solution in reasonable time without any proof of how good this solution is. However, in cryptographic problems, we often want security guarantees, so we mainly use exact methods to recover the best solution or all the solutions.

For each problem, deciding the adapted solving technique is an important question. Operations research includes the development and application of a wide variety of problem-solving techniques such as Linear Programming (LP), Integer Linear Programming (ILP), Constraint Programming (CP), Boolean satisfiability (SAT), Satisfiability modulo theory (SMT), ...

We will present these techniques with the following running example:

Example 1.1 (nqueens) *Let a chess board of size $n \times n$. We want to place the most queens possible such that no two queens share the same line, column, or diagonal. See a solution for the 4queen problem in Figure 1.5*

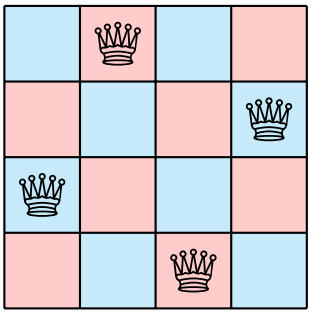


Figure 1.5 – 4queen solution

1.4.1 Linear programming (LP)

A linear program (LP) includes constraints in the form of linear equalities and inequalities and a linear objective function to maximize or minimize. It operates on real variables. The historical algorithm used to solve a linear program is the simplex algorithm. This idea originated in the Fourier–Motzkin elimination methods in 1827, but it was formalized in 1947 by Dantzig [Dan02]. The name simplex refers to the simplest possible polytope in any dimension. In the same years, the interior-point method (or barrier method) was also invented but was not suitable for computers of the time. In 1984, an efficient barrier algorithm was proposed with better performances than the simplex. However, it was not suited for all situations, so both algorithms are available in most solvers nowadays. An essential part of the solving process uses the dual problem. Indeed, linear inequalities form a polytope, and we can invert it to have a dual polytope that gives bounds on the objective or optimallity proofs.

An Integer Linear Program (ILP) is a LP with only integer variables The idea originates in 1958 [Gom02, Gom10]. The solving algorithm is a branch-and-cut algorithm. This algorithm uses a decision tree. A decision is the assignment of a value to a variable. After each decision, a relaxation of the ILP is solved as an LP. The solution is used to find some bounds and new constraints called cuts [Chv73, Raj90] to help the next decision. If the problem becomes infeasible, a backtracking procedure changes a previous decision to create a new branch. The well known Mixed Integer Linear program (MILP) uses both real and integer variables.

Example 1.2 *The nqueen problem can be described with the following ILP. The variables of the problem are Boolean, they represent the presence of a queen on a square (1.1). The objective is to maximize the number of queens (1.2). The first constraint ensures that there is no more than one queen by column (1.3). Same for the lines with constraint (1.4). And same for the two diagonals with the last four constraints (1.5),(1.6),(1.7),(1.8).*

$$x_{i,j} = \begin{cases} 1 & \text{if there is a queen on the square } i,j \\ 0 & \text{otherwise} \end{cases} \tag{1.1}$$

$$\max \sum_{i=1}^{n} \sum_{j=1}^{n} x_{i,j} \tag{1.2}$$

$$\sum_{i=1}^{n} x_{i,c} \leq 1 \quad \forall c \in [\![1, n]\!] \tag{1.3}$$
$$\sum_{j=1}^{n} x_{l,j} \leq 1 \quad \forall l \in [\![1, n]\!] \tag{1.4}$$
$$\sum_{d=0}^{n-c} x_{1+d,c+d} \leq 1 \quad \forall c \in [\![1, n]\!] \tag{1.5}$$
$$\sum_{d=0}^{n-l} x_{l+d,1+d} \leq 1 \quad \forall l \in [\![1, n]\!] \tag{1.6}$$
$$\sum_{d=0}^{c-1} x_{1+d,c-d} \leq 1 \quad \forall c \in [\![1, n]\!] \tag{1.7}$$
$$\sum_{d=0}^{n-l} x_{l+d,n-d} \leq 1 \quad \forall l \in [\![1, n]\!] \tag{1.8}$$

MILP solvers have gained the most interest in the cryptographer community, especially with the performances of the solver Gurobi [SHW+14, BJK+16, ST17, MWGP11, HSWW20, HLM+21]. Gurobi is a commercial solver known for his strong preprocess *i.e.,* a model simplification algorithm [Gur21]. It is used in academic research because it has an academic licence and because it is one of the best MILP solvers, as we can see on some comparison benchmarks [1] on MILP problem libraries like [GHG+21].

1.4.2 Boolean satisfiability (SAT)

A Boolean satifiability problem is a problem composed of Boolean variables and simple logical operators such as OR, AND, and NOT $(\vee, \wedge, \neg)$ [Coo71]. The SAT problem is a Boolean formula in the Conjunctive Normal Form (CNF) *i.e.,* a conjunction of disjunction of variables or their negation. Each disjunction is a constraint called a clause and the goal is to determine if the problem is satisfiable or not *i.e.,* if there is an assignment of variables where no clause is false.

The state of the art SAT solving algorithm is called the Conflict Driven Clause Learning algorithm (CDCL) [SS96]. This algorithm uses a decision tree. After each decision, the clauses are simplified with an algorithm called the unit clause propagation. If the problem becomes infeasible, a backtracking procedure changes a previous decision to create a new branch. The specificity of the CDCL algorithm is that when the problem becomes infeasible, a learning algorithm will search in the decision tree for the responsible previous decisions that explain this failure. This will produce an explanation of the conflict that can be added to the model as a new clause in the CNF. The conflict learning algorithm is very efficient but we must take care of the cases where the learnt clauses would be too many. SAT solvers were used for various cryptanalysis problems like

1. https://plato.asu.edu/ftp/milp.html

[MZ06, Sto16, DKM+17, Laf18, GLST22]. The best SAT solvers are compared in annual competitions [2].

Example 1.3 *The nqueen problem can be described with the following CNF. The variables of the problem are Boolean, they represents the presence of a queen on a square (1.9). The satisfaction problem is to know if we can put n queens on the $n \times n$ board. Therefore we need at least one queen on each column (1.10) and each line (1.11). We also want no more than one queen by column (1.12) and line (1.13) and same for the diagonals with the clauses (1.14),(1.15),(1.16),(1.17).*

$$x_{i,j} = \begin{cases} true & \text{if there is a queen on the square } i,j \\ false & \text{otherwise} \end{cases} \tag{1.9}$$

$$\bigvee_{i=1}^{n} x_{i,c} \quad \forall c \in [\![1,n]\!] \tag{1.10}$$

$$\bigvee_{j=1}^{n} x_{l,j} \quad \forall l \in [\![1,n]\!] \tag{1.11}$$

$$\neg x_{i,c} \vee \neg x_{i',c} \quad \forall i \forall i' \forall c \in [\![1,n]\!], i' \neq i \tag{1.12}$$

$$\neg x_{l,j} \vee \neg x_{l,j'} \quad \forall j \forall j' \forall l \in [\![1,n]\!], j' \neq j \tag{1.13}$$

$$\neg x_{1+d,c+d} \vee \neg x_{1+d',c+d'} \quad \forall d \forall d' \in [\![0,n-c]\!], d' \neq d, \forall c \in [\![1,n]\!] \tag{1.14}$$

$$\neg x_{l+d,1+d} \vee \neg x_{l+d',1+d'} \quad \forall d \forall d' \in [\![0,n-l]\!], d' \neq d, \forall l \in [\![1,n]\!] \tag{1.15}$$

$$\neg x_{1+d,c-d} \vee \neg x_{1+d',c-d'} \quad \forall d \forall d' \in [\![0,c-1]\!], d' \neq d, \forall c \in [\![1,n]\!] \tag{1.16}$$

$$\neg x_{l+d,n-d} \vee \neg x_{l+d',n-d'} \quad \forall d \forall d' \in [\![0,n-l]\!], d' \neq d, \forall l \in [\![1,n]\!] \tag{1.17}$$

1.4.3 Constraint programming (CP)

Constraint programming (CP) is a much more flexible paradigm than the others. It was invented in the mid 70s ([Mon74, Lau78, Ros88]) to solve combinatorial problems.

The main definitions and usual notations can be found in the Handbook of Constraint Programming [RvBW06]. A constraint program is composed of two things:

— Variables are defined with a domain. The domains are usually finite (a set of integers) or sometime infinite (real bounds) but some variables can also be sets or graphs as long as you can define their domains. For example, a graph variable would have a graph domain with a lower bound graph containing the mandatory

2. www.satcompetition.org

nodes and edges and an upper bound graph containing the possible nodes and edges.

— Constraints are relations over variables. Unlike LP and SAT that only support linear equations and CNF clauses, the CP constraints can be of any type as long as we can define their relation (also called contract) and that this relation can be used in an algorithm called filtering algorithm. The filtering algorithm removes the impossible values of the variable domains according to the constraint.

To solve a CP model, the solver uses a backtracking algorithm paired with the filtering algorithms of each constraint in the model. The solving process starts with all the possible values in each variable domain. According to a strategy, the solver takes a decision (for example, a variable is assigned to a value). After this decision, the filtering algorithms are called to remove all the inconsistent values in the other variables domains. This process is called constraint propagation. When no more values can be filtered, the solver makes another decision and repeats the process. If a filtering algorithm empties a domain, we have a conflict and must backtrack to a previous decision. The solver stops when it has found one or more solutions or if all the decisions lead to conflicts *i.e.,* the problem has no solution. The fact that constraints have independent filtering algorithms makes CP extremely expressive and expandable.

Example 1.4 *The nqueen problem can be described with the following CP constraints. The variables of the problem are integers. Each variable represents a line and each integer value represents the column of the queen. For example,* $x_2 = 3$ *is a queen at line 2 and column 3 (1.18). To ensure that all queens are on different columns we use the constraint* AllDifferent *(1.19). We use the same constraint on the diagonals (1.20),(1.21).*

$$x_i \in [\![1, n]\!] \quad \forall i \in [\![1, n]\!] \tag{1.18}$$

$$\textsc{AllDifferent}(\{x_i,\ \forall i \in [\![1, n]\!]\}) \tag{1.19}$$

$$\textsc{AllDifferent}(\{x_i + i,\ \forall i \in [\![1, n]\!]\}) \tag{1.20}$$

$$\textsc{AllDifferent}(\{x_i - i,\ \forall i \in [\![1, n]\!]\}) \tag{1.21}$$

To compare CP solvers, there is the MiniZinc [3] and xcsp [4] competitions. Some of the

3. www.MiniZinc.org/challenge
4. www.xcsp.org/competitions

best ones are Or-tools [5], chuffed [6] and Choco [PF22]. Although Constraint programming is less used, it has been successfully applied to differential cryptanalysis [GMS16, ENP19, DDH+21]. Sometimes, the cryptographic problem can be split into multiple smaller problems. For example, the differential analysis can be divided into two problems, enumerate the optimal truncated trails and instantiate the trails with differences. It turns out that CP solvers are very efficient in the second step. The differential analysis described in [DDH+21] is then a combination of methods, an ad hoc method for the first step and a CP solver for the second one. However, there are many ciphers and many distinguishers, and developing a model for each solver for every distinguisher to find the most appropriate one is time-consuming. Hopefully, there are modelling languages that can communicate with multiple solvers.

1.4.4 MiniZinc

One of the best ways to find the appropriate solver for a problem is to use the MiniZinc modelling language. With MiniZinc, we can specify the problem variables and constraints using a simple high-level syntax. Indeed, MiniZinc is not a solver itself, but a MiniZinc model can be translated into the FlatZinc format. This FlatZinc representation is more machine-readable and is designed to close the gap between the model and the solvers. Many solvers can understand and solve problems in the FlatZinc format, for example, Gurobi, PicatSAT, OR-tools, Choco, ... For each solver we choose to connect to MiniZinc, the FlatZinc model will automatically rewrite the MiniZinc constraints into constraints in the solver's scope. For example, an integer product constraint in MiniZinc will be decomposed into a set of linear equations for the linear solvers or a CNF for SAT solvers. The translation to the solver constraints may not be the most efficient one but thanks to this we can test very different solvers with only one model. We used it for most of our early analyses.

1.5 Contributions

There are many distinguishers and many ciphers. Since all of them need to be resistant to all the known attacks, much development is needed. However, the development of

5. github.com/google/or-tools
6. https://github.com/chuffed/chuffed

operations research methods and solvers is making them viable solutions to solve some complex problems of cryptography.

This thesis began with the observation that constraint programming was an excellent paradigm for solving cryptographic problems like differential analysis. The most used methods in the cryptographic community are MILP and SAT. Therefore, the project was to work on CP models for other cryptographic problems and CP solving algorithm enhancement. Although our target was CP modelling, we did not restrain our models to CP and we also have some competitive MILP models and hand-made algorithms.

The first problem we studied is the search for the main component of the cube attack, the superpoly, with CP techniques. We found that a graph representation of the stream cipher Trivium made the modelling easier for both MILP and CP. Moreover, this representation allowed us to highlight new constraints on this problem.

The second problem we studied is the diffusion in the Generalized Feistel Networks, especially the search for the best diffusing permutation. The CP models we developed did also tend to a graph representation. With this new representation, we could find new properties and propose an efficient algorithm to find the optimal permutations.

The third problem we studied was differential analysis. In particular, we were interested in the development of a generic tool to generate CP models to solve the second step of the differential analysis. This was made in collaboration with the tool Tagada that already implemented the first step. The goal of the tool is to offer a simple DAG representation of ciphers and the automatic search for differential characteristics on it.

Finally we were interested in the improvement of the CP solving methods. Conflict learning in CP solvers is not widely used because all the constraint types need an explanation. In this work, we propose an algorithm to generate these explanations. Furthermore, we demonstrate that these explanations are competitive with the state-of-the-art conflict learning solvers on public instances.

1.5.1 Thesis outline

Chapter 2 is dedicated to the cryptographic problem of superpoly recovery on the stream cipher Trivium. The superpoly recovery is the main component of an integral attack variant called the cube attack. After some prototype models, we published a graph-based MILP model to recover the superpolys more efficiently [DDGP21]. Our new graph representation of this problem gives a helpful structure to add more constraints to the model.

Chapter 3 concerns the diffusion study in the Generalized Feistel Networks. The diffusion study is essential to make new secure and efficient ciphers. After some prototype models, we published a path-based ad hoc algorithm to solve this problem [DDGP22]. We also used a graph representation to define new properties to help the algorithm.

Chapter 4 contributes to the Tagada tool, especially the second part of the differential analysis handled with CP solvers. From a generic graph representation of any cipher, we generate and solve CP models of differential analysis with the Choco solver. We compared the generic approach to the previous works with good results. Genericity and efficiency are often contradictory, and we tried to implement all the known methods to conceal both. This work has been accepted at Indocrypt 2023.

Chapter 5 focuses on improving the solving algorithms of CP solvers. We were interested in the conflict analysis in CP solvers, particularly the explanation formulas essential to this algorithm. We proposed an explanation generator based on a set of constraints. Our method relies on a rewriting algorithm to deduce new explanations.

Chapter 2

Superpoly recovery on Trivium

Introduction

Trivium is a stream cipher with a rather simple description and yet a strong resistance to cryptanalysis [CP08]. Indeed, one round of Trivium is composed of three equations plus some bit shifts. These equations have four terms, and the non-linear operation is a product of only two bits. Therefore algebraic attacks may be very suited. In this chapter, we first present the cipher Trivium and recall the previous methods used to study the algebraic structure of stream ciphers. These methods rely on MILP models [HSWW20, HLM+20] to recover information on the cipher, namely the *superpoly*. We first tried to transcribe these methods in CP but we did not get interesting results. Then, we introduce a new graph representation to recover the superpoly of Trivium. This new representation allows us to find new *forbidden patterns* and strengthen the model with new constraints. This representation also allows us to use previous works on the degree approximation to implement an efficient strategy. In the end, our graph model can recover the superpoly of reduced round Trivium at least ten times faster than previous models and opens the question of exploiting new patterns.

Takeaway

This contribution supports the following conclusions:

- Trivium can be represented as a simple automaton. From the automaton, we can generate Directed Acyclic Graphs (DAGs) representing the Algebraic Normal Form (ANF) of Trivium.
- DAGs highlight useful doubling patterns that can be used to constrain the search of specific monomials in the ANF of Trivium, but these patterns might conflict, so they must be used carefully.
- The graph model can exploit arity approximation as a search strategy to be faster than previous divide-and-conquer strategies.

2.1 Background

This chapter will focus on the analysis of the stream cipher Trivium. We will search for algebraic properties of this cipher that can be used in cube attacks.

2.1.1 Trivium

Trivium is the Latin word for three-way crossroad and the semantic root of the word trivial. In symmetric cryptography, it was introduced to renew the stream ciphers. In the late 90s, stream ciphers were slowly replaced by more recent block ciphers for communication standards. For example, the GSM standard A5/1 (1987) was replaced by the block cipher A5/3 (1999) [BSW00] and the stream cipher RC4 (1987) of the wifi standard IEEE 802.11 was replaced by AES (1997) [MS01, Mir02, SP03]. This decreasing usage might be explained by a better understanding of block ciphers security and the improvement of block ciphers efficiency. Indeed, the main reason to choose a stream cipher over a block cipher was the efficiency gain in a limited hardware setup [Bir04]. In 2004, the question "Are stream ciphers dead or alive ?" led to the eSTREAM project to promote new stream ciphers like Trivium [CP08] and Grain [HJM07]. In this chapter, we will focus on Trivium, a stream cipher with a simple description designed for hardware-oriented

performances.

TRIVIUM is a Non-linear Feedback Shift Register cipher (NFSR). As its name suggests, it is composed of a shift register and a non-linear feedback function. A shift register is an array of bits paired with a notion of clock. For ciphers, the clock is simply the successive rounds. At each round, the register is shifted, meaning that each bit is stored in the next index of the register. The last bit of the register is the output, and the first bit is the input. When we use a feedback function to generate the input bit from some well-chosen bits of the register, the sequence of output bits can be used as a pseudo-random generator.

The internal state of TRIVIUM is represented by a 288-bit state $(s_1, s_2, \ldots, s_{288})$ distributed on three registers A, B, and C. The first state is initialized with the secret key K and a set of public variables called Initialization Vector (IV). This vector is mandatory to be able to encrypt two plaintexts to two different ciphertexts.

To initialize TRIVIUM, the 80-bit secret key K is loaded to register A, and the 80-bit IV is loaded to register B. The other state bits are set to 0 except the last three bits in register C. Namely, the initial state bits are represented as:

$$\begin{aligned} s_1, \ldots, s_{80}, s_{81}, \ldots, s_{93} &\leftarrow K[1], \ldots, K[80], 0, \ldots, 0 \\ s_{94}, \ldots, s_{174}, s_{175}, s_{176}, s_{177} &\leftarrow IV[1], \ldots, IV[80], 0, 0, 0, 0 \\ s_{178}, \ldots, s_{285}, s_{286}, s_{287}, s_{288} &\leftarrow 0, \ldots, 0, 1, 1, 1 \end{aligned}$$

At each round, we first compute the results of the feedback functions t_1, t_2, and t_3 as:

$$\begin{aligned} t_1 &\leftarrow s_{66} + s_{91}s_{92} + s_{93} + s_{171} \\ t_2 &\leftarrow s_{162} + s_{175}s_{176} + s_{177} + s_{264} \\ t_3 &\leftarrow s_{243} + s_{286}s_{287} + s_{288} + s_{69} \end{aligned}$$

Then, the three registers are shifted, and the first bit of each register is updated as follows:

$$\mathsf{A} \leftarrow t_3, s_1, \ldots, s_{92} \quad \mathsf{B} \leftarrow t_1, s_{94}, \ldots, s_{176} \quad \mathsf{C} \leftarrow t_2, s_{178}, \ldots, s_{287}$$

The state is updated 1152 times, and then, at each new round, an output bit z is produced: $z \leftarrow s_{66} + s_{93} + s_{162} + s_{177} + s_{243} + s_{288}$. Figure 2.1 depicts graphically transitions of the TRIVIUM stream cipher.

The output bits are then XORed with the plaintext bits to produce the ciphertext. One key of TRIVIUM can produce up to 2^{64} output bits until it needs to be reset with a new key.

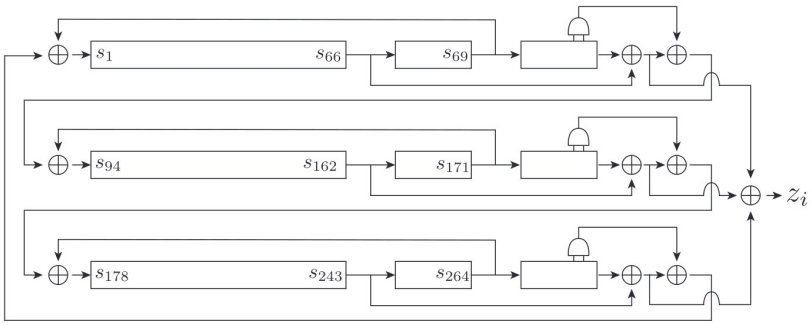


Figure 2.1 – The Trivium cipher.

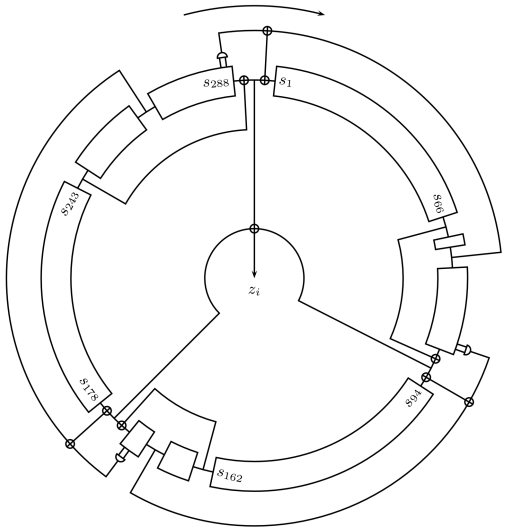


Figure 2.2 – The Trivium cipher in circle shape.

In cryptography, the feedback function was mainly linear, like in A5/1 for GSM communications (composed of 3 linear feedback shift registers) or E0 (1999) for Bluetooth (composed of 4 LFSRs). However, LFSRs were shown amenable to fast correlation attacks [MS89, ÅLHJ12] and the algebraic attacks [CM03, Arm06]. To explain them briefly, in a fast correlation attack, we try to find correlations between the keystream and the output bits. In algebraic attacks, we first look for a set of equations in the secret bits of the key and the output bits. Then, we solve this equation system to recover the key using methods like Gröbner bases. Unlike these LFSRs, Trivium uses a non-linear feedback function with only one bit-product. Despite its simplicity, Trivium showed remarkable resistance to cryptanalysis. When designing a symmetric cipher, one has to ensure that it cannot be broken by any known attack. In the security analysis section, the authors of

Trivium themselves stated that Trivium might be "a particularly attractive target for algebraic attacks" because of the small size and low degree of the equations. The linearization techniques might not be easy to apply, but "other techniques might be applicable and [...] need to be investigated". The monomial prediction is a method to find information on the polynomial of an iterated function [Tod15, HSWW20]. Since its discovery, the monomial prediction has been studied on round-reduced Trivium to find polynomials that can be used to find distinguishers for algebraic attacks like cube attacks. The community optimized this technique to use it on round-reduced Trivium with more and more rounds. The current maximum number of rounds is 848 rounds [HHPW22]. Trivium has yet to be threatened by this attack because it has 1152 initialization rounds.

2.1.2 Cube attacks

The cube attack was first introduced in [DS09] and successfully applied against various stream ciphers, e.g. [ADMS09, DS09, FV13]. It is a descendant of high order differential attacks [Lai94] and algebraic input vector differential attack [Vie07].

This attack relies on the fact that stream ciphers are Boolean functions and that Boolean functions can be described by a unique polynomial, their ANF. More precisely, a stream cipher has n Boolean variables $x \in \mathbb{F}_2^n$ as input and one Boolean output $z \in \mathbb{F}_2$ at each round. Therefore, for each output bit z, the cipher is a Boolean function $f(x_1, \ldots, x_n) = z$ and its ANF is:

$$f(x_1, \ldots, x_n) = \sum_{I \subseteq [\![1,n]\!]} \prod_{i \in I} x_i \quad = z$$

We will note x^I the monomial $\prod_{i \in I} x_i$.

An important observation is that for any function f, we can recover the monomial with the highest degree using a sum. For example, the function $f(x_1, x_2) = x_1x_2 + x_1 + x_2$ can be summed with all the possible combination of variables $C = \{(x_1, x_2), (x_1, 0), (0, x_2), (0, 0)\}$ as:

$$\begin{aligned}\sum_C f(x_1, x_2) &= f(x_1, x_2) + f(x_1, 0) + f(0, x_2) + f(0, 0) \\ &= (x_1x_2 + x_1 + x_2) + (0 + x_1 + 0) + (0 + 0 + x_2) + (0 + 0 + 0) \\ &= x_1x_2\end{aligned}$$

This can be used to make a distinguisher on simple functions [EJT07].

In the context of a stream cipher, the function f has two types of variables: The secret key variables k and the IV variables v. The cube attack uses the sum idea to sum the function f on the IV variables. A cube is a monomial on the IV variables v^I. Given this cube, the polynomial of f can then be separated into two polynomials: the polynomial for which each monomial contains the cube ($v^I p_I$) and rest of the ANF q.

$$f(k, v) = v^I p_I + q$$

The polynomial p_I is called the *superpoly* of I in f. It is a polynomial over non-cube variables.

Example 2.1 *Let f be the Boolean function described by the following ANF.*

$$k_1 v_1 v_2 + k_2 v_3 + v_1 v_2 + k_1 + 1$$

Let $v^I = v_1 v_2$ be a cube. The ANF can be split according to v^I as follows:

$$v^I p_I = v_1 v_2 (k_1 + 1) \qquad q = k_2 v_3 + k_1 + 1$$

In this example, the superpoly is $p_I = k_1 + 1$.

If C_I is the set of all the possible values that the variables of the cube can take, then the sum over all these values reduces to the superpoly:

$$\sum_{c^I \in C_I} \left(c^I p_I + q \right) = p_I$$

Indeed, each monomial in the polynomial q lacks at least one variable from the cube (say v_i). Consequently, these monomials will be eliminated in the sum since they will appear an even number of times: once when $v_i = 0$ and once when $v_i = 1$. Moreover, only the cube $(1, 1, \ldots, 1)$ will not cancel the superpoly monomials. As a consequence, the sum over the values of the cube is equal to the superpoly.

To make a cube attack, we need to use cubes to recover superpolys. These superpolys can then form a system of equations that can be solved in some situations. For example, if the superpolys have a maximum degree of one, we can set a linear system of equations and use it to recover bits of the key. For example, if we find a cube $v^I = v_1 v_2$ with a

linear superpoly of one monomial k_1 then we can retrieve this bit of the key. To do so, we encrypt a message for each value of the cube C_I. We sum the output bits of all these encryptions to recover the first value of the secret key k_1. This secret bit is found with only $\#C_I$ encryptions and thus reduces the cipher security from 2^n to $2^{n-1} + \#C_I$. If we use more cubes to recover more bits, the complexity is $2^{n-s} + \#(\bigcup_{I \in L} C_I)$ where s is the number of bits recovered, and L is a list of cubes. Of course, this is an example, and such perfect superpolys may never exist. In the case of a more complex superpoly, the encryptions will give the right-hand side of a non-linear equation $p_I = \sum_{v \in C_I} f(k, v)$. To solve the non-linear system, we have to use more complex algorithms like Gröbner bases, XL or XSL algorithms [AFI+04, CKPS00, CP03, YCC04, CP02]. Therefore, the efficiency of the cube attacks depends on the ability to find small-degree superpolys with small cubes, and we will focus on this problem. The most successful methods to retrieve some superpolys are based on the monomial prediction and its variants.

2.1.3 Monomial prediction

The monomial prediction is a method to recover the ANF of an iterated function. Fortunately, stream ciphers are iterated functions, so we can use it to search for superpolys.

The monomial prediction has been introduced in [HSWW20] and used in [HST+21] to find the best-known superpolys for Trivium, Grain, and Kreyvium (a Trivium variant). The monomial prediction uses a notion of trail. A trail is the propagation of a monomial through iterated functions. We introduce it with the following example.

Example 2.2 (Running example) *Consider the functions f and g defined as follows:*

$$\begin{array}{rcccl} (y_1, y_2) & = & f(x_1, x_2, x_3) & = & (x_1 + x_3,\ x_1x_2 + x_1) \\ (z_1, z_2) & = & g(y_1, y_2) & = & (y_1y_2,\ y_1 + y_2) \end{array}$$

Assume that our cipher is given by $g \circ f$. We can compute the ANF of the entire cipher to determine the monomials that compose this function, but performing this computation is not possible on real ciphers. Here, we have that:

$$(z_1, z_2) = (g \circ f)(x_1, x_2, x_3) = (x_1 + x_1x_2 + x_1x_3 + x_1x_2x_3,\ x_3 + x_1x_2)$$

Instead of computing the ANF, consider the monomial x_1. Since the ANF of f and g are known, we can compute the ANF of each monomial of f and g and deduce the trails of

x_1 (a trail is noted with a succession of arrows):

ANF of each monomial of f	*monomial trails of x_1*
$y_1 = \underline{x_1} + x_3$	$x_1 \to y_1$
$y_2 = x_1x_2 + \underline{x_1}$	$x_1 \to y_2$
$y_1y_2 = x_1x_2x_3 + x_1x_2 + x_1x_3 + \underline{x_1}$	$x_1 \to y_1y_2$
ANF of each monomial of g	*monomial trails of x_1*
$z_1 = \underline{y_1y_2}$	$x_1 \to y_1y_2 \to z_1$
$z_2 = \underline{y_1} + \underline{y_2}$	$x_1 \to y_1 \to z_2, x_1 \to y_2 \to z_2$
$z_1z_2 = \underline{y_1y_2} + \underline{y_1y_2} = 0$	

We have one trail from x_1 to z_1, so we know that x_1 appears in the ANF of z_1. However, there are two trails from x_1 to z_2. If we compute the complete ANF of z_2, we can see that the two occurrences of the monomial x_1 cancel each other $z_2 = \underline{x_1} + x_3 + x_1x_2 + \underline{x_1} = x_1x_2 + x_3$. More generally, if there is an even number of trails, we can deduce that x_1 will not appear in the ANF of z_2.

To recover a polynomial in the ANF of an iterated function, we have to consider all the monomials and to decide whether there is an even or an odd number of trails for each of them.

Division property. The division property is an older method to recover some information about the ANF of an iterated function. The first versions of the method were unable to count the number of trails and thus could only say that some monomials were not in the ANF [DS09, TM16]. Therefore the method does not allow us to recover the exact superpoly, nevertheless it is still useful as it broke the full version of the cipher MISTY [Tod17]. The division property was upgraded several times [BC16, HLM+20, HLM+21]. The last upgrade, namely the three-subset bit-based division property without unknown subset, is able to count exactly the number of trails and therefore is equivalent to the monomial prediction.

MILP models for monomial prediction. Several algorithms have been developed to evaluate the monomial propagation on ciphers. Some are based on the so-called breadth-first search algorithm [Tod15, TM16] whereas some others implement this search using the MILP method [HLM+20].

In MILP models, the monomial propagation is represented within tables. For example, the tables below represent respectively the behaviours of the functions f and g.

x_1	x_2	x_3	y_1	y_2
0	0	0	0	0
1	0	0	1	0
0	0	1	1	0
1	0	0	0	1
1	1	0	0	1
1	1	0	1	1
1	0	0	1	1
1	1	1	1	1
1	0	1	1	1

y_1	y_2	z_1	z_2
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0
0	0	1	1

The tables are small for this simple example. However, for real ciphers computing the whole propagation table of the round function may be infeasible or requires too many inequalities to be described in MILP solvers. Hence, in most cases, propagation rules are added for each basic operator of the cipher (xor, and, copy). Each of the three rules are modelled with the following equations (a, b, c are Boolean variables):

$$a \texttt{ and } b = c \iff \begin{cases} a = c \\ b = c \end{cases} \qquad a \texttt{ xor } b = c \iff \begin{cases} a + b = c \\ a + b \leq 1 \end{cases}$$

$$(a, b) = \texttt{copy}(c) \iff \begin{cases} a + b \geq c \\ c \geq a \\ c \geq b \end{cases}$$

A MILP model is then constructed from these linear constraints. For example, the

model for our running example is:

$$\begin{aligned}(x_{11}, x_{12}, x_{13}) &= \text{copy}(x_1)\\ y_1 &= \text{xor}(x_{11}, x_3)\\ a &= \text{and}(x_{12}, x_2)\\ y_2 &= \text{xor}(a, x_{13})\\ (y_{11}, y_{12}) &= \text{copy}(y_1)\\ (y_{21}, y_{22}) &= \text{copy}(y_2)\\ z_1 &= \text{and}(y_{11}, y_{21})\\ z_2 &= \text{xor}(y_{12}, y_{22})\end{aligned}$$

The main solver used in the literature to solve MILP models is Gurobi [1] because it is one of the fastest and it allows solution enumeration. However, the monomial propagation models become very large when we try to find the superpoly on higher rounds of TRIVIUM. The solution usually used to reduce the problem size is to cut the search space with one or multiple cuts. The idea is that when the model is too big to be solved efficiently, we cut it into several smaller models. This can be done with one arbitrary cut [HLM+20] or some adaptive cuts like the nested method in [HST+21].

However, MILP solvers like Gurobi are usually made to solve optimization problems. Therefore, we thought that Constraint Programming might be more suited for this enumeration problem and we tested two new CP models.

CP model with table constraints. In CP, we can model the propagation table of the round functions of TRIVIUM with only one table constraint. We recall that the round function of TRIVIUM has the form:

$$F(a, b, c, d, e) = (f, g, h, t, j)$$

with

$$t = a + bc + d + e, \quad a = f, \quad b = g, \quad c = h, \quad e = j$$

Note that d is the variable at the end of the register. Moreover, t is the new variable at the beginning of the register and f, g, h, j are just the shifted variables that becomes respectively a, b, c, e. All the valid tuples to describe this function are given in Table 2.1.

This table constraint (Table 2.1) is not so big for a CP solver, but the issue is that

1. https://www.gurobi.com/

a	b	c	d	e	f	g	h	t	j
0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1
1	0	0	0	0	0	0	0	1	0
0	1	1	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	1	0
0	0	0	0	1	0	0	0	1	0
1	0	0	0	1	0	0	0	1	1
0	1	1	0	1	0	0	0	1	1
0	0	0	1	1	0	0	0	1	1
0	0	0	0	1	0	0	0	1	1
0	0	1	0	0	0	0	1	0	0
0	0	1	0	1	0	0	1	0	1
1	0	1	0	0	0	0	1	1	0
0	1	1	0	0	0	0	1	1	0
0	0	1	1	0	0	0	1	1	0
0	0	1	0	1	0	0	1	1	0
1	0	1	0	1	0	0	1	1	1
0	1	1	0	1	0	0	1	1	1
0	0	1	1	1	0	0	1	1	1
0	0	1	0	1	0	0	1	1	1
0	1	0	0	0	0	1	0	0	0
0	1	0	0	1	0	1	0	0	1

a	b	c	d	e	f	g	h	t	j
1	1	0	0	0	0	1	0	1	0
0	1	1	0	0	0	1	0	1	0
0	1	0	1	0	0	1	0	1	0
0	1	0	0	1	0	1	0	1	0
1	1	0	0	1	0	1	0	1	1
0	1	0	1	1	0	1	0	1	1
0	1	0	0	1	0	1	0	1	1
0	1	1	0	1	0	1	0	1	1
0	1	1	0	0	0	1	1	0	0
0	1	1	0	1	0	1	1	0	1
1	1	1	0	0	0	1	1	1	0
0	1	1	0	0	0	1	1	1	0
0	1	1	1	0	0	1	1	1	0
0	1	1	0	1	0	1	1	1	0
1	0	0	0	0	1	0	0	0	0
1	0	0	0	1	1	0	0	0	1
1	0	0	0	0	1	0	0	1	0
1	1	1	0	0	1	0	0	1	0
1	0	0	1	0	1	0	0	1	0
1	0	0	0	1	1	0	0	1	0
1	1	1	0	1	1	0	0	1	1
1	0	0	1	1	1	0	0	1	1

a	b	c	d	e	f	g	h	t	j
1	0	1	0	0	1	0	1	0	0
1	0	1	0	1	1	0	1	0	1
1	0	1	0	0	1	0	1	1	0
1	1	1	0	0	1	0	1	1	0
1	0	1	1	0	1	0	1	1	0
1	0	1	0	1	1	0	1	1	0
1	1	1	0	1	1	0	1	1	1
1	0	1	1	1	1	0	1	1	1
1	1	0	0	0	1	1	0	0	0
1	1	0	0	1	1	1	0	0	1
1	1	0	0	0	1	1	0	1	0
1	1	1	0	0	1	1	0	1	0
1	1	0	1	0	1	1	0	1	0
1	1	0	0	1	1	1	0	1	0
1	1	1	0	1	1	1	0	1	1
1	1	0	1	1	1	1	0	1	1
1	1	1	0	0	1	1	1	0	0
1	1	1	0	1	1	1	1	0	1
1	1	1	1	0	1	1	1	1	0
1	1	1	0	1	1	1	1	1	0
1	1	1	0	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

Table 2.1 – Monomial propagation table of Trivium

there are a lot of them (three per round). Therefore, the model was not very efficient as it was unable to find the trails of Trivium with 675 initialization rounds (Trivium 675) in less than a day.

CP model with max constraints. Since we can use non-linear constraints in CP, we can also replace the copy and operators with max constraints. The model for one equation of Trivium then becomes:

$$\begin{aligned} \max(a^{copy}, f) &= a \\ \max(y^{tmp}, g) &= b \\ \max(y^{tmp}, h) &= c \\ \max(e^{copy}, j) &= e \\ a^{copy} + y^{tmp} + d + e^{copy} &= t \end{aligned}$$

However, this model was still unable to find the trails of Trivium 675 in less than a day.

CP model difficulties The main problem of these models is that they all model an equation that is close to a linear equation, and a constraint solver is very unlikely to be faster on a linear problem than a linear solver. Moreover, as we increase the number of rounds, the number of possible trails becomes larger. For example, for TRIVIUM 842, there are around 3 millions trails for a superpoly of only several hundred monomials.

The downside of all these approaches is that we can hardly add new properties to cut these redundant trails and strengthen the model, as a global view of the problem is missing.

2.2 New graph representation

In this section, we present a novel and simple graph-based model to recover monomials in iterated functions, and we use it to recover the superpoly of a stream cipher for a given cube. It is a graph representation of the monomial propagation since it can recover the superpoly, but it has the main advantage of being much simpler, more intuitive and easier to manipulate to find new properties.

We represent all the intermediate variables and monomials using a directed graph G. A node of G represents a variable, and an edge from x to y indicates that y appears in the ANF of x. We present a simple example.

Example 2.3 *Recall the functions f and g of our running example (Example 2.2).*

$f \circ g$ can be expressed as a DAG $G = (V, E)$ where :

— *$V = \{x_1, x_2, x_3, y_1, y_2, z_1, z_2\}$ is the set of nodes,*

— *$E = \{(y_1, x_1), (y_1, x_3), (y_2, x_1), (y_2, x_2), (z_1, y_1), (z_1, y_2), (z_2, y_1), (z_2, y_2)\}$ is the set of edges.*

***A trail** is then a DAG with a root and some leaves. The root is the output bit, and the leaves are the variables of the monomial.*

The graph and the trails of x_1 are represented in Figure 2.3. In these graphs, the red double edges represent bit products. To find these trails, we use the following constraints over the edges of a trail $\mathcal{T}$. For example the constraint 2.1 represents the product of y_1 and y_2 in z_1, the constraint 2.2 represents the xor of y_1 and y_2 in z_2. The two last constraints ensure that the trail does not end midway. Note that they are variants of the conservation of flow constraints in the Maximum flow problem i.e., a classic optimization problem where

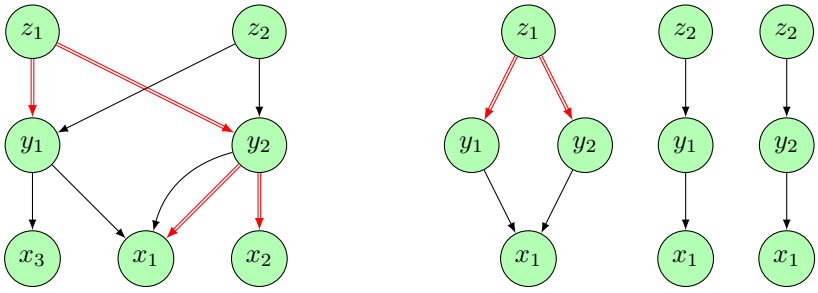


Figure 2.3 – Graph of $f \circ g$ (left) and 3 monomial trails of x_1 (right)

a flow is propagated through a DAG.

$$(z_1, y_1) \in \mathcal{T} \Longleftrightarrow (z_1, y_2) \in \mathcal{T} \tag{2.1}$$

$$(z_2, y_1) \notin \mathcal{T} \vee (z_2, y_2) \notin \mathcal{T} \tag{2.2}$$

$$(y_2, x_2) \in \mathcal{T} \implies (y_2, x_1) \in \mathcal{T} \tag{2.3}$$

$$(y_1, x_1) \notin \mathcal{T} \vee (y_1, x_3) \notin \mathcal{T} \tag{2.4}$$

$$\cup_i((z_i, y_1) \in \mathcal{T}) \Longleftrightarrow \cup_i ((y_1, x_i) \in \mathcal{T}) \tag{2.5}$$

$$\cup_i((z_i, y_2) \in \mathcal{T}) \Longleftrightarrow \cup_i ((y_2, x_i) \in \mathcal{T}) \tag{2.6}$$

It is interesting to compare our model to the model based on monomial propagation with basic propagation rules. The monomial propagation model contains 15 variables, four copy-constraints, three xor-constraints and two and-constraints, which result into 22 linear constraints. Our model has eight variables (the edges) and six constraints (or eight linear constraints).

The main advantage of our model relies on its simplicity and the ease of adding extra constraints to remove false (even) trails and deploying strategies. To illustrate this, we use the graph model to search for some superpoly of the stream cipher Trivium.

2.2.1 Graph of Trivium

Figure 2.4 depicts the Deterministic Finite Automaton (DFA) obtained from the description of Trivium. Note that the DFA develops Trivium backwards: from the output bit to the first round. There are four possible transitions to go from one register (A, B or C) to its successors: three of them are simple, and one is doubling ($\Longrightarrow$). In the following, the three simple transitions will be named the looping ($\cdots>$), the short ($\dashrightarrow$), and the long ($\longrightarrow$) one. For example, in the possible transitions from the register A, 69 is the

looping transition, 66 is the short transition, 111 is the long transition and $\{110, 109\}$ is the doubling transition.

One may have noticed that the DFA relies only on the first bit of each register. Indeed, for the other bits, the application of the round function simply consists of shifting them to the left. The value of the shifted bits only changes when they turn back to the first position of a register. Moreover, none of these shifted bits is a result of a round function, so they have no use in the DFA. In summary, the DFA already simplifies all the shifted bits at each round on each register to focus on the one produced in the corresponding round function (t_1,t_2, and t_3). The node A (resp. B, C) represents the first bit of register A (resp. B, C). Whenever a transition is taken, the generated bit will have to be shifted k times to be on the first position of its register again. The number of shifts k is reported on the edges of the DFA. For example, if the bit at the first position of register A is set to 1 at round R, then it can be propagated to register C or A. If A is selected, then the first bit of A will be activated at round $R - 69$ because it will be shifted $k = 69$ before returning on the first bit of a register. Otherwise, if C is selected, there are two scenarios. Either a simple transition is taken (short or long), which corresponds to the activation of the first bit of C at either round $R - 66$ or $R - 111$. Or the doubling transition is picked, and the first bit of the register C will be activated at round $R - 110$ and $R - 109$.

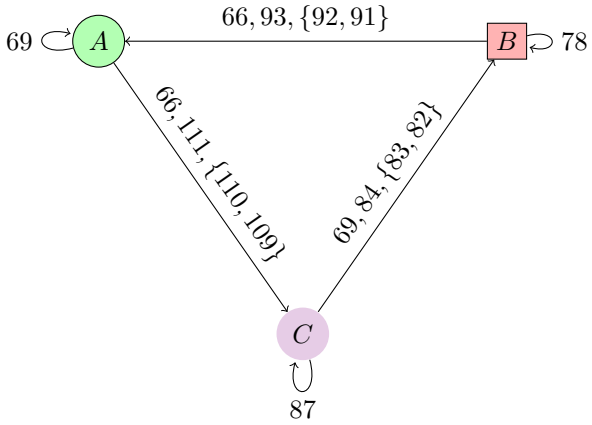


Figure 2.4 – A DFA that encodes possible transitions for the Trivium cipher.

Now we will present the way to build a graph modelling a division trail based on the DFA (Fig. 2.4), using a breadth-first search algorithm (Algorithm 1). The node corresponding to the output bit z at round R is created first and pushed into a queue. This triggers a loop that ends when the queue is empty. A node is popped from the queue and marked as visited. If the node has a positive R value, its child nodes described by the

DFA are created if needed (if they are not in the graph already), added to the queue, and the involved edges are created. Creating a node requires knowing its round R: anytime a transition is visited, the number of shifts k that label it on the DFA is subtracted from the R value of its parent node. If a popped node has a negative R value, which corresponds to the first state of the cipher, no action is performed. When the loop stops, the graph of all possibilities is declared. Such a graph is not very deep, but it is very wide since any node has potentially five child nodes. However, each node in a solution *i.e.,* a trail, has only one or two outgoing edges.

Algorithm 1: MakeGraphfromDFA(*reg*, *round*)

```
Data: reg: starting register, round: starting round, G = (V, E): a graph
1 Push (reg,round) in Queue
2 while Queue is not empty do
3     (reg,round) ← Pop Queue
4     if round > 0 then
5         for (reg2,shift) in successors of reg from DFA do
6             if (reg2,round−shift) not in V then
7                 add (reg2,round−shift) to V
8                 add (reg2,round−shift) to Queue
9             add ((reg,round),(reg2,round−shift)) to E
```

Figure 2.5 shows a graph solution for Trivium 672 with the starter node s_{243}, i.e. the 66^{th} bit of register C. Therefore, the source node is labelled by C with $R = 672 - 66 = 606$ since the bit at position 66 in register C has to be shifted 66 times to be on the first position. The blue nodes are the cube bits, and the red ones are the key bits. This solution represents one trail for the superpoly monomial x_{16}.

Once we have formalized our problem as a graph problem, we can rely on a MILP solver or a CP solver to enumerate all the solutions.

In the following, we choose Gurobi [Gur21] for our MILP model because it already showcased its efficiency on division property, and Choco [PF22] for our CP model since it natively supports constraints over graph variables [Fag15].

2.2.2 Doubling patterns

To retrieve the superpoly, we need to enumerate all the trails and count how many trails there are for each monomial. Indeed, a monomial with an even number of trails will

Figure 2.5 – A solution for TRIVIUM 672 considering s_{243} (66$^{\text{th}}$ bit of register C) as starter node. Blue nodes are the cube bits; the red one is the key bit. Double-line edges are for doubling transitions, plain-line edges for long transitions, dashed-line edges for short transition and dotted-line edges for looping transitions.

not appear in the superpoly.

In the graph-based representation, a doubling pattern is a pair of distinct sub-graphs connecting the same two sets of starting and ending nodes. If a trail uses one sub-graph of a doubling pattern, then the trail using the other sub-graph will produce the same monomial. Therefore, preventing doubling patterns from existing in the graph will reduce the number of trails which cancel each other out. By studying the DFA of TRIVIUM, we identified several doubling patterns.

Pattern 1 (long-double)

Between each pair of registers, there is the long transition with a given number of shifts p, and the doubling transition with $p-1$ and $p-2$ shifts. Therefore, if the doubling edge is followed by two long edges, we will get the same leaves than taking the long edge first and the doubling edge after, as depicted in Figure 2.6.

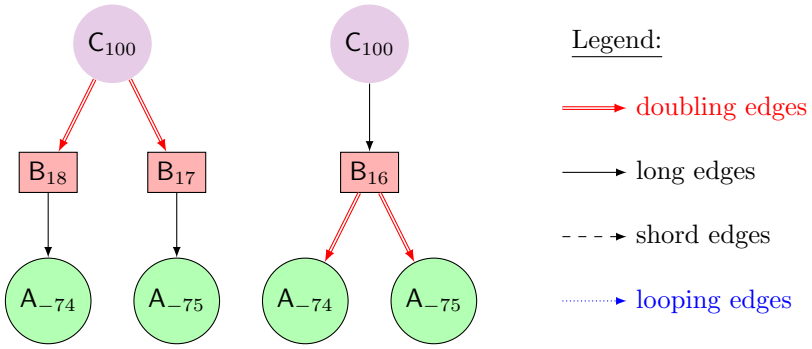


Figure 2.6 – long-double pattern

To discard this pattern, we have to take care that the intermediate nodes, here B_{16}, B_{17}, and B_{18}, are not used in any other part of the trail. Indeed, if one of these nodes is mandatory for another part of the trail, there is only one half of the doubling pattern available, and thus the cancellation will not be correct.

Pattern 2 (3 consecutive bits)

Another doubling pattern is when three bits are at consecutive rounds on the same register, as depicted in Figure 2.7. On this figure, we have three example nodes C_{97}, C_{98}, and C_{99}. If the doubling edges are taken on C_{97} and C_{99}, the long and the doubling edges of the middle bit can then be removed because these two choices lead to the same output nodes (B_{14},B_{15},B_{16},B_{17}). The advantage of this pattern is that there are no intermediate

nodes that can be mandatory in other parts of the trails like for the Pattern 1. However, we still need to take care of the case where we have four consecutive bits or more because if we constrain two times the same edges, the cancellation will not be correct.

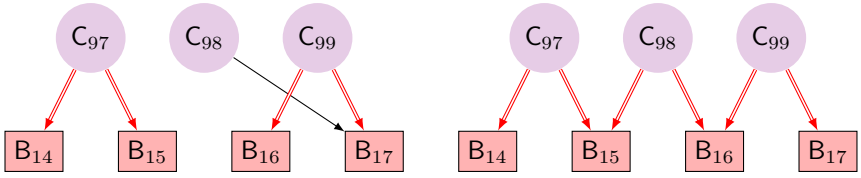


Figure 2.7 – 3 consecutive bits pattern

Pattern 3 (looping)

When a looping transition is taken *i.e.,* the bit stays on the same register, and if all the outgoing edges of the looping register return to the same register at least once in the trail, then a similar result can be obtained by not taking the first looping transition but taking it on each outgoing edge as shown in Figure 2.8 (numbers are omitted for clarity). This pattern has a lot of configurations, and constraining all of them will end up in a large model.

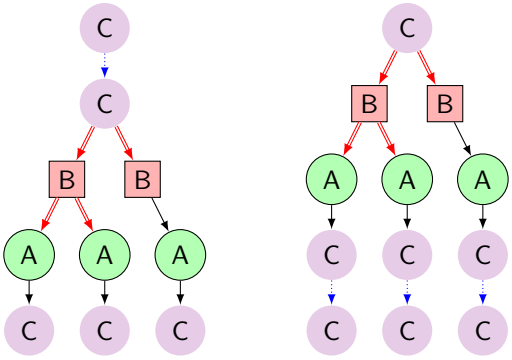


Figure 2.8 – looping pattern

Pattern 4 (simple cycle)

A cycle pattern is completed whenever the path returns to the first register without doubling, then taking any different edge of the cycle after is a doubling pattern. Indeed, any edge after the cycle could be taken before the cycle, as shown in Figure 2.9.

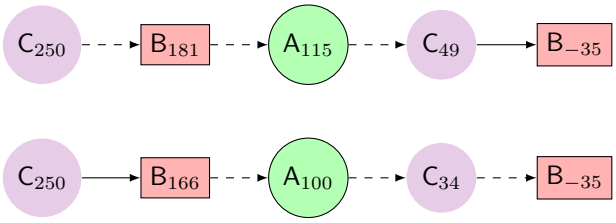


Figure 2.9 – simple cycle pattern

By considering all these patterns, it seems possible to reduce the number of even solutions and save some useless trail explorations without changing the parity of each solution. However, we faced many problems. First, adding the constraints for all these patterns slows down the solving process and finding the right trade-off between solution space reduction and time consumption is not easy. Second, we have to ensure that a doubling pattern does not interfere with another one. More precisely, let (p_1, p_2) be doubling patterns. We may have an issue if it is possible to reach a trail containing p_1 while it is impossible to reach p_2 because of another doubling pattern. This is a big issue because the parity of the trails is very important. We want to make sure that each constrained pattern is free to be constrained. One option to do this is to apply a constraint if and only if all the nodes involved in a doubling pattern are not reached by other edges than the ones from the pattern. However, in practice, doing so highly limits the number of times doubling patterns are applied. As a result of our experimentation, we decided to take into account pattern two only as it seems to be the most efficient.

Thus selecting the right patterns to add to the model is still an open and interesting question.

2.2.3 Arity approximation

The idea of approximating the number of cube bits reachable for each bit of the cipher was explored in [Liu17], and we propose to use it to reduce the search space in our graph-based model.

The reasoning on arity is as follows. Starting from the bits of the IV and going back to the active bit, each intermediate node aggregates an over-approximation of the number of bits of the cube that it would allow to reach if we took it. This value is called arity and is built by consulting all or part of its descendants. Under certain conditions between the arity of a node and the arity of its predecessors or successors, it is possible to deduce whether the node may belong to a trail or not. This is particularly useful in the search

for superpolys because to have small superpolys, we constrain big cubes. For example, the cube size for TRIVIUM 841 [HLM+20] is 78 bits. Therefore, we can discard the part of the graph that do not achieve at least this arity.

As shown in [Liu17], an approximation regarding the arity can be computed by recursively taking two consecutive transitions into account and by propagating the arity from the cube to the output bit.

Example 2.4 *Consider the case where one wants to compute the arity of register* C *at round* 100*, and the doubling transitions are selected.*

$$ar(\mathsf{C}_{100}) = ar(\mathsf{B}_{18}) + ar(\mathsf{B}_{17}) \tag{2.7}$$

These terms are developed as follows:

$$ar(\mathsf{B}_{18}) = \max(ar(\mathsf{B}_{-60}), ar(\mathsf{A}_{-48}), ar(\mathsf{A}_{-75}), ar(\mathsf{A}_{-74}) + ar(\mathsf{A}_{-73})) \tag{2.8}$$
$$ar(\mathsf{B}_{17}) = \max(ar(\mathsf{B}_{-61}), ar(\mathsf{A}_{-49}), ar(\mathsf{A}_{-76}), ar(\mathsf{A}_{-75}) + ar(\mathsf{A}_{-74})) \tag{2.9}$$

Suppose now that the arity of registers with negative rounds are all equal to the same value, say the value 1. Then (2.8) *and* (2.9) *can be simplified to:*

$$ar(\mathsf{B}_{18}) = ar(\mathsf{A}_{-74}) + ar(\mathsf{A}_{-73}) \tag{2.10}$$
$$ar(\mathsf{B}_{17}) = ar(\mathsf{A}_{-75}) + ar(\mathsf{A}_{-74}) \tag{2.11}$$

By replacing $ar(\mathsf{B}_{18})$ *and* $ar(\mathsf{B}_{17})$ *by* (2.10) *and* (2.11) *in* (2.7)*, we remark that the arity of* A_{-74} *is counted twice. In the graph representation, the node labelled* A_{-74} *is reachable multiple times from* C_{100}*. Such an over-approximation would be accumulated along the way to the output bit, making the bound pretty far from the real value. An example can be found in Figure 2.10 where the root node would be wrong by 20 on the arity for counting only the first successors, and this error is growing when the information is further propagated to the predecessors.*

For a given node, the approximation of its arity has to take into account all the child cases of the doubling edges and take the maximum of their arities to better approximate the arity of the source. Note that a similar reasoning can also be applied to compute the minimum arity of a node.

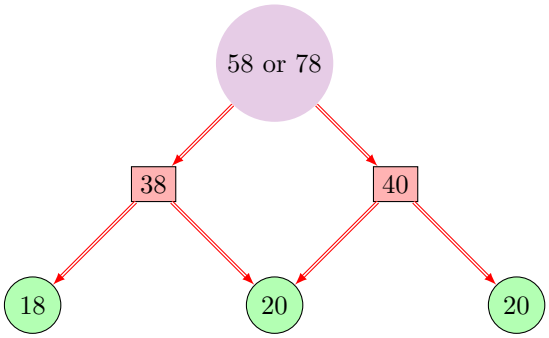


Figure 2.10 – Arity example

Let n be a node and a, b, c, d, e its successors respectively obtained with the looping, short, long and the two doubling edges as well as the successors of the successors aa, ab, ac, ad, ae obtained in the same way. The arity of n can be simply computed as:

$$n = max(a, b, c, d + e)$$

To detect the redundancies, we must develop the long and doubling edges:

$$n = max(a, b, max(ca, cb, cc, cd + ce), max(da, db, dc, dd + de) + max(ea, eb, ec, ed + ee))$$

We know from the description of TRIVIUM that the long and the doubling edges are always side by side with one round separating them as seen in Figure 2.11. Therefore we have that $cd = dc$, $ce = dd = ec$, and $de = ed$. We can use it to reduce the formula and keep only one of each in every scenario.

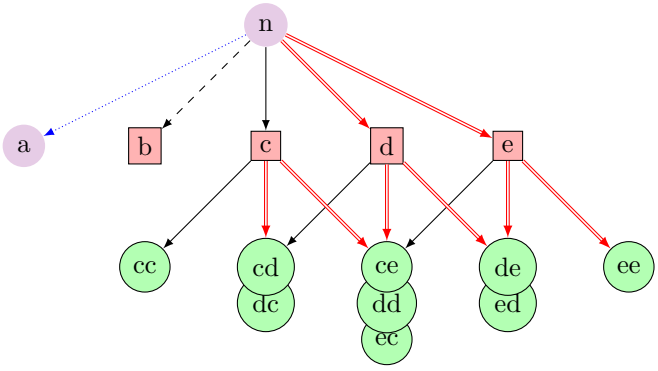


Figure 2.11 – Arity formula redundancy on two rounds

$$n = max(a, b, c, da + e, db + e, dc + e, dd + de + max(ea, eb, ee))$$

This approximation of the arity can then be used as a strategy or a constraint. Since the goal is to find the superpoly, it is expected that a significant part of the graph will be cut from the search due to its low arity.

2.3 New models using our graph representation

This section presents the different models we made with our graph representation. For the most efficient ones, we added some constraints for the doubling patterns and tried to use the arity approximation. Finally, we tested our models to compare them to state-of-the-art methods on round reduced TRIVIUM (675, 735, 840, 841, 842 rounds).

2.3.1 SAT

SAT solvers can solve Boolean satisfaction problems encoded in the Conjunctive Normal Form (CNF). The CNF is a conjunction of disjunction of variables. It can be seen as the opposite idea of a table since a Boolean table constraint is essentially a disjunction of conjunctions, which is a Disjunctive Normal Form (DNF). All the edges of the graph of TRIVIUM are Boolean variables, so we can model the round function of TRIVIUM as follows:

Let p be the input edges of a node and s the output edges such that s_1, s_2, s_3 are respectively the looping edge, the short edge and the long edge, and s_4, s_5 are the doubling edges. The Boolean formula for the round function of TRIVIUM is:

$$\bigwedge_{i\in[\![1,5]\!]} \neg p_i \bigvee_{i\in[\![1,3]\!]} (s_i \bigwedge_{j\in[\![1,3]\!], j\neq i} \neg s_j) \vee \bigwedge_{i\in[\![1,3]\!]} \neg s_i \bigwedge_{i\in[\![4,5]\!]} s_i$$

We introduce the five variables, P for the predecessor edges, S_i for each simple successor

edge and Sd for the doubling edge:

$$\begin{aligned} P &= \bigwedge_{i\in[\![1,5]\!]} \neg p_i \\ S_i &= s_i \bigwedge_{j\in[\![1,3]\!],j\neq i} \neg s_j \quad \forall i \in [\![1,3]\!] \\ Sd &= \bigwedge_{i\in[\![1,3]\!]} \neg s_i \bigwedge_{i\in[\![4,5]\!]} s_i \end{aligned}$$

We transform the first formula and the conservation of flow constraints in the following list of clauses:

$$\begin{aligned} \neg P \vee \neg p_i \quad \forall i \in [\![1,5]\!] \qquad & \neg S_i \vee s_i \quad \forall i \in [\![1,3]\!] \\ P \bigvee_{i\in[\![1,5]\!]} p_i \qquad & \neg S_i \vee \neg s_j \quad \forall i \in [\![1,5]\!] \\ \neg P \vee \neg s_i \quad \forall i \in [\![1,5]\!] \qquad & \neg Sd \vee s_i \quad \forall i \in [\![4,5]\!] \\ P \bigvee_{i\in[\![1,3]\!]} S_i \vee Sd \qquad & \neg Sd \vee \neg s_i \\ Sd \bigvee_{i\in[\![1,3]\!]} s_i \bigvee_{i\in[\![4,5]\!]} \neg s_i \qquad & S_i \vee \neg s_i \bigvee_{j\in[\![1,5]\!],j\neq i} s_j \end{aligned}$$

The SAT solvers usually only solve satisfaction problems. However, when it finds a solution, we can restart it with a new constraint that discards this solution, and thus count the number of solutions. There is still one problem with this in our case. Due to the number of solutions (3 million for 842 rounds), the model becomes very hard and cannot be solved within a day for Trivium 840.

2.3.2 CP

Constraint programming [RvBW06] is a technique for solving combinatorial problems. Unlike SAT and MILP, it is not necessary to express the rules solely in terms of linear constraints or Boolean clauses. In addition, CP solves a problem in a way similar to branch-and-bound except that it eliminates, by *filtering*, impossible states or combinations. CP techniques have already been successfully applied to cryptanalysis problems [GMS17, GLMS20, DDH+21].

Here, we use a directed graph variable G [DDD05, Fag15]. In CP, a graph variable G

has a domain defined by a graph interval $[\underline{G}, \overline{G}]$. $\underline{G}$ is the lower bound of G and defines nodes and edges that must appear in any solution. In our case, it is declared with the mandatory nodes of the cube. $\overline{G}$ is the upper bound of G and defines nodes and edges that can appear in any solution. In our case, it is the full graph developed from the automaton. A solution is found when $\overline{G} = \underline{G}$. The solving processes by adding nodes or edges from $\underline{G}$ or by removing nodes or edges from $\overline{G}$. Such modifications are triggered by constraints defining properties on G that need to be satisfied in any solution.

In the following, D is a view ([JP22]) of G, which only contains doubling edges and the endpoint nodes; L is another view of G, which only contains the long edges and the endpoint nodes; K stores leaf nodes of G. The functions $pred_X(n)$ and $succ_X(n)$ give the predecessors and the successors of a node n in the (sub-)graph X.

The graph model is declared as (2.12) $\wedge$ ((2.13) $\vee$ (2.14)) $\wedge$ (2.15) $\wedge$ (2.16) where:

$$|pred_G(n)| > 0 \qquad \forall n \in \underline{G},\ n \neq source \tag{2.12}$$

$$|succ_G(n)| = 1 \wedge (n, s) \notin D \qquad \forall n \in \underline{G},\ n \notin K, \forall s \in succ_G(n) \tag{2.13}$$

$$|succ_G(n)| = 2 \wedge (n, s) \in D \qquad \forall n \in \underline{G},\ n \notin K, \forall s \in succ_G(n) \tag{2.14}$$

$$(n, s_1) \in \underline{G} \iff (n, s_2) \in \underline{G} \qquad \forall n \in \underline{G},\ (n, s_1) \in D, (n, s_2) \in D \tag{2.15}$$

$$(n, s_1) \notin \overline{G} \iff (n, s_2) \notin \overline{G} \qquad \forall n \in \overline{G},\ (n, s_1) \in D, (n, s_2) \in D \tag{2.16}$$

The constraint (2.12) ensures that each node selected in a solution has at least one predecessor except the source node. Constraints (2.13) and (2.14) maintain the number of successors of each node except the leaf ones. If a given node takes a simple transition, then it has exactly one successor; if it takes a doubling transition, then it has exactly two successors. The two conditions cannot hold simultaneously. Finally, constraints (2.15) and (2.16) ensure that either a single edge or a pair of doubling edges is selected.

The doubling constraints are added to the CP model in the form of clauses expressed on the disjoint membership of edges in G and are propagated using an SAT-like constraint. The algorithm for estimating the degree of TRIVIUM-like ciphers [Liu17] can directly be integrated into the graph model as an additional constraint. Without going into too much details, it imposes to declare additional integer variables that we call RIV (for Reachable Initialization Vector). An integer variable v has a domain $[\underline{v}, \overline{v}]$ where $\underline{v}$ (resp. $\overline{v}$) denotes the smallest (resp. the largest) value it can be assigned to. The RIV variables store, for each node in $\underline{G}$, an approximation of the number of nodes of the cube it can reach.

The algorithm [Liu17] is directly applied dynamically to refine the bounds of each RIV_i variable associated to node i, based on RIV_j, $\forall j \in succ_G(i)$. It is important to note that RIV variables are bounded as long as the involved nodes and edges are in $\overline{G}$. When a RIV domain is emptied or is inconsistent with those of its neighbours, then the corresponding node is removed from $\overline{G}$.

Strategy. Unlike Gurobi, we can fully control the strategy deployed by Choco. However, this solver is inherently sequential, and thus we decided to apply the divide-and-conquer strategy to run several instances in parallel. The main issue we faced was that only a few instances were hard to solve. Thus, we regularly need to redivide models in order to maximize the use of available cores.

2.3.3 MILP

Mixed Integer Linear Programming aims at solving problems described with linear constraints. The MILP graph model is written as a relaxed flow problem. A flow problem is usually defined with the conservation of flow constraints. This constraint states that anything that enters a node must leave it. In our case, this is relaxed because multiple incoming transitions are possible. Having multiple incoming transitions means that a variable is in the monomial multiple times. Regardless of the incoming number of edges, if it is reached, then the out transition is either simple or double.

In the following, $Pred(i)$ gives all the predecessors of the node i, and $Succ(i)$ gives all the linear successors and one of the doubling successors. The functions $brother_1(i)$ and $brother_2(i)$ give the two doubling sons of i.

First, all the edges are declared as Boolean variables:

$$X_{i,j} = \begin{cases} 1 & \text{if the edge } (i,j) \text{ is in the trail} \\ 0 & \text{otherwise} \end{cases}$$

To implement the graph model of TRIVIUM, we add the following constraints:

$$\sum_{j \in Pred(i)} X_{j,i} \geq \sum_{j \in Succ(i)} X_{i,j} \qquad \forall i \in V \tag{2.17}$$

$$\sum_{j \in Pred(i)} X_{j,i} \leq |Pred(i)| \sum_{j \in Succ(i)} X_{i,j} \qquad \forall i \in V \tag{2.18}$$

$$X_{i,brother_1(i)} = X_{i,brother_2(i)} \qquad \forall i \in V \tag{2.19}$$

$$\sum_{j \in Succ(i)} X_{i,j} \leq 1 \qquad \forall i \in V \tag{2.20}$$

The constraints (2.17) and (2.18) are the conservation of flows constraints while (2.19) and (2.20) are related to the edges outputting a node (and thus dedicated to TRIVIUM). The cube and the output bit are constrained in the solution by the following:

$$\sum_{j \in Pred(i)} X_{j,i} \geq 1 \qquad \forall i \in cube \tag{2.21}$$

$$\sum_{j \in Pred(i)} X_{j,i} = 0 \qquad \begin{matrix} \forall i \in leaves, \\ i \notin cube,\, i \notin key, \\ i \notin non\text{-}zero\text{-}constants \end{matrix} \tag{2.22}$$

The key bits are free, as well as the non-zero constants, because they can also appear in the superpoly.

Constraints for doubling patterns. The MILP model can be strengthened with constraints to discard the doubling patterns. Let P be a set of doubling patterns (p_1, p_2) with p_1, p_2 sub-graphs with the same sources and the same leaves.

We used only Pattern 2, for which both the sub-graphs p_1 and p_2 are composed of, respectively, 5 and 6 edges of the form:

— $p_1 = \{(x_1, y_1), (x_1, y_2), (x_2, y_4), (x_3, y_3), (x_3, y_4)\}$
— $p_2 = \{(x_1, y_1), (x_1, y_2), (x_2, y_2), (x_2, y_3), (x_3, y_3), (x_3, y_4)\}$

Thanks to the equalities of doubling edges, we can simplify both p_1 and p_2 such that:

— $p_1 = \{(x_1, y_2), (x_2, y_4), (x_3, y_4)\}$
— $p_2 = \{(x_1, y_2), (x_2, y_2), (x_3, y_4)\}$

Because (x_2, y_4) and (x_2, y_2) cannot be active both at the same time, we can add the inequality $X_{(x_1,y_2)} + X_{(x_2,y_2)} + X_{(x_2,y_4)} + X_{(x_3,y_4)} \leq 2$ to remove both p_1 and p_2. However, a problem occurs if the node x_4 consecutive to x_3 is active and reaches y_4. Indeed, the configuration for which the four consecutive nodes follow the doubling edges would be

removed twice. Thus we modified the inequality into:

$$2X_{(x_1,y_2)} + 2X_{(x_2,y_2)} + X_{(x_2,y_4)} + 2X_{(x_3,y_4)} - X_{(x_4,y_4)} \leq 4$$

This inequality forbids three consecutive nodes to all take the doubling edge and forbids x_2 to take the long edge if x_4 does not take the doubling edge. Thus adding this inequality to all consecutive nodes leads to the right ANF.

Strategy. In both [HLM+20] and [HLLT20], authors used a divide-and-conquer strategy together with their MILP models. Basically, they developed the polynomial of the root node for several hundreds of rounds (between 200 and 400) and then applied their models to each monomial of the polynomial. Without this strategy, the solving times are much higher, making it unfeasible to retrieve the superpoly in a reasonable time. This shows that Gurobi fails to identify the right variables to branch on. While Gurobi does not allow the user to fully control the branch-and-cut strategy, it offers several options to modify it, and we mainly used two of them:

— **BranchPriority:** With this option, it is possible to give each variable of the model a priority during the selection of the next variable to branch on. We tried several strategies, and it seems that the best choice is to sort the variables according to their arity. More precisely, given an edge (x, y), we choose to:

1. set a negative priority if $ar(y) \leq 0$, i.e. if the variable y cannot lead to any cube variable;
2. set the priority to zero for all simple edges, based on the idea that we have to focus on doubling edges;
3. set the priority to $ar(x)$ for all doubling edges to focus on the edges which can reach the most cube variables.

— **VarHintVal:** With this option, we can tell Gurobi that we think the value of a variable will be in a solution. We choose to set to 0 all simple edges, again to focus on doubling edges.

Using both those options, it became unnecessary to use the divide-and-conquer strategy as we reached approximately the same running times with and without it. However, we believe there is still room for improvement. First, the **BranchPriority** is static, while a dynamic approach would be much better. Second, both options above apply to variables only, while we may want to use them on linear combinations of variables. The problem is

that if we create a new variable x and add a constraint $x = y + z$, x will be removed from the model during the presolve, and it seems Gurobi does not keep its branch priority.

2.3.4 Results regarding the CP and MILP models

We ran our new models together with the MILP ones from both [HLM+20] and [HSWW20] on our server (AMD EPYC 7742 64-Core Processor), limiting the number of available cores to 32. Indeed, Gurobi supports parallelism but faces issues when the number of threads is too high and the model is too large. Actually, the large models are slower if we run them with 64 threads.

Results are given in Tables 2.2 and 2.3 while the cubes used to perform our experiments are detailed in Table 2.4. The code is publicly available at https://gitlab.inria.fr/agontier/trivium-superpoly

Model	[HSWW20]	[HLM+20]	MILP Graph	CP Graph
$R = 675$	3m	1m	3s	15s
$R = 735$	4m	2m	10s	31m
$R = 840/1$	472m	269m	10m	> 24h
$R = 840/2$	316m	91m	10m	
$R = 840/3$	351m	108m	6m	
$R = 841$	956m	282m	19m	
$R = 842$	> 24h	990m	182m	

Table 2.2 – Results on Trivium

We see that the graph model of Trivium performs better with the MILP implementation and the Gurobi solver. One explanation might be that Trivium is not highly combinatorial. Indeed, the round function of Trivium has only one non-linear case, and it is a simple product. Our graph model implemented in MILP is consistently much faster than the models from Hu *et al.* [HSWW20] based on monomial prediction and from Hao *et al.* [HLM+20] based on division property. One important note is that our models do not need the cut proposed in [HLM+20]. In fact, our model is slower if we add the same cut. We think that this is mainly due to the arity approximation strategy that fulfils the same role more efficiently.

Regarding the number of trails outputted by our model, it is reduced by a factor between 2 and 4, which shows how useful are the doubling patterns we described. But as we already explained, we were not able to use all of them. Checking a posteriori the trails for Trivium 842 shows that taking into account all the doubling patterns could remove much more trails. We believe this is an interesting research direction for future work.

Graph solver	$R = 840/1$	$R = 841$	$R = 842$
without any patterns	12 909	30 177	3 188 835
with Pattern 2	5 953	18 929	720 779

Table 2.3 – Number of solutions

Rounds	Cube indices
675	3, 14, 21, 25, 38, 43, 44, 47, 54, 56, 58, 68
735	2, 5, 9, 12, 13, 14, 19, 28, 36, 38, 40, 47, 49, 51, 52, 53, 55, 57, 63, 64, 66, 73, 79
840 /1	$IV \setminus \{34, 47\}$
840 /2	$IV \setminus \{71, 73, 75, 77, 79\}$
840 /3	$IV \setminus \{73, 75, 77, 79\}$
841	$IV \setminus \{9, 79\}$
842	$IV \setminus \{19, 35\}$

Table 2.4 – Cubes used in our experiments for Trivium

2.4 Conclusion

In this work, we introduced a new graph representation of the superpoly recovery problem for the symmetric stream cipher Trivium. We found some patterns and we were able to integrate them as well as the arity approximation in a MILP model to solve this problem more efficiently. This work was published in 2021 and more research has been done on this topic latter on. In the following, we present some recent work on the superpoly recovery of Trivium and we give the DFA representation of Grain, another stream cipher that could be modeled within our graph representation in some future work.

2.4.1 Nested monomial propagation

A recent update on the monomial prediction is the nested monomial prediction [HST+21]. This method was able to recover a superpoly for Trivium 845 and then Trivium 848 among other superpolys in [HHPW22]. The idea is to cut the problems into sub-problems whenever they are too hard to solve *i.e.,* if they reach a time limit. This allows for clean parallelism and smaller models for Gurobi to handle.

2.4.2 Nested graph model and new patterns

In 2023, [CQ23] improved our work by proposing an algorithm to search for all the doubling patterns with one layer of edges (like the three consecutive bit patterns). They focus on one-layer patterns only because multi-layer patterns are hard to constrain with all the possible overlaps we have seen before. They were able to find 11 new patterns and found that these doubling patterns are most useful in the first rounds of the cipher. Moreover, they also improved the graph-based model by adding the nested method [HST+21]. As a result, they were able to find a superpoly for Trivium 843 on limited hardware (32 threads and 96G of RAM) in 36h. For some comparison, our graph model alone can find the superpoly of Trivium 842 in around three hours, whereas the graph model with the nested approach finds it in eleven hours. However, the graph model alone encounters memory issues for higher cubes. A problem partially solved by the nested approach. This work showed that studying doubling patterns can lead to more efficient models.

2.4.3 Ternary world of Trivium

Some unpublished work on the graph model has also been done by A. Derrien and C. Prud'homme. They noticed that all the linear transitions of Trivium are multiple of three. The doubling transitions are $3n + 1$ and $3n + 2$. We can see it as three worlds that are connected by the doubling edges. On the trails example of Figure 2.12, we represented the three worlds by 0, 1 and 2.

When searching for a superpoly, we know the output bit and the cube. Therefore we may be able to use the ternary property to know which transition of which type is needed to reach the cube. This information may be used for additional constraints or refined strategies for future models.

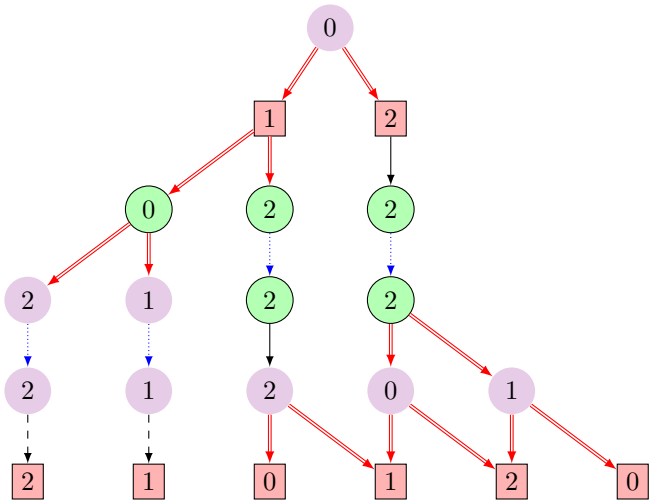


Figure 2.12 – Example trail with world number on nodes

2.4.4 Graph model on Grain

Grain is another stream cipher of the eSTREAM competition [HJM07]. It has two registers, a bigger key than Trivium (128 bits) and a much more complex round function. However, it has fewer initialization rounds, 256 (versus 1152 for Trivium).

Grain description. The two 128-bit registers are initialized with the key and the Initialization Vector:

$$\begin{aligned}(b_0, \ldots, b_{127}) =& (K_1, \ldots, K_{128}) \\ (s_0, \ldots, s_{127}) =& (IV_1, \ldots, IV_{96}, 1, \ldots, 1, 0)\end{aligned}$$

Each round, the registers will be updated with the following functions:

$$\begin{aligned}(b_0, \ldots, b_{127}) \leftarrow& (b_1, \ldots, b_{127}, g + s0 + z) \\ (s_0, \ldots, s_{127}) \leftarrow& (s_1, \ldots, s_{127}, f + z)\end{aligned}$$

where g, f, h, and z are the following functions:

$$\begin{aligned}
g \leftarrow & b_0 + b_{26} + b_{56} + b_{91} + b_{96} + b_3 b_{67} + b_{11} b_{13} + b_{17} b_{18} + b_{27} b_{59} \\
& + b_{40} b_{48} + b_{61} b_{65} + b_{68} b_{84} + b_{22} b_{24} b_{25} + b_{70} b_{78} b_{82} + b_{88} b_{92} b_{93} b_{95} \\
f \leftarrow & s_0 + s_7 + s_{38} + s_{70} + s_{81} + s_{96} \\
h \leftarrow & b_{12} s_8 + s_{13} s_{20} + b_{95} s_{42} + s_{60} s_{79} + b_{12} b_{95} s_{94} \\
z \leftarrow & h + s_{93} + b_2 + b_{15} + b_{36} + b_{45} + b_{64} + b_{73} + b_{89}
\end{aligned}$$

Grain DFA. We can develop the update functions and deduce the automaton of GRAIN. Note that they are much more complex doubling edges in the h function. There are bit products combining bits of the two registers. These are the red doubling edges and the additional point nodes in the graph of Figure 2.13.

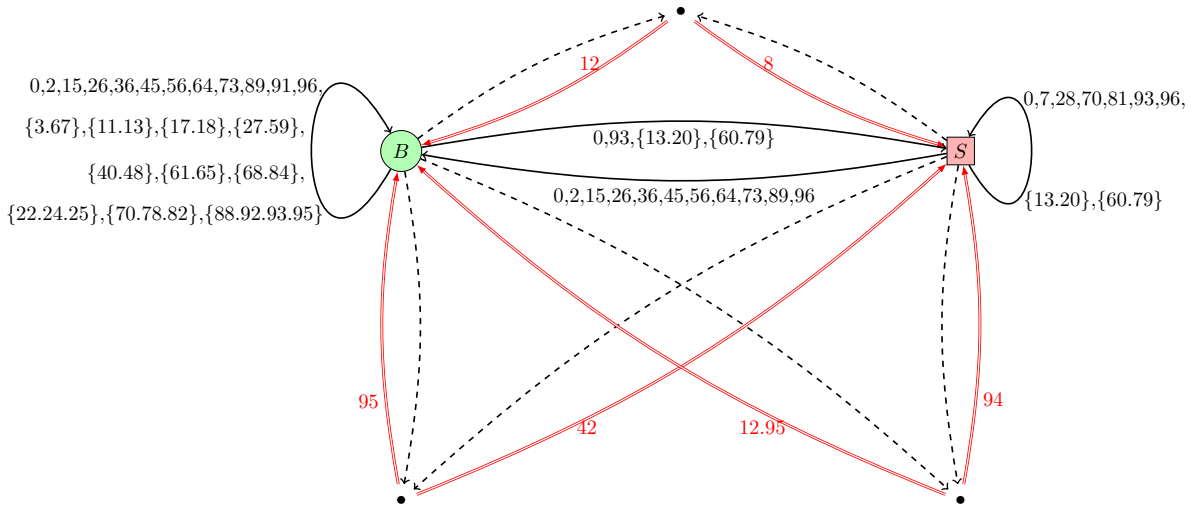


Figure 2.13 – GRAIN DFA

On this automaton, we can see several properties. First of all, some of TRIVIUM's doubling patterns also hold for GRAIN. These are the looping pattern and the simple cycle pattern. These patterns seem to be on every steam cipher with looping edges and a cycle-shaped automaton. However, there must be much complex doubling patterns. For example, we can see that there are several edges with common values that can be taken on both registers, namely: $0, 93, \{13.20\}, \{60.79\}, 2, 15, 26, 36, 45, 56, 64, 73, 89, 96,$ combined

with the doubling edges available from both registers we can see patterns like the one in Figure 2.14. However, there might be some pattern overlaps, as we have seen on TRIVIUM.

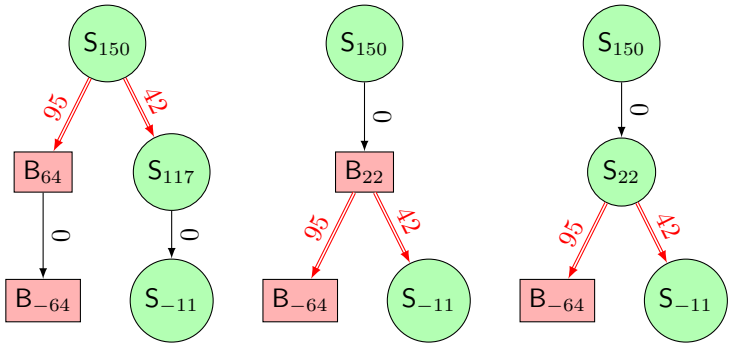


Figure 2.14 – Example of doubling pattern of GRAIN

Chapter 3

Diffusion analysis on Feistel ciphers

Introduction

The Feistel structure is an old but non-aging idea to design symmetric block ciphers. The NIST Lightweight Cryptography competition had several propositions featuring Feistel structures (ACE [AAG+19], FlexAEAD [dNX19], LOTUS [CDJ+19], Oribatida [BLLN21], Saturnin [CDL+20], SPARKLE [BBdS+19], SPIX [AGH+19]) and Generalized Feistel inspired structures (CiliPadi [ZJR+19], lilliput [BFMT15], SKINNY [BJK+16], SpoC [AGH+], TGIF [IKM+19]). In block ciphers, the diffusion is the notion of one block influencing the others. Indeed, if each output block is influenced by all the input blocks, then two similar plaintexts will have completely different ciphertexts. To design a secure block cipher, we want the diffusion to be as fast as possible. The diffusion in the Generalized Feistel Network is an ongoing research topic that lately focuses on even-odd permutations. In this chapter, we first recall the previous representations and strategies to build GFN with optimal full diffusion. Then, we present our new representations, properties and strategies to find optimal diffusion in the general case.

Takeaway

This contribution supports the following conclusions:

— GFN can be represented with graphs on which we can find useful properties.

— The optimal diffusion in the general case of GFN is not strictly better than the previously studied even-odd case for up to 32 blocks.

— New criteria might be needed to find GFN permutations with both good diffusion and good differential characteristics.

3.1 Background

The Feistel network is a generic symmetric cipher structure that was originally proposed by Horst Feistel and his colleagues at IBM in the early 70s [Smi71]. To design a cipher with a Feistel network, one needs only to choose a function F and put it in the Feistel network. The network is composed of several identical rounds. Each round will use the F function with a round sub-key deduced from the key of the cipher. A Feistel network round is shown in Figure 3.1. The plaintext X is split into two blocks (X_0, X_1).

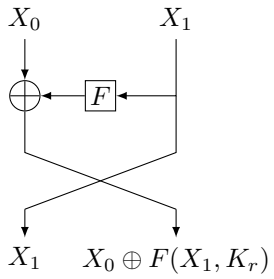


Figure 3.1 – Feistel Network round

These blocks have the same length in most of the cases but the F function also allows for unbalanced ciphers like in [BS94] or for more recent works in [AGP+19]. In both cases, the block X_1 passes through the function F that may depend on a round key K_r and is then XORed with X_0.

Definition 3.1 (Feistel Scheme) *Let $X \in \mathbb{F}_2^{s+t}$ be the plaintext split into two blocks $(X_0 \in \mathbb{F}_2^s, X_1 \in \mathbb{F}_2^t)$. Let $K_0, K_1, \ldots, K_n$ be $n+1$ sub-keys in $\mathbb{F}_2^u$ and let F be a function*

from $\mathbb{F}_2^{u+t}$ *to* $\mathbb{F}_2^s$. *For each round* $r \in \{0, 1, \ldots, n\}$ *the round function of a Feistel network is :*

$$(X_0, X_1) \rightarrow (X_1, X_0 \oplus F(X_1, K_r))$$

Feistel network-based ciphers have the really interesting property that the decryption of the ciphertext is only made by running the encryption algorithm with the round keys in reverse order. This also means that the F function of a Feistel network does not need to be invertible. Moreover, if this function is pseudorandom, then a Feistel network with at least 4 rounds is indistinguishable from a pseudorandom permutation [LR88, Pie90, PRS02]. Because of this nice property, many Feistel ciphers have been designed and some of them are still used today. The first cipher using this structure was Lucifer in 1971 [Smi71, Fei73]. It was proposed to the National Bureau of Standards (ancestor of the NIST). After some modifications, it became the Data Encryption Standard (DES [S+99]) and was used for all unclassified data encryption *i.e.,* for all non-military uses. DES was used from 1977 to 1999. It was deprecated in 1999. The size of the key (56 bits) was judged too small to resist the increasing computer performances [DH77, HC99] and the public discovery of differential cryptanalysis [BS90] and linear cryptanalysis [Mat93, Mat94]. In 1999, Triple DES (3DES [DH77]) was recommended by the NIST as a replacement. 3DES is simply three DES ciphers one after another. There are different keying options but even with three independent keys, 3DES was shown to have only 112 bits of security against meet-in-the-middle attacks. This is why we use it with only two keys, one for the middle DES and one for the first and last DES. In 2016, the sweet32 attack [1] showed that block collision attacks on 64-bits block ciphers are practical [BL16]. Thus, 3DES was deprecated by the NIST in 2017. The long life of DES and 3DES as standards shows the robustness of the Feistel network designs. The only thing left to choose is a F function and there are many different designs because a practical perfect pseudorandom function does not exists. Indeed, the F function can be composed of several operators that may vary a lot from cipher to cipher. For example the F function of DES first XOR the block with the round key, then applies in parallel 8 S-Boxes from 6 to 4 bits and then applies a permutation. It is a rather complex function that allows the cipher to have only 16 rounds to be secure. In contrary, the SIMON [BSS+13] cipher (also based on a Feistel network) uses an F function that only uses bit shifts, logical ANDs and XORs. Because of this simple F function, more rounds are needed. DES has 16 rounds and SIMON family members have 32 to 72 rounds depending on the block and key sizes. The function schemes are shown side by side in

1. https://sweet32.info/

Figure 3.2.

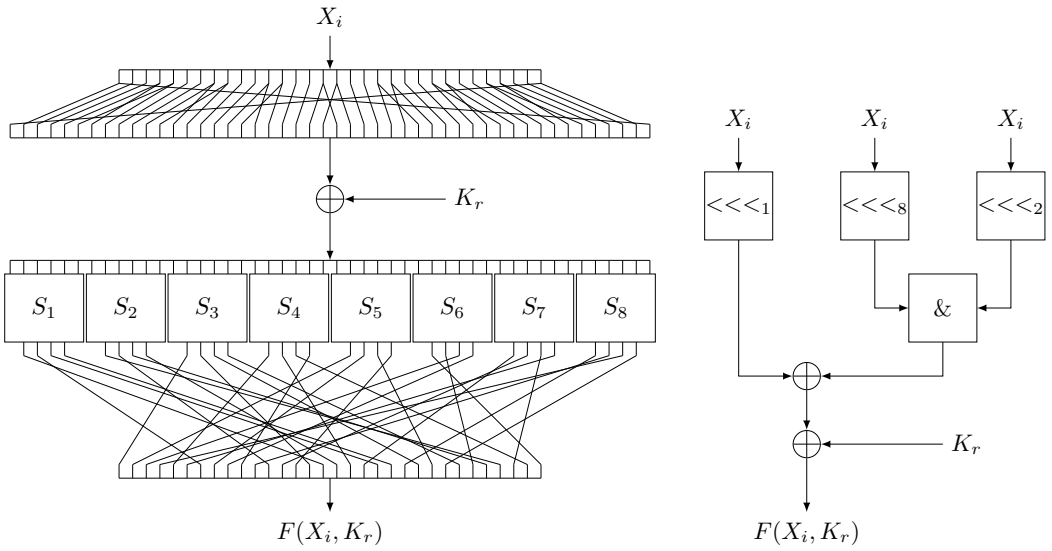


Figure 3.2 – F function of DES and SIMON

Three generalizations of the Feistel scheme were proposed and studied in [ZMI89], namely Type-1, Type-2 and Type-3 Feistel transformations. They can be found respectively in Figures 3.3, 3.4 and 3.5. The authors concluded that only the Type-2 Feistel transformation is "optimal" because they proved that it was the only transformation with the following property: If the F functions are independent pseudorandom functions, and $2k$ is the number of blocks, then a $(2k+1)$-round Type-2 Feistel cipher is indistinguishable from a pseudorandom permutation. Therefore, most of the generalized Feistel ciphers are of Type-2 *e.g.,* HIGHT [HSH+06], CLEFIA [SSA+07], ... However, there are some examples of ciphers using Type-1 Feistel (CAST 256 [AG99]) and Type-3 Feistel (MARS [BCD+98]). Here, we will focus on the Type-2 ciphers. The Type-2 Feistel network is essentially a parallel application of k Feistel networks followed by a cycle shift on all the blocks. Later, it was upgraded into Type-2 Generalized Feistel Network (GFN) in [Nyb96]. In this scheme, the cycle shift is replaced by a permutation π as seen in Figure 3.6 and Definition 3.2.

Definition 3.2 (Type-2 Generalized Feistel Network) *Let k be the number of Feistel pairs, n the number of rounds, m a word size, π a permutation over $2k$ elements and $k \times n$ keyed functions F_j^i from $\mathbb{F}_2^m$ to $\mathbb{F}_2^m$ (with $1 \leq i \leq n$, and $1 \leq j \leq k$). The ciphertext*

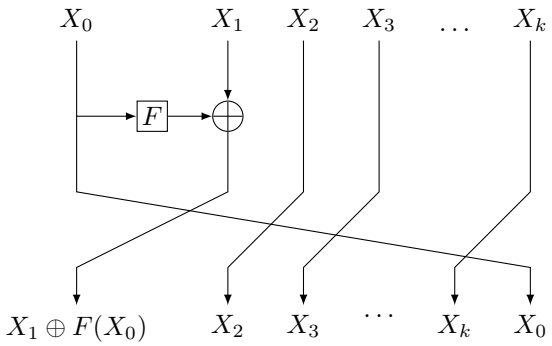


Figure 3.3 – Type-1 Feistel transformation

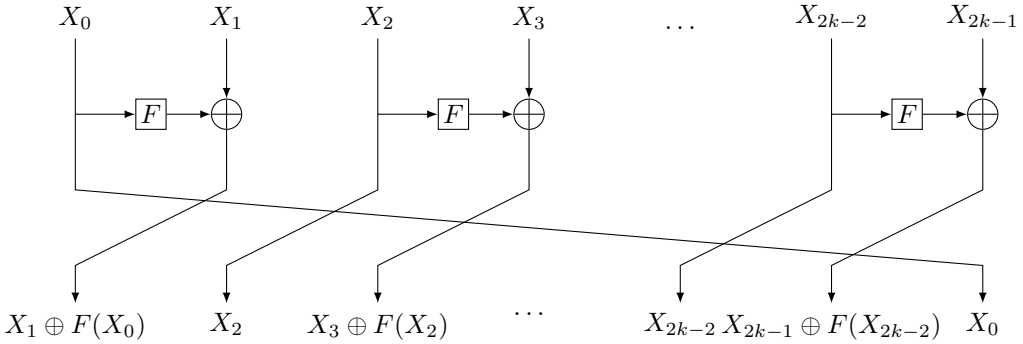


Figure 3.4 – Type-2 Feistel transformation

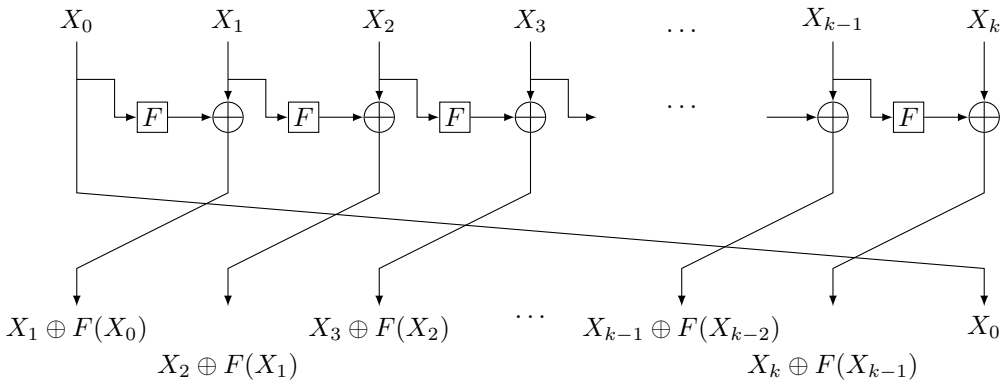


Figure 3.5 – Type-3 Feistel transformation

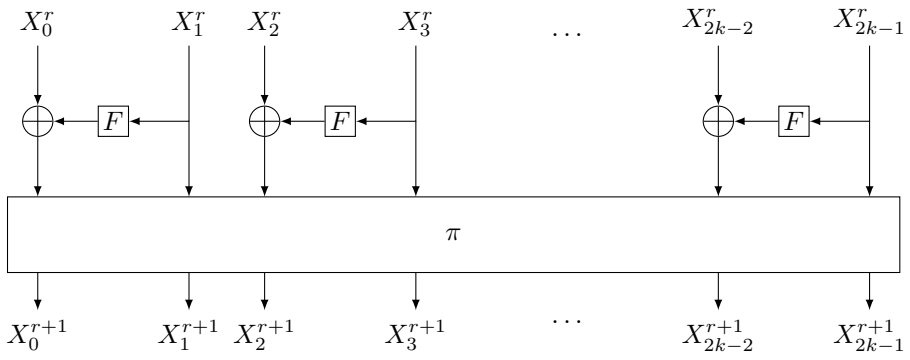


Figure 3.6 – GFN round function

of a message of size $2k \cdot m$ *is given by* $\mathcal{R}_1 \circ \ldots \circ \mathcal{R}_n$*, where* $\mathcal{R}_i$ *is the round function:*

$$\mathcal{R}_i : (X_0, \ldots, X_{2k-1}) \rightarrow \pi(X_0 \oplus F_1^i(X_1), X_1, \ldots, X_{2k-2} \oplus F_k^i(x_{2k-1}), X_{2k-1})$$

The advantage of a GFN over a Type-2 Feistel with a simple cycle shift is that it can use this permutation to be able to shuffle all the blocks faster than the cycle shift. The shuffling efficiency can be measured with the notion of diffusion round first introduced in [SM10]. Intuitively, it is the number of rounds needed such that each output block depends on each input block. A diffusion comparison is given in Figure 3.7. The left GFN uses a chosen permutation and the right one uses the cycle shift. We can see that the GFN with the cycle shift needs much more rounds to fully diffuse the first block on all the blocks. This means that for the same number of rounds, a GFN with a good permutation will be more shuffled. Moreover, if the diffusion round can be lower, then we can reduce the number of rounds of a GFN without affecting security. The diffusion round was first introduced in [SM10] and its definition is recalled below (Definition 3.3).

Definition 3.3 (Diffusion Round) *Given a GFN with a permutation* π *over* $2k$ *elements,* $DR_i(\pi)$ *is the minimum number of rounds* r *such that* X_i^0 *is diffused to all output blocks. Then, the* diffusion round *of a permutation* π *is given by* $DR(\pi) = \max_{0 \leq i < 2k}\{DR_i(\pi)\}$.

The GFN scheme has been used in many recent ciphers (LBlock [WZ11], Piccolo [SIH+11], TWINE [SMMK12], WARP [BBI+20]). These ciphers have also in common to be lightweight ciphers, meaning they need security but also efficiency. Therefore the number of rounds has to be as low as possible. To lower the number of rounds of GFN ciphers, permutations with low diffusion round are needed. This is why the search for permutations with the

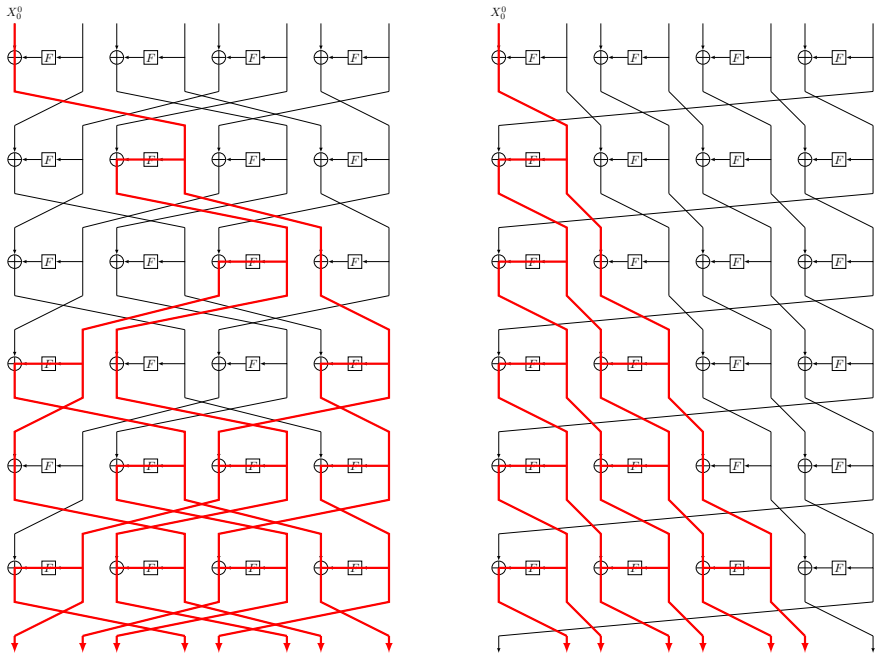


Figure 3.7 – Full Diffusion in a GFN and a Type-2 Feistel

optimal diffusion round is an important research topic. Note that the decryption of a GFN is the GFN with keys in reverse order and inverse permutation. As a consequence, the diffusion round of the inverse GFN with the inverse permutation π^{-1} is also important. Indeed, a GFN with a good diffusion round needs to have at least the same diffusion round for the inverse GFN. Another way to decrease the diffusion round is to change the permutation at every round. For example, this was studied in [AGP+19] for secure multi-party computation (each person of a group has a secret and they want to compute a function together). In this specific context it was shown that changing the permutation at each round of a Feistel was a good trade-off between diffusion and complexity of the cipher. They also observed that unbalanced Feistel networks are more suited in this context than the most widely used balanced Feistel network.

3.2 Related work on diffusion in GFNs

To study diffusion in GFNs, the authors of [SM10] used an exhaustive search to find all the permutations with the best diffusion round from 2 to 8 pairs of blocks. In all the sets of best permutations, they observed that there is always at least one permutation

with the even-odd property.

Definition 3.4 (even-odd permutation) *Let π be a permutation over $2k$ elements. An even-odd permutation is a permutation that sends each even-indexed element on an odd-indexed element and conversely i.e., $\forall i \in [\![1, k]\!]\ \exists j \in [\![1, k]\!]\ \pi(2i) = 2j + 1$ and $\forall i \in [\![1, k]\!]\ \exists j \in [\![1, k]\!]\ \pi(2i + 1) = 2j$.*

The even-odd property is very interesting in several ways. Firstly, it brings a lower bound for the diffusion round, given by the Fibonacci sequence [CGT19, DFLM19].

Proof. In a GFN, each odd-indexed block X_i (i an odd integer) diffuses to both the blocks $X_{\pi(i)}$ and $X_{\pi(i-1)}$ at the next round. Therefore, if π is an even-odd permutation and $F(X_i)$ is the maximum number of nodes we can reach in the diffusion of X_i at a round R, we have that $F(X_i) = F(X_{\pi(i-1)}) + F(X_{\pi(\pi(i))})$. Since $\pi(i-1)$ and $\pi(\pi(i))$ are odd, they have the same diffusion as $F(X_i)$ with respectively one and two rounds less. Thus if we write $F(X_i)$ as F_n (with n the number of round), we have that $F_n = F_{n-1} + F_{n-2}$. □

Thanks to this, we can compute a lower bound on the number of rounds needed to have full diffusion. It is the smallest integer i such that the Fibonacci sequence at index $i-1$ is greater than the number of pairs of blocks k ($F_{(i-1)} \geq k$). Although the optimal diffusion round is not always equal to this lower bound, for all the current known optimal diffusion round they are at most one round away from the bound. The bounds are given in Table 3.1.

k	1	2	3	4	5	6-8	9-13	14-21	22-34	35-55	56-64
Fibonacci lower bound	2	4	5	6	6	7	8	9	10	11	12

Table 3.1 – Fibonacci lower bound of the DR

The second advantage of even-odd permutations is that there are far fewer of them. Indeed, for $2k$ Feistel branches, there are $(2k)!$ possible permutations but only $(k!)^2$ even-odd ones. This reduced search space makes the search of good permutations easier.

In [SM10], the exhaustive search for optimal permutations is limited to 8 pairs of blocks but the authors presented a method to find good even-odd permutations for a higher number of pairs. This method comes with the following drawbacks: we do not know if the permutations are optimal, and the number of blocks is limited to powers of two. This method uses a graph representation of the even-odd permutations. The GFN

is represented as a directed graph where each node represents a pair of blocks and each edge represents a permutation transition. This compact representation allowed them to find a family of even-odd permutations with a diffusion round of $2log_2(2k)$ for GFN with $k = 2^s$ (s an integer $s \geq 2$). The method uses Colored De Bruijn graphs. These directed graphs have node labels in $\mathbb{F}_2^n$ and the specific property that for each edge, the next node has the same label with the first bit forgotten and one new bit at the end. For example a node labelled 10 is connected to the nodes 00 and 01 (the De Bruijn graph with $s = 2$ is given in Figure 3.8).

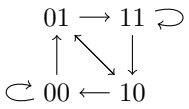


Figure 3.8 – De Bruijn graph for s=2

Designing GFN with De Bruijn graphs structure helped the authors of [SM10] to find some good permutations regarding the diffusion round. Indeed, the diffusion rounds they obtained are quite close to the lower bound. However, in the context of truncated differential analysis, the De Bruijn graph structure makes the cipher quite weak. Indeed, in truncated differential analysis, we are interested in propagating differences into a cipher. If an input difference is efficiently diffused in the cipher, it will go through more S-Boxes, making a differential attack harder. However, if one difference is XORed to the same difference, they are cancelled and this reduces the diffusion. In the context of De Bruijn graphs, each two nodes with the same name except the first bit ($a|x$ and $\overline{a}|x$) will end up in a difference cancellation. Indeed, each node of the De Bruijn graph represents a Feistel pair of blocks and the four outgoing edges of these two pairs will end up on the same two pairs ($x|1$ and $x|0$) (See Figure 3.9). This makes difference cancellation common in GFN permutations generated from a De Bruijn graph. To see this in practice, we can perform truncated differential analysis on all the GFN with $2k = 16$ blocks because there are not too much of them. Then, we can compare the optimal number of active S-Boxes to the number of active S-Boxes of the permutations deduced from the De Bruijn graphs (Table 3.2). In this table, we can see that none of these GFNs have the optimal number of active S-Boxes after round 11.

Search strategies. To find the optimal permutations, the exhaustive search used by [SM10] is not practical as soon as we want to search for permutations with more than 16

Round	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Optimal	0	1	2	3	4	6	8	11	14	19	22	26	29	31	34	37
De Bruijn	0	1	2	3	4	6	8	10	12	12	14	16	16	18	20	20
	0	1	2	3	4	6	8	10	12	14	16	20	24	30	32	34
	0	1	2	3	4	6	8	10	12	14	16	20	22	26	30	31
	0	1	2	3	4	6	8	11	14	19	21	24	25	27	30	31
	0	1	2	3	4	6	8	11	14	19	21	24	25	27	30	31
	0	1	2	3	4	6	8	11	14	18	22	24	27	30	32	35
	0	1	2	3	4	6	8	10	12	14	16	20	24	30	32	34
	0	1	2	3	4	6	8	10	12	14	16	20	22	26	30	31
	0	1	2	3	4	6	8	10	12	14	16	20	22	26	30	31
	0	1	2	3	4	6	8	10	12	14	16	20	24	30	32	34
	0	1	2	3	4	6	8	10	12	12	14	16	16	18	20	20

Table 3.2 – Best number of active S-Boxes for De Bruijn GFN

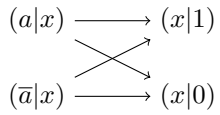


Figure 3.9 – Double cancellation in De Bruijn graph

blocks. To go further, [CGT19] first defined equivalence classes of Feistel permutations. Indeed, the pairs of blocks of a permutation π can be re-indexed to give a new permutation with the same diffusion properties. We can find a permutation ϕ such that $\pi' = \phi^{-1} \circ \pi \circ \phi$. The specificity of Feistel permutations is that the blocks are in pairs due to the transition with the S-Box from each odd to each even node. Thus the equivalence classes are pair-equivalence classes and the permutation ϕ needs to keep the two blocks of a pair together. The number of pair-equivalence classes compared to the number of permutations for general and even-odd cases are shown in Table 3.3 and Table 3.4.

$2k$	number of classes	number of solutions $(2k)!$
4	16	24
6	134	720
8	1796	40320

Table 3.3 – Number of pair-equivalence classes in the general case

There are far less pair-equivalence classes than possible permutations. The main idea is then to find a way to compute efficiently only one representative of each class. To do so [CGT19] gives two strategies; one for the general case and another one for the even-odd

$2k$	number of classes	number of solutions $(k!)^2$
4	4	4
6	11	36
8	43	576
10	161	14400
12	901	518400
14	5579	25401600

Table 3.4 – Number of pair-equivalence classes in the even-odd case

case. Note that a permutation can be categorized by its cycle decomposition a.k.a the number and length of cycles it contains.

Example 3.1 *For example, the permutation* $\pi = (1\ 0\ 3\ 2\ 4\ 6\ 7\ 5)$ *has* 4 *cycles that can be represented by* $(1\ 0)(3\ 2)(4)(5\ 6\ 7)$. *The cycle decomposition of* π *can be written only with number of cycles and their length: one cycle of length one* $(1, 1)$ *two cycles of length two* $(2, 2)$ *and one cycle of length three* $(3, 1)$. *Any permutation with the same cycles will have the same decomposition for example* $\phi = (1\ 0\ 2\ 4\ 5\ 3\ 7\ 6)$ *also has a cycle decomposition in* $(1, 1), (2, 2), (3, 1)$. *Another way to write a cycle decomposition is to give a list of the cycle lengths like* $(3, 2, 2, 1)$ *for our example.*

The number of cycle decompositions of a permutation of length k is the number of partitions of the integer k noted $\mathcal{N}_k$.

The strategies presented in [CGT19] first takes only one representative of each cycle decomposition to reduce the permutation enumeration. The first strategy focuses on even-odd permutations.

1. The first step sets one cycle decomposition representative of half of the permutation:
   For each cycle decomposition type t *of size* k, *set an arbitrary permutation* ϕ_t *that satisfies this decomposition type.*
2. The second step enumerates the second half of the permutation:
   For all permutations θ *of* k *elements, construct the permutation* $\pi_{\phi_t,\theta}$ *given by*

$$\begin{cases} \pi_{\phi_t,\theta}(2j+1) = 2\phi_t(j) \\ \pi_{\phi_t,\theta}(2j) = 2\theta(j) + 1 \end{cases}$$

3. The last step computes the diffusion round of this permutation:
   Compute the diffusion of $\pi_{\phi_t,\theta}$

With this strategy, the number of permutations on which we have to compute the diffusion round on is reduced to $\mathcal{N}_k \times k!$.

In the general case, the strategy becomes more complex.

1. The first step takes one cycle decomposition representative of the whole permutation.
   For each cycle decomposition type t of size $2k$, set an arbitrary permutation g_t that satisfies this decomposition type.
2. The second step chooses k sorted elements among $2k$ to represent the first half of the permutation and put them on the odd indexes.
   For each possible set $\{x_1, \ldots, x_k\}$ of k elements among $2k$ assumed sorted, set an arbitrary permutation ϕ_a such that $\phi_a(x_i) = 2i + 1$.
3. The third step enumerates the second half of the permutation.
   For all permutations θ of k elements, construct the permutation π given by

$$\begin{cases} \pi(2j+1) = 2j+1 \\ \pi(2j) = 2\theta(j) \end{cases}$$

4. The last step computes the diffusion of everything together.
   Compute the diffusion of $(\pi \circ \phi_a)^{-1} \circ g_t \circ (\pi \circ \phi_a)$

With this strategy, the number of permutations on which we have to compute the diffusion round is reduced to $\mathcal{N}_{2k} \times \frac{(2k)!}{k!}$. The number of diffusion round to test can be further reduced by removing the cycles decompositions with non diffusing cycles such as 1-cycles or subsets of cycles which only diffuse to themselves.

With these strategies, the optimal permutations were found for 18 pairs in the general case, and for 24 pairs on the even-odd case. As in the previous works, the authors of [CGT19] then proposed a heuristic to build good even-odd permutations. This method tries to build permutations with a low number of collisions in the first rounds of the diffusion. A collision occurs when two branches join in the diffusion tree (see Figure 3.7). The belief that low collision permutations may lead to better diffusion rounds comes from the cases $k = 8$ and $k = 13$ pairs. In these cases of GFN, there is a low number of solutions with optimal DR and they all have as few collisions as possible in their diffusion tree. However, in other cases, there are some permutations with optimal diffusion round that do not have this property. The heuristic takes as input an integer r as a small number of rounds on which no collision is allowed. The even-odd strategy is then adapted in step

2 to only enumerate the permutations that satisfy this property. This heuristic has the complexity of $O(2^{\frac{k}{4}})$ which allows them to compute good permutations for 32, 64 and 128 blocks. In this paper, they also open a discussion on a lower bound of the diffusion round on the general case which we will discuss later on.

Search strategies for even-odd. In the paper [DFLM19] the focus was made on even-odd permutations. On this case, the authors highlighted the following property.

Corollary 3.1 *Let $\pi = (p, q)$ be an even-odd permutation over 2k elements. $DR(\pi) = r$ if, and only if, each even block X^0_{2j} diffuses to at least one block of each pair of the $(r-1)$-th round $X^{(r-1)}_{2j'}$ or $X^{(r-1)}_{2j'+1}$.*

By separating the even-odd permutation π into two permutations p and q for even blocks and odd blocks, they were able to define the diffusion set of indexes and to compute the diffusion round of a permutation faster than previous methods. For example the diffusion set of indexes for 6 rounds is given by

$$\mathbb{J}^6_j = \{(p^4)(j), (qp^3)(j), (pqp^2)(j), (p^2qp)(j), (qpqp)(j), (p^3q)(j), (qp^2q)(j), (pqpq)(j)\}$$

They are interested in the diffusion of the even nodes only (Corollary 3.1) so the first common permutation p is omitted to simplify the diffusion computation.

They will also exploit the fact that if p is known, $\mathbb{J}^6_j$ can be computed only with 7 guesses over the image of q namely:

$$q(j), (qp)(j), (qp^2)(j), (qp^3)(j)(qpq)(j), (qp^2q)(j), (qpqp)(j)$$

Moreover, they found that if we separate $\mathbb{J}^6_j$ into two sets

$$\mathbb{X}^6_j = \{(qp^3)(j), (pqp^2)(j), (p^2qp)(j), (qpqp)(j), (p^3q)(j), (qp^2q)(j), (pqpq)(j)\}$$

and

$$\mathbb{Y}^6_j = \{(qp^3)(j), (qp^2q)(j), (pqpq)(j)\}$$

one can express $\mathbb{J}^8_j$ as

$$\mathbb{J}^8_j = p^2(\mathbb{X}^6_j \cup \mathbb{Y}^6_j) \cup (pq)(\mathbb{X}^6_j) \cup (qp)(\mathbb{X}^6_j \cup \mathbb{Y}^6_j)$$

With these properties, [DFLM19] presented an efficient algorithm to find all the even-odd permutations with a diffusion round of 9. The first step of their search is similar to the first steps of [CGT19]. They set half of the permutation (namely p) with one representative of each cycle decomposition of size k. The search exploits all the previous properties in a branch-and-bound algorithm optimized to perform as few guesses as possible on q.

1. For each cycle decomposition type t of size k, set an arbitrary permutation p that satisfies this decomposition type.
2. Take a first element j of the smallest cycle
3. Guess $\mathbb{J}_j^6$ (7 guesses) and add a constraint

$$|p^2(\mathbb{X}_j^6 \cup \mathbb{Y}_j^6) \cup (pq)(\mathbb{X}_j^6) \cup (qp)(\mathbb{X}_j^6 \cup \mathbb{Y}_j^6)| \geq k$$

   (this constraint enforces that at least all blocks can be reached from j in 9 rounds)
4. While the constraints are all valid and q is not entirely found, continue to guess on $p(j)$ (3 guesses) to add constraints. If $p(j)$ is already guessed, go to a new j on step 2. If some constraints are incompatible, backtrack to change the previous guess.

With this efficient algorithm, the authors of [DFLM19] were able to find all the optimal even-odd permutations for GFN with $26, 28, 30, 32$, and 36 blocks. The algorithm found no even-odd permutations of diffusion round of 9 for GFN with $34, 38, 40$, and 42 blocks.

The authors also performed a security analysis for each permutation found and in particular, differential analysis. They found that the truncated differential characteristics vary a lot between two permutations that have optimal diffusion round. For $k = 16$, they found between 26 and up to 40 active S-Boxes whereas some non optimal permutations with regard to the diffusion round may have 70 active S-Boxes.

One of the main question left open in the literature remains to be whether a non-even-odd permutation can achieve better diffusion than the well studied even-odd permutations. Indeed, we have no proof that the optimal even-odd permutations are actually optimal permutations in the general case beyond $k = 10$ pairs of blocks. The authors of [CGT19] also raised the question of a lower bound for the general case. Moreover, the results in [DFLM19] tend to question if the diffusion round is a good metric for differential analysis resistance. In the next section, we present new representations of GFN to hopefully answer these three questions.

3.3 New representations

The first goal of our work was to find a representation for the diffusion in the GFN with non-even-odd permutations (Definition 3.5).

Definition 3.5 (non-even-odd permutation) *Let π be a permutation over $2k$ elements. A non-even-odd permutation is a permutation that sends at least one even-indexed element on an even-indexed element.*

Before finding a good representation, we tried several models described in the following sections. In graph theory, both walk and path define a sequence of edges which joins a sequence of vertices but the walk allows for redundant edges and vertices. Since we do not need to know whether the edges are redundant or not for the diffusion round, this section will use the term path for any sequence of edges which joins a sequence of vertices even if it is not graph theory terminology.

3.3.1 Boolean matrix product

A permutation π can be described as an adjacency matrix A_π with exactly one 1 in each line and column. For example if $\pi = (1, 4, 0, 2)$ then:

$$A_\pi = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

The GFN is a permutation with some additional transitions from each odd block to its even block. To fully represent a GFN in an adjacency matrix, one needs to add to each odd line a 1 in the same column as in the previous even line. For example, we add the ones (in red) and B is obtained from A_π:

$$B = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & \color{red}{1} & 0 & 1 \\ 1 & 0 & 0 & 0 \\ \color{red}{1} & 0 & 1 & 0 \end{pmatrix}$$

Then, we can use Property 3.1 to check the diffusion round.

Property 3.1 *An adjacency matrix A to the power of n contains in each cell $A^n_{i,j}$ the number of paths from i to j.*

Therefore, we have the following results:

Corollary 3.2 *Let π be a GFN permutation over $2k$ elements. $DR(\pi) = r$ if, and only if, r is the smallest integer such that the adjacency matrix of the GFN to the power of r contains only non-zero integers.*

If we compute powers of the example matrix B, we can see that the first one to have no 0 is B^4 so the diffusion round is 4.

$$B^3 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 2 & 0 & 1 \end{pmatrix} \quad B^4 = \begin{pmatrix} 2 & 2 & 1 & 1 \\ 2 & 4 & 1 & 2 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 1 & 2 \end{pmatrix} \quad B^5 = \begin{pmatrix} 2 & 4 & 1 & 2 \\ 3 & 6 & 2 & 4 \\ 2 & 2 & 1 & 1 \\ 3 & 3 & 2 & 2 \end{pmatrix}$$

The diffusion round property is obtained if there is at least one path between each pair of blocks. As a consequence, we can replace the integer matrices by Boolean matrices where a true Boolean indicates if there is at least one path. Furthermore, the matrix product is simplified. Indeed, we can replace additions and multiplications by disjunctions and conjunctions. We also use fast exponentiation *i.e.,* instead of computing B^8 with seven matrix products ($B \times B \times B \times B \times B \times B \times B \times B$) we can perform three products and compute only $B2$, $B4$, and $B8$ ($B2 = B \times B, B4 = B2 \times B2$ and $B8 = B4 \times B4$).

To test this representation, we modelled it in MiniZinc [NSB+07]. MiniZinc is an expressive CP modelling language which is suitable for modelling problems for a range of solvers. We made a model using the matrix representation to search for non-even-odd permutations with a given *DR* for GFN. The model is composed of the following constraints:

A is the adjacency matrix of π, each line and column must have only one true Boolean:

$$\sum_{j \in [\![0,2k-1]\!]} A_{i,j} = 1 \quad \forall i \in [\![0, 2k-1]\!]$$
$$\sum_{i \in [\![0,2k-1]\!]} A_{i,j} = 1 \quad \forall j \in [\![0, 2k-1]\!]$$

The full diffusion is impossible if a block is only sent on itself therefore 1-cycles are constrained by:

$$A_{i,i} = 0 \quad \forall i \in [\![0, 2k-1]\!]$$

The full diffusion is also impossible if the two blocks of a pair are sent on each other therefore they are constrained by:

$$A_{2i,2i} \; + \; A_{2i,2i+1} \; + \; A_{2i+1,2i} + A_{2i+1,2i+1} \geq 1 \quad \forall i \in [\![0, k-1]\!]$$

The GFN permutation B is constructed from A with the additional 1:

$$\begin{aligned} B_{2i,j} &= A_{2i,j} \quad \forall i \in [\![0, k-1]\!] \; \forall j \in [\![0, 2k-1]\!] \\ B_{2i+1,j} &= A_{2i,j} \vee A_{2i+1,j} \quad \forall i \in [\![0, k-1]\!] \; \forall j \in [\![0, 2k-1]\!] \end{aligned}$$

B^r is constrained with fast exponentiation and Boolean matrix products like:

$$B2_{i,j} = B_{i,k} \wedge B_{k,j} \quad \forall i, j \in [\![0, 2k-1]\!] \; \exists k \in [\![0, 2k-1]\!]$$

In practice, this model was very slow paired with any solver (CP, MILP and SAT). Furthermore, many permutations were similar. This corresponds to the pair-equivalence classes described in [CGT19]. To break these symmetries and to find less redundant solutions, we tried static and dynamic symmetry breaking constraints. A static way to prevent some symmetries is to first set an order in which the variables will be chosen (from 0 to $2k$ in our case). After that, we can let only one pair of free variables available on each line. For example, in the following matrix the zeros in the top right corner are the forbidden transitions.

$$\begin{pmatrix} ? & ? & ? & ? & 0 & 0 & 0 & 0 \\ ? & ? & ? & ? & ? & ? & 0 & 0 \\ ? & ? & ? & ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? & ? & ? & ? \end{pmatrix}$$

To go in more details, the search will first try to set $p(0)$ and will only have the free pair $(2, 3)$ in addition to its own pair $(0, 1)$ to choose from. Then to set $p(1)$ only one new free pair is available $(4, 5)$, ... The problem with these static constraints is that they only constrain the small upper right triangle of the matrix as if the search will always use the new free pair but if this pair is not chosen at one step, it remains the free pair for the next step. To constrain this, we designed a dynamic version of this idea where the pair $(A_{i,2j}, A_{i,2j+1})$ is always the only free pair available in the constraint:

$$A_{i,2j} \leq \sum_{i' \in [\![0,i-1]\!]} \sum_{j' \in [\![2j-2,2k-1]\!]} A_{i',j'} \quad \forall i \in [\![2, 2k-1]\!] \; \forall j \in [\![1, k-1]\!]$$

We used this model with several solvers and we were able to recover the results presented in [CGT19]. This MiniZinc model allows to quickly identify the most efficient solver and we found the performances of the solvers interesting. The model is using only Boolean variables but Picat-SAT solver was very slow. The best solver was Chuffed followed by Or-Tools CP-SAT, and the worst one was Gurobi because of the highly combinatorial constraints of the matrix product. Unfortunately, the models based on the matrix representation were not fast enough to find solutions in reasonable time for a higher number of pairs. Moreover, we tested Chuffed with and without the clause learning and found no significant difference. Our hypothesis for this behavior is that the constraints are too small and basic (sum and logical constraints) for the explanation process to be useful.

3.3.2 Successors union set

The second representation we proposed was much closer to CP. Each element of the permutation is represented with an integer variable $P_i \; \forall i \in [\![0, 2k-1]\!]$. To model the fact that P is a permutation we declare the constraint

$$\mathit{AllDifferent}(P)$$

Then, we add set variables $S_{i,r}$ that contain all the reachable blocks after r rounds from each starting block i. On the first round, the reachable blocks are deduced from the permutation variables P_i as:

$$\begin{cases} S_{2i,1} = \{P_{2i}\} \\ S_{2i+1,1} = \{P_{2i}, P_{2i+1}\} \end{cases} \quad \forall i \in [\![0, k-1]\!]$$

For the reachable variables with a greater r they are defined by the union of all the successors of the reachable blocks at the previous round $r-1$:

$$S_{i,r} = \bigcup_{j \in S_{i,r-1}} S_{j,1} \quad \forall i \in [\![0, 2k-1]\!] \; \forall r \in [\![2, R]\!]$$

Unfortunately, there is no Union constraint with the index set as a variable in the constraint catalog or in the available solvers. To model our problem we defined the filtering algorithms of this constraint and we added it into the solver Choco. A filtering algorithm should describe all the situations where a value can be removed from each variable domain ($e \notin \overline{S}$) and all the situations where a value becomes mandatory in the domain of its variable ($e \in \underline{S}$). Let the set variables U, I and S be such that $U = \bigcup_{i \in I} S_i$

- If a value is removed from U, then it is also removed from the possible values of the mandatory sets S:
  $e \notin U, \; \forall i \in \underline{I} \implies e \notin \overline{S_i}$
- If a value is added to the mandatory set of U and if there is only one set S with this value, then it becomes mandatory:
  $e \in \underline{U}, \; \exists i \in \overline{I}, \; \forall j \in \overline{I}, j \neq i, \; e \notin \overline{S_j} \implies e \in \underline{S_i}, i \in \underline{I}$
- If an index is removed from I then all the values only supported by this set in U are removed:
  $i \notin \overline{I}, \; \forall e \in \overline{S_i}, \; \forall j \in \overline{I}, j \neq i, \; e \notin \overline{S_j} \implies e \notin \overline{U}$
- If an index becomes mandatory, then all the mandatory values of the corresponding set S becomes mandatory in the union U:
  $i \in \underline{I}, \; \forall e \in \underline{S_i} \implies e \in \underline{U}$
- If a value is removed from a set and this value is in no other sets, then it is removed from the union too:
  $e \notin \overline{S_i}, \; \forall j \in \overline{I}, j \neq i, \; e \notin \overline{S_j} \implies e \notin \overline{U}$
- If a value is mandatory in a set and this set is mandatory, then the value is mandatory in the union too:
  $e \in \underline{S_i}, \; i \in \underline{I} \implies e \in \underline{U}$

Note that some of the filtering rules can be redundant which is not a problem.

We reused the dynamic symmetry breaking constraints of the matrix representation by adapting them to the permutation integer variables P. As a result we were able to find optimal permutations for 24 blocks which is better than the best previous methods for the general case [CGT19]. However, we wanted to go further and solution we proposed to

solve the symmetry issue was not very satisfying.

3.3.3 Graph representation

The third and main representation we explored is to model GFN as directed graphs for the general case (the graph of [CGT19] was representing only even-odd permutations). We were then able to define new properties, explore some ideas towards a lower bound proof, and finally find an efficient algorithm to find all the optimal permutations for non-even-odd GFN up to 32 blocks.

Definition 3.6 (GFN Graph) *Given a permutation π over $2k$ elements, the* GFN graph *associated to π is the graph $G_\pi = (V, E)$ where:*

$V = V_{\mathsf{e}} \cup V_{\mathsf{o}}$ *with*
$V_{\mathsf{e}} = \{0, 2, \ldots, 2k-2\}$, *the even nodes*
$V_{\mathsf{o}} = \{1, 3, \ldots, 2k-1\}$, *the odd nodes*
$E = E_\epsilon \cup E_\pi$ *with*
$E_\epsilon = \{(1,0), (3,2), (5,4), \ldots, (2k-1, 2k-2)\}$, *pairwise edges*
$E_\pi = \{(u, v) \mid u, v \in V \wedge \pi(u) = v\}$, *permutation edges*

The set V is the set of all nodes which is divided into two halves, the set of even nodes V_{e} and the set of odd nodes V_{o} representing respectively the even blocks and the odd blocks of a GFN. The set E_π is the set of all the edges representing the permutation π, whereas E_ϵ is the set of edges representing the S-Box passages from the odd to the even blocks (also called ϵ-transitions). For example, the GFN graph of the permutation $\pi = (2, 4, 5, 6, 9, 11, 7, 1, 3, 12, 15, 0, 13, 14, 8, 10)$ can be found on Figure 3.10.

In the following, we will often refer to the GFN graph G_π of a permutation π. The sets $V_{\mathsf{e}}, V_{\mathsf{o}}, E_\pi, E_\epsilon$ will be used to represent the even blocks, the odd blocks, the permutation transitions and the ϵ-transitions.

To formally define the Diffusion Round on a GFN graph we need to define the following notion of path and diffusable path (also called d-path)

Definition 3.7 (Path & Diffusable Path) *A* path $p = (e_1, \ldots, e_n)$ *is a finite sequence of edges from E which joins two nodes from V. Moreover, when $e_n \in E_\pi$, such a path is called a* diffusable path *(or* d-path *for short).*

We say that a path p is of *length* ℓ if there are exactly ℓ edges from E_π in p. We consider paths were there can be multiple occurrences of the same edge, if so they will be counted

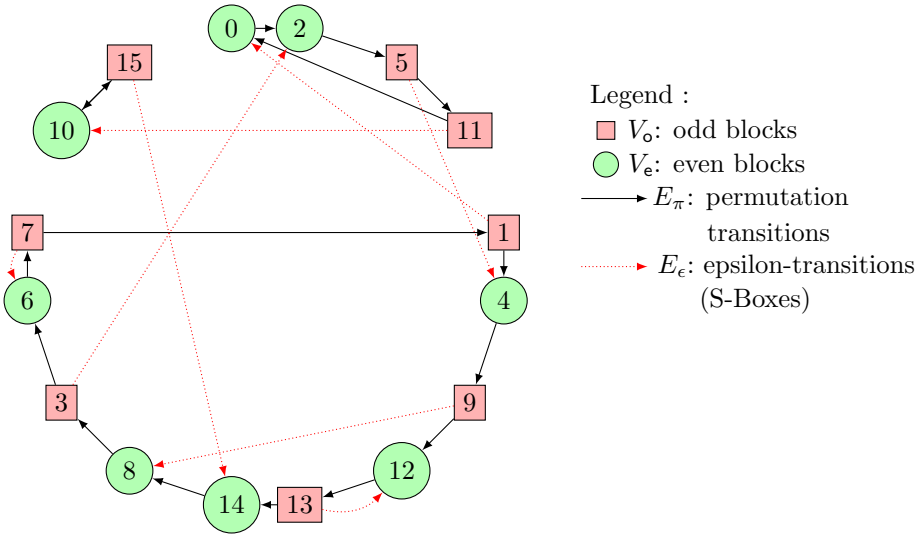


Figure 3.10 – GFN graph G_π associated to the permutation π

in the length as many times as they appear in the path. For some properties, we need to consider d-paths since a GFN round is composed of one potential edge in E_ϵ followed by one edge in E_π. This is because the GFN round always end by the permutation. Based on this graph representation, we proposed a new characterization of $DR(\pi)$.

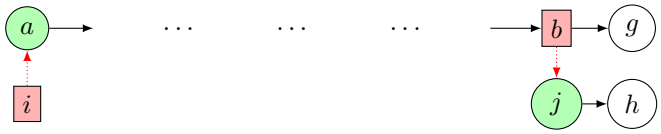
Corollary 3.3 *$DR(\pi)$ is the smallest integer R such that:*

$$\forall\ u, v \in V, \text{ there exists a } d\text{-path of length } R \text{ from } u \text{ to } v \text{ in } G_\pi.$$

In order to compute the diffusion round of a permutation π, we can consider the d-paths of a certain length between all pairs of nodes in the graph G_π. However, as already noticed by [DFLM19] and recalled in Corollary 3.1, in the specific setting of even-odd permutations, it is actually sufficient to consider some specific sets of nodes, and only paths of length $R-1$ to establish that the diffusion round is equal to R. With the GFN graph we can define a similar property for the non-even-odd case (Proposition 3.1) and we can extend it for the even-odd case (Proposition 3.2).

Proposition 3.1 *Let π be a permutation, $DR(\pi)$ is the smallest integer R such that: $\forall a \in V_e$, $\forall b \in V_o$, there exists a path of length $R-1$ from a to b in G_π.*

Proof. Let $a \in V_e$ and $b \in V_o$, we have that $(a+1, a), (b, b-1) \in E_\epsilon$ with $a+1 \in V_o$ and $b-1 \in V_e$. Furthermore, we have $g, h \in V$ such that $(b, g), (b-1, h) \in E_\pi$ (see the graph below with $i = a+1$ and $j = b-1$).



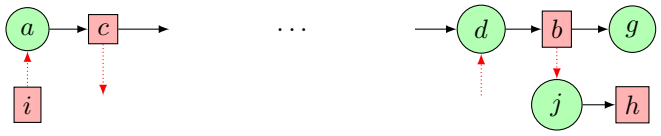
1) From Corollary 3.3, we know that there is a d-path of length R from a to g, thus there is a path of length $R-1$ from a to b.

2) Now, suppose that there is a $R' < DR(\pi)$ such that $\forall\ a \in V_{\mathsf{e}},\ b \in V_{\mathsf{o}}$ there is a path of length $R'-1$ from a to b. We then have a d-path of length R' from i to g, from i to h and from a to h. Since we have these d-paths for each pair $a \in V_{\mathsf{e}}, b \in V_{\mathsf{o}}$, we have full diffusion with R' leading to a contradiction. □

For any permutation π, the Proposition 3.1 reduces the number of paths we have to consider when studying diffusion. In the case of an even-odd permutation, the length of these paths can be further reduced.

Proposition 3.2 *Let π be an even-odd permutation, $DR(\pi)$ is the smallest integer R such that: $\forall c \in V_{\mathsf{o}}$, $\forall d \in V_{\mathsf{e}}$, there exists a path of length $R-3$ from c to d in G_π.*

Proof. Let $b, c \in V_{\mathsf{o}}$ and $a, d \in V_{\mathsf{e}}$ with $(a,c),(d,b) \in E_\pi$. We have that $(a+1,a),(b,b-1) \in E_\epsilon$ with $a+1 \in V_{\mathsf{o}}$ and $b-1 \in V_{\mathsf{e}}$. Furthermore, we have $g, h \in V$ such that $(b,g),(b-1,h) \in E_\pi$ (see the graph below with $i = a+1$ and $j = b-1$).



1) From Proposition 3.1, we know that there is a path of length $R-1$ from a to b, thus there is a path of length $R-3$ from c to d.

2) Now suppose that there is $R' < DR(\pi)$ such that $\forall\ c \in V_{\mathsf{o}},\ d \in V_{\mathsf{e}}$ there is a path of length $R'-3$ from c to d. We then have a d-path of length R' from i to g, from i to h and from a to h. Since we have these d-paths for all pairs $a \in V_{\mathsf{e}}, b \in V_{\mathsf{o}}$ then we have full diffusion with R' leading to a contradiction. □

Proposition 3.1 and 3.2 are useful for both our previous models because they reduce the search space. They are also very useful for path based algorithms. The graph representation will be further used in the next sections especially to get rid of symmetries.

3.3.4 GFN graph as automaton and regular expressions

The GFN graph defined in the previous section looks like an automaton. A finite-state automaton is composed of a finite set of states, symbols of an alphabet and transition functions. They can be represented using state diagrams like in Figure 3.11.

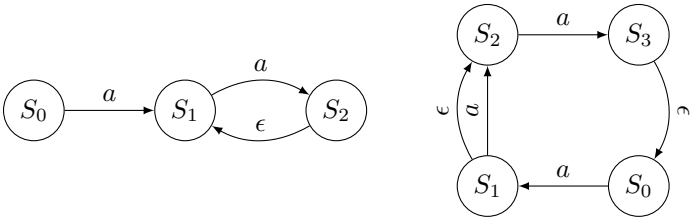


Figure 3.11 – Automaton examples

An automaton can recognise a language (set of words). On the leftmost automaton of Figure 3.11, let S_0 be the starting state and S_2 the ending state. In formal language theory, the letter ϵ represents a free transition so this automaton can recognise the words: $aa, aaa, aaaa, \ldots$. Therefore it defines the language given by the regular expression aaa^*. Moreover, if we take a look at the rightmost automaton of Figure 3.11 from state S_0 to state S_3, we can find its regular expression: $(aa|aaa)^*$. This expression is deduced from a different graph but it is the same as aaa^* after simplification. Since the two graphs have the same regular expression, this means that they contains paths of same lengths. This remark may be useful to simplify the search of graphs with optimal diffusion round. If we note the transitions of E_π with the letter a and the transitions of E_ϵ with the letter ϵ, we can give an other characterisation of the diffusion round:

Corollary 3.4 *$DR(\pi)$ is the smallest integer R such that: $\forall\, u, v \in V$, the regular expression corresponding to G_π from u to v recognise the word $a \ldots a$ (R times).*

The advantage of regular expression is that we can simplify them to detect the equivalent sub-graphs one can use to build a GFN graph. For example, if we need to link two nodes with a path of length $2, 4$ and 5, both sub-graphs of Figure 3.11 can be used interchangeably.

However, dealing with regular expressions simplifications for each pair of node while building a GFN graph is a very complicated task. It is left as an open question to see if this can be useful to improve the efficiently of a search algorithm.

3.4 New strategies

As seen in the previous approaches, the number of possible permutations of a GFN is too high to be simply enumerated. A GFN of k pairs have $(2k)!$ potential permutations (red curve of Figure 3.12) so we will need strategies to reduce the search space.

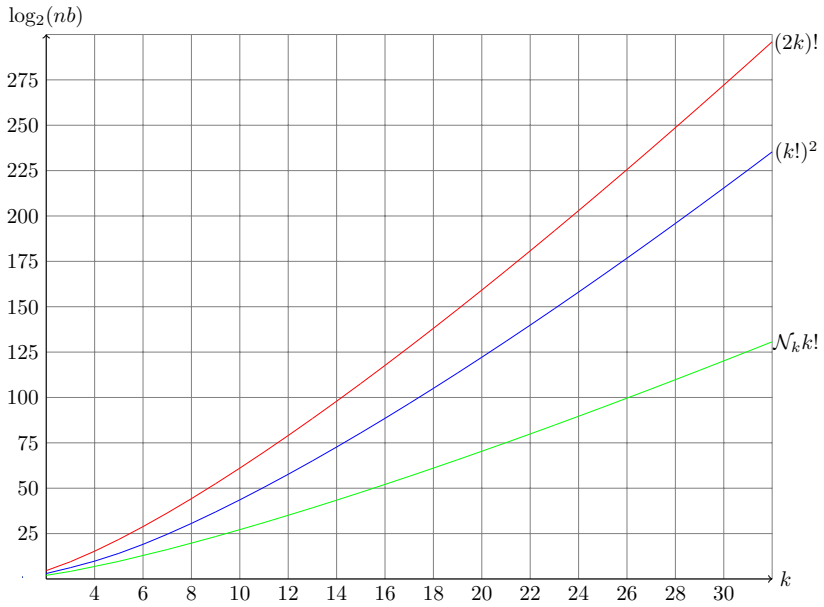


Figure 3.12 – Number of possible permutations for values of k

As explained in [CGT19], in the even-odd case, the permutation can be split in two parts. This reduces the search space to $(k!)^2$ (blue curve of Figure 3.12). Moreover, the first half of the permutation can be further reduced to all its possible cycle decompositions to break some symmetries. This reduces the search space to $\mathcal{N}_k \times k!$ where $\mathcal{N}_k$ is the number of partitions of k (green curve of Figure 3.12).

Skeletons. In this section, we propose a generalization of the cycle decompositions to consider non-even-odd permutations as well. With this structure that we call skeleton, the number of permutations in the search space is then reduced from $(2k)!$ to the curve [2] $(k!)^2$ of Figure 3.12. This skeleton will also guide the enumeration algorithm. For this generalization, we rely on our graph representation and the following definitions of cycles and chains.

2. There is no exact formula but when we count the possibilities, the curve is very close to $(k!)^2$

Definition 3.8 (ϵ-cycle) *An ϵ-cycle is a path $c = (e_1, \ldots, e_{2l})$ in which the first and last nodes are equal and edges alternate between E_π and E_ϵ one by one.*

We note a l-ϵ-cycle an ϵ-cycle of size l i.e. with l ϵ-transitions. Moreover, we will only use one representative of $c = (e_1, \ldots, e_{2l})$ and we will not consider all the equivalent ϵ-cycles like $(e_{2l}, e_1, \ldots, e_{2l-1})$ or $(e_1, \ldots, e_{2l}, e_1, \ldots, e_{2l})$. Some examples are given in Figure 3.13.

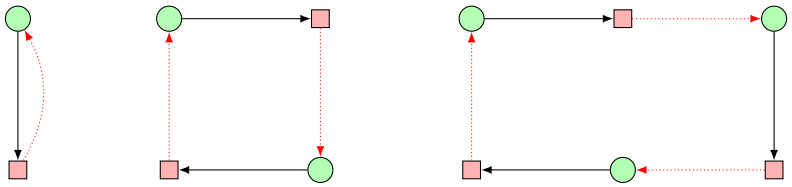


Figure 3.13 – 1-ϵ-cycle, 2-ϵ-cycle, and 3-ϵ-cycle

Let P be a partition of the integer k. For each $i \in P$, we set one representative ϵ-cycle of the corresponding size. For example, there are three possible decompositions in ϵ-cycle for $k = 3$, i.e. $\{3\}$, $\{2, 1\}$, and $\{1, 1, 1\}$. This corresponds to one 3-ϵ-cycle, or one 2-ϵ-cycle with one 1-ϵ-cycle, or three 1-ϵ-cycles. This holds only for the even-odd case because even nodes can only have edges to the odd nodes and conversely whereas even to even, and odd to odd edges are allowed in the general case. To have a similar method in the general case, we rely on ϵ-chains to handle the non-even-odd parts of the permutation.

Definition 3.9 (ϵ-chain) *An ϵ-chain is a path $c = (e_1, \ldots, e_{2l+1})$ in which the two first nodes are in V_o and the two last nodes are in V_e. The edges alternate between E_π and E_ϵ one by one.*

We note an l-ϵ-chain an ϵ-chain of size l, i.e. with l ϵ-transitions. Except for the first and the last node, all the nodes in an ϵ-chain are pairwise distinct. Indeed, if a node appears twice in an ϵ-chain, then it is not an ϵ-chain but an ϵ-cycle because each node has only one successor. However, the first and last node of the chain can be in an other ϵ-cycle or an other ϵ-chain, or in the same ϵ-chain, making the ϵ-chain loops on itself (Examples of chains and looping chains in Figure 3.14). This loop may occur at the beginning of the ϵ-chain, at its end, or on both sides.

Definition 3.10 (Skeleton) *A* skeleton *of size k is a set of ϵ-cycles and ϵ-chains whose sum of sizes is k.*

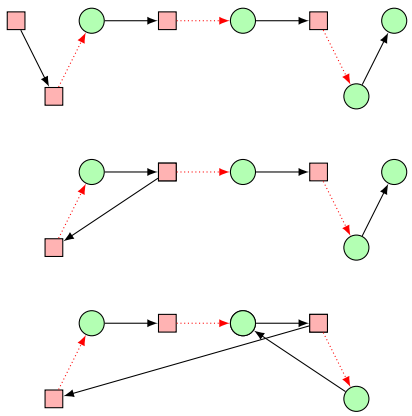


Figure 3.14 – A 3-ϵ-chain with looping examples

We will use the same notation as for cycle decomposition but with two sides, the ϵ-cycle side and the ϵ-chain side. The skeleton in Figure 3.15 is noted $(3,1,1)(2,1)$.

Example 3.2 *The skeleton of the graph given in Example 3.10 is depicted below (see Figure 3.15). It is composed of three ϵ-cycles of size* 3, 1, *and* 1, *as well as two ϵ-chains of size* 2 *and* 1.

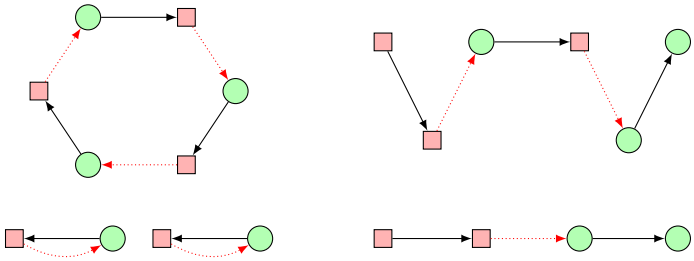


Figure 3.15 – Skeleton of Figure 3.10

The skeleton of Figure 3.15 is also valid for graphs similar to Example 3.10 but with different node names. In fact we can permute two pairs of nodes to find a different permutation having the same skeleton (some examples in Figure 3.16). This is why we will only use one representative of each skeleton.

The number of possible skeletons is given by the formula $\sum_{i=0}^{k} \mathcal{N}_i \times \mathcal{N}_{k-i}$ with $\mathcal{N}_i$ the number of partitions of the integer i. The formula has two parts, one for the ϵ-cycles with $\mathcal{N}_i$ and one for the ϵ-chains with $\mathcal{N}_{k-i}$. The formula then sums the skeletons with each possible division into ϵ-cycles and ϵ-chains. To give an idea, for $2k = 16$, there are only 22

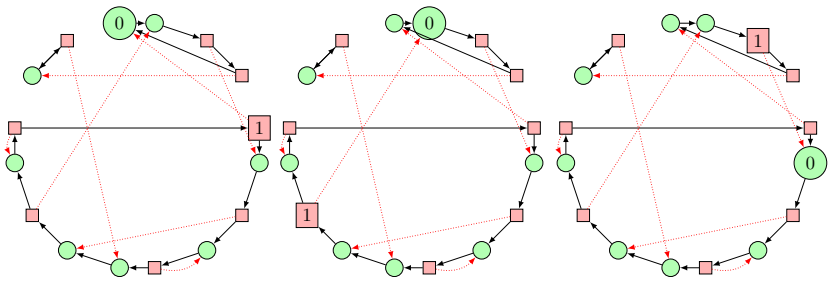


Figure 3.16 – Three graphs with same diffusion obtained with pair renumbering

even-odd skeletons whereas there are 163 skeletons with at least one ϵ-chain. For $2k = 32$, there are 231 even-odd skeletons, and 5591 non-even-odd ones.

Skeleton completion to enumerate GFN. Thanks to these skeletons we have reduced the number of permutation we have to enumerate for the general case of GFN. The number of permutation we have is "The number of skeletons $\times$ The enumeration of the edges left to set". The edges left to set are the odd to even edges ($\{(a, b) \mid a \in V_{\mathsf{o}}, b \in V_{\mathsf{e}}\}$), and the first and last edges of the ϵ-chains. If there is only one ϵ-chain, the number of edges to enumerate is $k! \times k$. If there are k ϵ-chains the formula is $(k!)^2$. An example of skeleton completion is given in Figure 3.17. The leftmost graph is the skeleton of Figure 3.15 and the rightmost one is a GFN. The number of edges to test is still too high to be naively

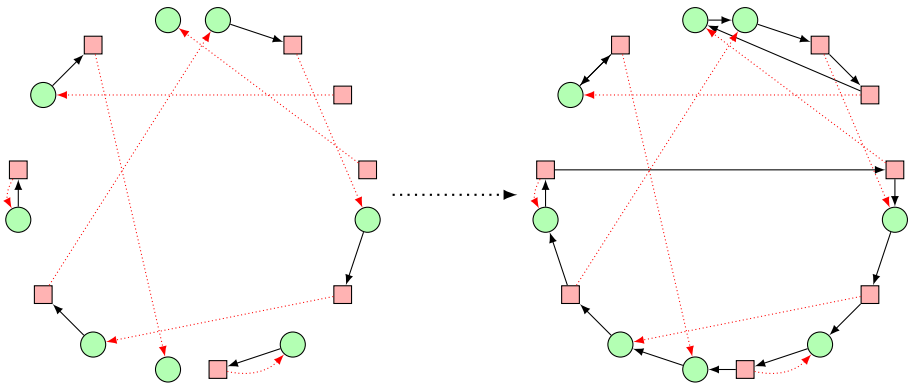


Figure 3.17 – Skeleton completion example

enumerated for high values of k. Therefore we will complete each skeleton using a path algorithm we developed for that purpose.

3.4.1 New recursive path algorithm

The previous enumeration methods *i.e.,* ([SM10, CGT19, DFLM19], were based on simple enumerations or restricted to even-odd permutations. The focus was made on satisfying the full diffusion for each node of the GFN. The new algorithm we are going to describe now, focuses on each path between two nodes and is not restricted to even-odd permutations. The algorithm will try every possible path between each pair of nodes. By doing so it will complete a GFN graph until all the edges are chosen. The complexity of this algorithm is $k(\frac{3k}{2})^R$ in the worst case (the edges are always free). If we compare it to the $(2k)!$ tests of diffusion round of the naive search, this algorithm is more efficient when R is quite low. Hopefully, the lower bound of the even-odd case suggests that R does grow much slower than k. Moreover, most of the paths are incompatible so the building of a GFN graph with optimal diffusion round is faster in practice. Thanks to Propositions 3.1 and 3.2, we will only consider paths of length $R-3$ from odd to even nodes in the even-odd case and paths of length $R-1$ from even to odd nodes in the general case. To obtain effective procedures, we enumerate the paths while building a GFN graph. With this method, the more paths we add to the graph, the fewer possibilities remain for the following ones. Thanks to this, we can also define a strategy to cut the search as soon as possible by trying the paths with the least possibilities first.

Our algorithm is composed of the three following functions:

— MakePath builds all the possible paths from a node a to a node b and is described in Algorithm 2. Starting from node a, the function calls itself on each possible next node for the path until it reaches b with the length R. More precisely, on a node x, there is only three possibilities:
  — If x is odd, there is one call to the even node $x-1$. In this call, the length l does not decrease because ϵ-transitions are not counted in the path length (line 2-3).
  — If $\pi[x]$ has already been set, we have no choice, and thus we follow it (line 4-5).
  — If $\pi[x]$ is free, we have to try all the remaining free nodes (line 7-9).

  On each valid path, the function calls NextPath that will choose the next path to build (line 13).

— HasProperty checks whether the property of interest is satisfied between two nodes. For example, when considering the full diffusion property, we have to check whether a path of length R exists between the two nodes. As we will see later,

some other properties can be considered too.

— NextPath chooses two nodes a and b that do not have the property described in HasProperty. If such a pair of nodes exists, it calls MakePath on it to link them with the next path. It is described in Algorithm 3.

Algorithm 2: MakePath(x, π, b, l)

```
Data: x: current node, π: partial permutation, b: target node, l: remaining length
      to reach R
1  if l > 0 then
2      if x is odd then
3          MakePath(x − 1, π, b, l);
4      if π[x] is set then
5          MakePath(π[x], π, b, l − 1);
6      else
7          for all y not used in π do
8              π[x] ← y;
9              MakePath(y, π, b, l − 1);
10         free π[x];
11 else if x = b then NextPath(π) ;
```

Algorithm 3: NextPath(π)

```
Data: π: partial permutation
1  for all (a, b) given by Strategy() do
2      if ¬HasProperty(a, π, b, R) then
3          MakePath(a, π, b, R);
4          return ;
5  Add π to solution pool
```

Start and stop conditions. Our algorithm starts by a call to NextPath with an undefined permutation and a given global parameter R. It stops when one of the following conditions holds:

1. There is no possible path from a to b, and thus there is no solution.
2. The permutation is complete, i.e. fully defined: it is a solution if HasProperty is true for each pair of nodes.

3. The algorithm ends without seting the whole permutation. In this case, any completion of the permutation leads to a valid solution.

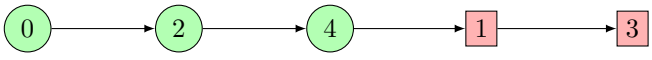
Once all the recursive branches of our algorithm have been explored, all the paths of length R have been exhausted. Thus, at the end of the algorithm, we find all the permutations achieving full diffusion at round R if any. The algorithm can build these permutations from scratch, but it will find a lot of similar solutions. This corresponds to the pair-equivalence classes defined in [CGT19]. To avoid these redundancies, we use the *skeletons* defined in the previous section to reduce the search space. The algorithm can be run on each skeleton independently to help parallelization. Nevertheless, there are some symmetries left in our algorithm. Indeed, a l-ϵ-cycle will produce l similar solutions. Moreover, if there are m times the same ϵ-cycle or ϵ-chain, there will be $m!$ similar solutions. Breaking these symmetries in our algorithm increases its running time, and it is left to future work to take them into account effectively.

The search strategy. For the choice of a and b in the NextPath function, the strategy consists of starting by the paths with the least possibilities. To do so, we can either count the remaining possible paths during the search, or we can set a static path priority based on the skeleton of the graph. The best strategy we found was to first build the paths that start and end on the smallest ϵ-chains. This is because the paths starting by consecutive even nodes and ending by consecutive odd nodes have fewer possibilities and therefore are most likely to be impossible to build.

Example 3.3 (MakePath execution to find a GFN with DR=5)
Let the four following nodes as a starting structure example:

If we want a $DR = 5$, we need paths of length 4 from the green to the red nodes (Proposition 3.1). The algorithm will first try to make a path from the node 0 to 3. There are two possibilities.

The first possibility for 0 to 3 *is to add a node to link 2 and 1 like:*



Once this path is completed the next path to build will be from 0 to 1 with length 4. Regardless of the nodes added in the previous step, this path cannot be built. The algorithm will then try the second possibility for the first path

The second possibility for 0 to 3 *is to add a pair of nodes linked by an ϵ-transition :*

The path from 0 *to* 3 *is completed and there is only one option left for the path from* 0 *to* 1*:*

The path from 2 *to* 3 *already exists so the next path to build is the one from* 2 *to* 1*. Again, this path is impossible and we have exhausted all the possibilities so the algorithm concludes that from this starting structure, there is no permutation with* $DR = 5$*. Note that this example is valid for any* k *except for* $k = 2$ *because in this case, the node* 4 *would not have been available.*

3.4.2 Results for the non-even-odd case

To test whether a non-even-odd permutation can have a better diffusion round than the even-odd ones, we used Algorithm 2 on all the skeletons having at least one ϵ-chain. We set R to be one round less than the diffusion round known for the best even-odd permutation, and ran our algorithm with the property HasPath (described in Algorithm 4).

Algorithm 4: HasPath(x, π, b, l)

Data: x: current node, π: partial permutation, b: target node, l: remaining length

```
1 if l > 0 then
2 |   return (x is odd ∧ HasPath(x − 1, π, b, l))
3 |          ∨ (π[x] is set ∧ HasPath(π[x], π, b, l − 1));
4 else return x = b ;
```

The case $2k = 22$ is easy with our algorithm so we increased R to find the optimal non-even-odd permutations. They are given in Table 3.5. These optimal permutations have a diffusion round of 9 which is one round more than the optimal even-odd permutations.

For $2k = 24$ to $2k = 32$, our algorithm ended without finding any non-even-odd permutations with a better diffusion round than the optimal even-odd ones. As a result, we establish that the non-even-odd permutations do not achieve a better diffusion round than the even-odd permutations up to $2k = 32$. For $2k = 34$ the algorithm found no non-even-odd permutations for $R = 8$ but the optimal even-odd permutation has a diffusion

π =(3, 18, 5, 16, 7, 12, 9, 10, 1, 14, 13, 2, 15, 8, 11, 21, 17, 4, 19, 6, 0, 20)
π =(3, 6, 5, 12, 7, 10, 9, 18, 1, 2, 13, 4, 15, 16, 17, 8, 11, 21, 19, 14, 0, 20)
π =(3, 12, 5, 0, 7, 10, 9, 18, 1, 2, 13, 4, 15, 16, 17, 21, 11, 8, 19, 14, 6, 20)
π =(3, 8, 5, 16, 7, 21, 9, 14, 1, 2, 13, 18, 15, 0, 17, 6, 11, 12, 19, 4, 10, 20)
π =(3, 21, 5, 10, 7, 0, 9, 14, 1, 2, 13, 18, 15, 8, 17, 6, 11, 12, 19, 4, 16, 20)
π =(3, 8, 5, 6, 7, 4, 1, 12, 11, 2, 9, 21, 15, 19, 13, 17, 10, 16, 14, 20, 0, 18)
π =(3, 4, 5, 14, 7, 0, 9, 16, 11, 2, 1, 12, 15, 21, 13, 6, 19, 10, 17, 8, 18, 20)
π =(3, 6, 5, 10, 7, 16, 9, 18, 11, 14, 1, 2, 15, 4, 13, 0, 19, 8, 17, 21, 12, 20)

Table 3.5 – Optimal non-even-odd permutations for 2k=22

round of $R = 10$. Sadly, our algorithm with $R = 9$ did not end in less than a week of computation. However, for $2k = 36$ since the best even-odd permutation has a DR of 9 we were able to show that there is no even-odd permutation strictly better. Results of this algorithm are summarised in Table 3.6. They have been published in [DDGP22]. These results have been obtained on a 128 core CPU (AMD EPYC 7742 64-Core Processor) and the source code is publicly available at:

https://gitlab.inria.fr/agontier/ANewAlgoForGFN

For a light comparison, in [CGT19], it is mentioned that "$2^{46.4}$ tests of diffusion rounds" are needed when considering 20 blocks. Actually, our algorithm is faster and tackles this instance in around 8 seconds on our server.

3.5 New criteria

As studied in the literature, the diffusion round is a property that can be used to find good Feistel permutations. This criteria is tied to the resistance of the resulting ciphertext against e.g. impossible differentials, saturation attacks and pseudorandomness analysis [SM10]. However, permutations with optimal diffusion round can also be weak against other cryptanalysis techniques. For instance, the designers of `WARP` [BBI+20] selected a permutation achieving full diffusion in 10 rounds while permutations with a diffusion round of 9 actually exist. The main reason is that all optimal permutations for the diffusion round are much weaker regarding truncated differential cryptanalysis than the one they selected. These permutations require at least 32 rounds to reach 64 active S-Boxes, while the permutation used in `WARP` (which is non optional w.r.t. the diffusion round) only requires 19 rounds to reach the same resistance.

$2k$	Fibonacci bound	even-odd		non-even-odd	
		DR	Ref	DR	Ref
6	5	5	[SM10]	6	[SM10]
8	6	6		6	
10	6	7		7	
12	7	8		8	
14	7	8		8	
16	7	8		8	
18	8	8	[CGT19]	9	[CGT19]
20	8	9		9	
22	8	8		9	[DDGP22]
24	8	9		≥ 9	
26	8	9	[DFLM19]	≥ 9	
28	9	9		≥ 9	
30	9	9		≥ 9	
32	9	9		≥ 9	
34	9	10		≥ 9	
36	9	9		≥ 9	

Table 3.6 – State of the art regarding optimal Diffusion Round.

Therefore, it would be interesting to look for other properties which might lead to stronger ciphers. With our algorithm it is quite simple to change the property we are looking for as we only need to provide a new HasProperty function. In this section, we thus propose several properties derived from the diffusion round and study the quality of their solutions against truncated differential cryptanalysis. We consider two properties, the first one is a generalization of the diffusion round where we consider not one but X paths between each pair of blocks. The second one consists of counting the S-Boxes on each path instead of the paths themselves.

3.5.1 Number of paths

The diffusion round property ensures that each solution has at least one d-path of length R between each pair of blocks. We propose a new property parameterized by an integer X, namely X-DR, which extends the diffusion round to at least X d-paths of length R between each pair of blocks.

Definition 3.11 *X-$DR(\pi)$ is the smallest integer R such that:*

$\forall u, v \in V$, there are at least X d-paths of length R from u to v in G_π.

This new property introduces the parameter X denoting the minimum number of paths we want between each pair of nodes. When $X = 1$, this corresponds to the full diffusion property. To use this new property in our algorithm, the call to HasProperty (line 2 of Algorithm 3) is replaced by a call to NumberOfPaths with the slight modification that this number of paths must be greater or equal to the parameter X. This function counts the number of paths between two nodes, it is given in Algorithm 5.

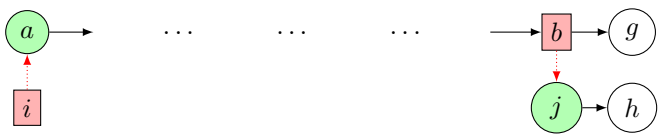
Algorithm 5: NumberOfPath(x, π, b, l)

```
   Data: x: current node, π: partial permutation, b: target node, l: remaining length
 1 if l > 0 then
 2 |  if π[x] is set then
 3 |  |  if x is odd then
 4 |  |  |  return
   |  |  |    NumberOfPath(x − 1, π, b, l) + NumberOfPath(π[x], π, b, l − 1);
 5 |  |  else return NumberOfPath(π[x], π, b, l − 1) ;
 6 |  else return 0 ;
 7 else
 8 |  if x = b then return 1 ;
 9 |  else return 0 ;
```

Since we want more than one path between two nodes, the function MakePath may need to create multiple paths. Due to these multiple paths, we must set an order between paths to prevent introducing new symmetries. For example, we should not build a path p after a path q if we already tried to build them in the reverse order. Proposition 3.2, stated and proved for the diffusion round, is still valid when considering X-DR. It is stated in Proposition 3.3, the proof is very similar.

Proposition 3.3 *Let π be an even-odd permutation, X-$DR(\pi)$ is the smallest integer R such that: $\forall c \in V_o, d \in V_e$, there are X paths of length $R-3$ from c to d in G_π.*

Proof. Let $b, c \in V_o$ and $a, d \in V_e$ with $(a, c), (d, b) \in E_\pi$. We have that $(a+1, a), (b, b-1) \in E_\epsilon$ with $a + 1 \in V_o$ and $b - 1 \in V_e$. Furthermore, we have $g, h \in V$ such that $(b, g), (b - 1, h) \in E_\pi$ (see the graph below with $i = a + 1$ and $j = b - 1$).



1) From Definition 3.11, we know that there is X d-paths of length R from a to g, thus there is X paths of length $R-3$ from c to d.

2) Now suppose that there is $R' < X\text{-}DR(\pi)$ such that $\forall\ c \in V_{\mathsf{o}},\ d \in V_{\mathsf{e}}$ there is X paths of length $R'-3$ from c to d. We then have X d-paths of length R' from i to g, from i to h and from a to h. Since we have these d-paths for all pairs $a \in V_{\mathsf{e}}, b \in V_{\mathsf{o}}$ then we have full diffusion with $X\text{-}DR(\pi) = R'$ and thus the contradiction $X\text{-}DR(\pi) < X\text{-}DR(\pi)$. □

To compare this criterion w.r.t. truncated differential analysis, we computed the minimal number of active S-Boxes for each possible permutation by enumerating all of them for $k = 6$, $k = 7$, and $k = 8$. We give in Table 3.7 the best number (i.e. the minimum one) we obtained from round 1 to round 16 :

Round / k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
6	0	1	2	3	4	6	8	11	14	16	19	22	24	26	28	29
7	0	1	2	3	4	6	8	11	14	19	23	26	28	30	33	35
8	0	1	2	3	4	6	8	11	14	19	22	26	29	31	34	37

Table 3.7 – Best minimal number of active S-Boxes for each round

The case $k = 8$ is the most interesting one and we describe it more precisely for each cycle decomposition in Table 3.8. The red bold numbers are the optimal number of S-Boxes for $R \geq 10$. In this table, we can see that there is a very low proportion of optimal solutions. We also see that a solution cannot have the maximal number of S-Box on all the rounds at the same time.

To compare the new criterion with the diffusion round, we took the 500 first solutions given by our algorithm for the criterion. We choose the number 500 arbitrary to have a representative solution pool and also have fast computing times. We computed the minimal number of active S-Boxes for each of these solutions, and we counted the number of solutions that reached the optimal value for each round from 10 to 16. The results are given in Table 3.9. Note that to get 500 solutions, we sometimes needed to consider the criterion to a higher round than the optimal one. For example four paths diffusion round for $k = 8$ is $R = 10$. However, there are less than 500 solutions with these parameters. Thus, we had to increase R until we reached 500 solutions. This is summarized in the range column of Table 3.9.

For $k = 8$, we do not see a trend and we have similar results for $k = 7$ and $k = 6$. In fact, the property seems uncorrelated to the optimal number of active S-Boxes. We

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14
Max number of S-Box	0	1	2	3	4	6	8	11	14	**19**	**22**	**26**	**29**	**31**
p.c. sol reaching max	100%			37%			10%	4.4%	3.9%	.0013%	.0012%	.0006%	.0012%	.0034%
nb sol reaching max	887040			326326			88654	39670	34679	52	48	24	48	136
Cycle decomposition	Best minimal number of active S-Box													
[8]	0	1	2	3	4	6	8	11	14	17	21	24	28	**31**
[7, 1]	0	1	2	3	4	6	8	11	14	**19**	21	24	28	30
[6, 2]	0	1	2	3	4	6	8	11	14	**19**	**22**	24	27	30
[6, 1, 1]	0	1	2	3	4	6	8	11	14	18	21	**26**	28	30
[5, 3]	0	1	2	3	4	6	8	11	14	17	21	24	27	30
[5, 2, 1]	0	1	2	3	4	6	8	11	14	18	21	24	28	30
[5, 1, 1, 1]	0	1	2	3	4	6	8	11	14	17	21	24	27	30
[4, 4]	0	1	2	3	4	6	8	11	14	18	**22**	24	27	30
[4, 3, 1]	0	1	2	3	4	6	8	11	14	18	21	**26**	**29**	**31**
[4, 2, 2]	0	1	2	3	4	6	8	11	14	17	21	24	27	**31**
[4, 2, 1, 1]	0	1	2	3	4	6	8	11	14	18	**22**	24	28	30
[4, 1, 1, 1, 1]	0	1	2	3	4	6	8	11	14	16	19	23	27	**31**
[3, 3, 2]	0	1	2	3	4	6	8	11	14	18	21	24	27	30
[3, 3, 1, 1]	0	1	2	3	4	6	8	11	14	**19**	**22**	24	**29**	**31**
[3, 2, 2, 1]	0	1	2	3	4	6	8	11	14	17	21	24	27	30
[3, 2, 1, 1, 1]	0	1	2	3	4	6	8	11	14	16	19	24	27	29
[3, 1, 1, 1, 1, 1]	0	1	2	3	4	6	7	9	12	14	17	20	24	26
[2, 2, 2, 2]	0	1	2	3	4	6	8	11	14	16	19	24	26	29
[2, 2, 2, 1, 1]	0	1	2	3	4	6	8	11	14	17	20	24	27	**31**
[2, 2, 1, 1, 1, 1]	0	1	2	3	4	6	8	11	14	16	19	22	24	27
[2, 1, 1, 1, 1, 1, 1]	0	1	2	3	4	6	7	9	12	14	16	19	22	25
[1, 1, 1, 1, 1, 1, 1, 1]	0	1	2	3	4	6	7	9	10	13	15	17	19	22
TWINE [SMMK12]	0	1	2	3	4	6	8	11	14	18	**22**	24	27	30

Table 3.8 – Best minimal number of active S-Boxes by cycle decomposition for $k = 8$

can see that increasing the parameter X increases the round R we need to go to find 500 solutions. Indeed when we search for two paths instead of one, the property is so strict that there are no solutions for $R = 8$. We also see that few to none of the 500 solutions are optimal in general.

3.5.2 Number of S-Boxes

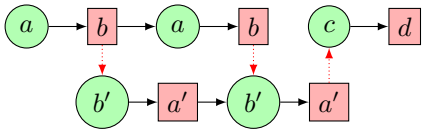
Having X paths between each pair of blocks does not ensure that these paths are good from the differential analysis point of view. Instead of constraining the number of paths, we propose to ensure that a minimum number of S-Boxes are present in the d-paths between each pair of blocks.

Definition 3.12 *X-$SB(\pi)$ is the smallest integer R such that: $\forall u, v \in V$, there are X S-Boxes traversed by d-paths of length R from u to v in G_π. A S-Box reached by two paths of the same length will be counted only once.*

Round / X-DR	10	11	12	13	14	15	16	Range
1 path	9	16	0	0	0	0	0	8
2 paths	24	37	0	0	0	0	0	9
3 paths	0	4	0	0	0	0	0	10
4 paths	15	15	0	0	0	0	0	10-11
5 paths	0	1	0	0	0	0	0	11
6 paths	9	9	0	0	0	0	0	11-12
7 paths	0	0	0	0	0	0	0	12
8 paths	0	0	0	0	0	0	0	12

Table 3.9 – Number of solutions with an optimal number of active S-Boxes from round 10 to round 16 in the 500 first solutions considering $k = 8$

For example, in the two paths of length 5 from a to d depicted below, the S-Box corresponding to the red edge (b, b') will be counted twice (as it occurs at two different lengths), whereas the S-Box corresponding to the red edges (a', c) will be counted only once (even if it occurs on both paths).



To use this new property in our algorithm, the call to HasProperty (line 2 of Algorithm 3) is replaced by a call to DetectS-Boxes with the slight modification that the sum of detected S-Boxes must be greater or equal to the parameter X. DetectS-Boxes is described in Algorithm 6. Unlike paths, we cannot simply count the S-Boxes because of the redundancy described in the previous example. We have to use a Boolean matrix of dimension 2 or an equivalent structure to remember at which path length l we encountered each S-Box.

Proposition 3.2, stated and proved for the diffusion round, is also valid when considering X-SB. It is stated in Proposition 3.4, the proof is very similar to previous ones.

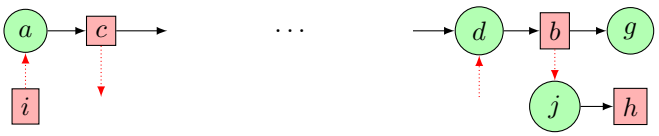
Proposition 3.4 *Let π be an even-odd permutation, X-$SB(\pi)$ is the smallest integer R such that: $\forall c \in V_o, d \in V_e$, there are X S-Boxes traversed by paths of length $R-3$ from c to d in G_π. A S-Box reached by two paths of the same length will be counted only once.*

Proof. Let $b, c \in V_o$ and $a, d \in V_e$ with $(a, c), (d, b) \in E_\pi$. We have that $(a+1, a), (b, b-1) \in E_\epsilon$ with $a + 1 \in V_o$ and $b - 1 \in V_e$. Furthermore, we have $g, h \in V$ such that

Algorithm 6: DetectS-Boxes(x, π, b, l, M)

```
Data: x: current node, π: partial permutation, b: target node, l: remaining
      length, M: Boolean matrix of dimension 2
M0 ← Matrix filled with false values;
if l > 0 then
    if π[x] is set then
        if x is odd then
            M2 ← copy(M);
            M2[x, l] ← true;
            M3 ← DetectS-Boxes(π[x], π, b, l − 1, M);
            M4 ← DetectS-Boxes(x − 1, π, b, l, M2);
            return Bit-wise OR(M3,M4);
        else return DetectS-Boxes(π[x], π, b, l − 1, M) ;
    else return M0 ;
else
    if x = b then return M ;
    else return M0 ;
```

$(b, g), (b-1, h) \in E_\pi$ (see the graph below with $i = a+1$ and $j = b-1$).



1) From Definition 3.12, we know that there is X S-Boxes in all the d-paths of length R from a to g, thus there is X S-Boxes in all paths of length $R-3$ from c to d.

2) Now suppose that there is $R' < X\text{-}SB(\pi)$ such that $\forall\ c \in V_o,\ d \in V_e$ there is X S-Boxes in all the paths of length $R'-3$ from c to d. We then have X S-Boxes in all the d-paths of length R' from i to g, from i to h and from a to h. Since we have these d-paths for all pairs $a \in V_e, b \in V_o$ then we have full diffusion with $X\text{-}SB(\pi) = R'$ and thus the contradiction $X\text{-}SB(\pi) < X\text{-}SB(\pi)$. □

As for the $X\text{-}DR$ criterion, we looked at the quality of optimal permutations for the $X\text{-}SB$ criterion regarding truncated differential cryptanalysis for $k = 6$, $k = 7$, and $k = 8$. The results are summarized in Table 3.10 for $k = 8$ and are similar for lower k.

Overall, these two new properties did not bring better solutions for the truncated differential analysis. For each criterion, the number of optimal solutions in the 500 first solutions is very low.

Round / X-SB	10	11	12	13	14	15	16	Range
1 S-Box	25	44	0	0	0	0	0	8
2 S-Boxes	25	44	0	0	0	0	0	8
3 S-Boxes	0	1	0	0	0	0	0	9
4 S-Boxes	18	30	0	0	0	0	0	9
5 S-Boxes	4	12	0	0	0	0	0	9-10
6 S-Boxes	4	9	0	2	2	2	0	9-10
7 S-Boxes	0	6	0	0	0	0	0	10
8 S-Boxes	0	9	0	0	0	0	0	10-11
9 S-Boxes	0	1	0	1	15	1	0	11
10 S-Boxes	0	4	0	0	0	0	0	11
11 S-Boxes	0	6	0	0	0	0	0	11-12
12 S-Boxes	0	0	0	0	0	0	0	11-12

Table 3.10 – Number of solutions with an optimal number of active S-Boxes from round 10 to round 16 in the 500 first solutions considering $k = 8$

Comparison with the permutation used in TWINE [SMMK12] The values of our criteria for `TWINE` are given in Table 3.11. To see if these are good values, we used our algorithm to enumerate permutations with strictly greater values for our criteria. The algorithm concluded that there is no permutation with a better X-SB than `TWINE` up to $X = 22$. The experimentation was not done beyond due to its computational cost. However, `TWINE` is not optimal for 4-DR and 6-DR. There is only one permutation that

1 to 2-SB	3 to 6-SB	7 to 8-SB	9 to 14-SB	15 to 22-SB
8	9	10	11	12
1-DR	2-DR	3-DR	4 to 5-DR	6 to 9-DR
8	9	10	11	12

Table 3.11 – X-DR and X-SB values for `TWINE`

is optimal on 4-DR and 6-DR at the same time. This permutation is π =(3, 4, 5, 8, 1, 12, 9, 10, 11, 2, 7, 14, 13, 6, 15, 0). To compare it with `TWINE`, we computed the truncated differentials on both permutations in Table 3.12.

Round	8	9	10	11	12	13	14	15	16
TWINE	**11**	**14**	18	**22**	24	27	30	32	35
π	**11**	**14**	**19**	**22**	24	26	28	30	32

Table 3.12 – Truncated Differentials for `TWINE` and π

This new permutation π is better than `TWINE` and optimal at round 10. However, it is worse for rounds 13 to 16. In fact, in all the permutations with $k = 8$, none can reach the optimal number of active S-Boxes at every round.

3.5.3 Maximum S-Box path

In this section, we are trying to build a GFN that contains the maximal number of S-Boxes in the even-odd case. To do so, we use paths with the maximal number of S-Boxes (*i.e.,* ϵ-transitions). Between two nodes a and b, the path with the maximal number of ϵ-transitions is unique and has $R + 1$ ϵ-transitions. This maximal path alternates edges from E_π and E_ϵ. We call it the max-path (see the first path of Figure 3.18). In order to achieve full diffusion, we need more paths. The second best path is one that alternates edges from E_π and E_ϵ except that it has somewhere three consecutive edges in E_π. We call it a sec-path path and it has $R - 1$ ϵ-transitions (see the second and third paths of Figure 3.18 as examples).

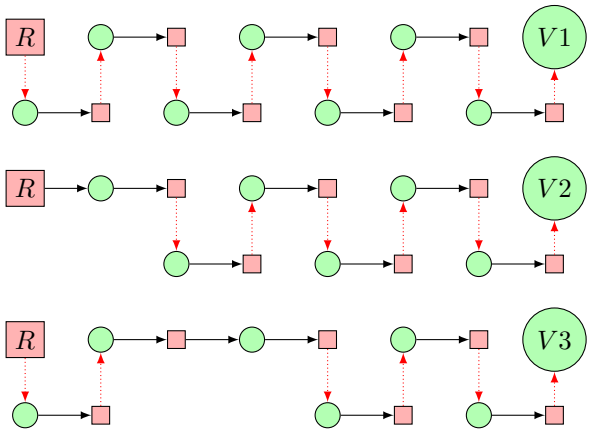


Figure 3.18 – Paths with maximal number of S-Boxes

To build a GFN with only these paths (the max-path and the sec-paths) we are in fact restricted to only one skeleton. The reason is the following:

— First of all, we need k paths to have the full diffusion so we need $k - 1$ sec-paths. To have $k - 1$ different sec-paths from the same starting node, we need them to be of length $R = k - 1$. Thanks to Proposition 3.2 this makes $DR = k + 2$.

— Secondly, from a starting node, we have k paths to reach k nodes so we cannot have two paths reaching the same destination.

When combined with a skeleton, this second reason becomes a very restrictive constraint. To properly define this constraint we need some new definitions first. First note that ϵ-cycles contain all the edges from V_{e} to V_{o}. Therefore, edges from V_{o} to V_{e} have either both ends in the same ϵ-cycle (inner edge) or each end on a different ϵ-cycle (outer edge).

Definition 3.13 (Shift) *Let an ϵ-cycle and an inner edge e from node a to node b. We call the shift of e the number of ϵ-transition from a to b in the ϵ-cycle (some shift examples are given in Figure 3.19).*

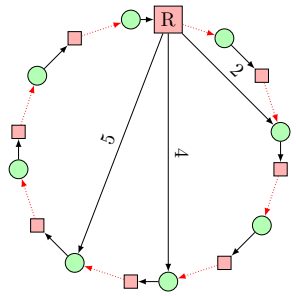


Figure 3.19 – Inner edges of shifts 2, 4 and 5 from a node R in a 8-ϵ-cycle

The max-path has no edge from V_{o} to V_{e} so it cannot leave an ϵ-cycle or take an inner edge. The sec-paths have exactly one edge from V_{o} to V_{e} so they can leave an ϵ-cycle once or they can have one inner edge. We then have the following property:

Property 3.2 *All the sec-paths starting on a node R cannot take an inner edge whose shift is equal to 2 and each of these shifts must be different.*

Proof. A sec-path from a node R with a shift equal to 2 has the same destination as the max-path from R. If two sec-paths start on R and have the same shift, they will end up at the same destination. □

Finally we have that:

Property 3.3 *A GFN of $DR = k+2$ built from the max-path and the sec-paths can only have the skeleton $(k-1, 1)$.*

Proof.

1) An ϵ-cycle without inner edge can only be of size one: Any sec-path that start on this ϵ-cycle cannot end in it because they take exactly one outer edge. Therefore, only the max-path stays on the ϵ-cycle.

2) An ϵ-cycle of size smaller than $k-1$ cannot have inner edges: The length of the sec-paths is equal to $k-2$, therefore on smaller ϵ-cycle the sec-paths will be able to take two times the same inner edge and end up on the same destination node.

3) An ϵ-cycle of size k cannot have all the paths: From each node $n_i \in V_o$ of the ϵ-cycle, the nodes $\{n_i, n_{i+1}, \ldots, n_{i+k-2}\}$ must map to the shifts $\{0, 1, 3, 4 \ldots, k-1\}$. But if $\{n_0, n_1, \ldots, n_{k-2}\}$ map to $\{0, 1, 3, 4, 5, \ldots, k-1\}$ then $\{n_1, n_2 \ldots, n_{k-1}\}$ map to $\{1, 3, 4, \ldots, k-1, 0\}$ and $\{n_2, n_3, \ldots, n_{k-1}, n_0\}$ map to $\{3, 4, \ldots, k-1, 0, 0\}$. Two sec-paths from n_2 have the same destination.

Therefore, the only ϵ-cycle decomposition available is $(k-1, 1)$ □

CP model. By setting this skeleton, a CP model can easily find all GFN permutations with the max-paths. Since the skeleton is set, we need to find the odd to even permutation q which is an array of integer variables of domain the node labels. To break some more symmetries, we choose the node 0 to be the 1-ϵ-cycle . It is connected to the $(k-1)$-ϵ-cycle with the edge $(1, 0)$.

$$\mathit{Alldifferent}(q)$$

$$q[1] = 0$$

All shifts must be different and not be equal to 2 (Property 3.3).

$$\mathit{Alldifferent}(\{shift(i, q[i]),\ \forall i \in [\![2, k-1]\!]\})$$

$$\mathit{shift}(i, q[i]) \neq 2 \quad \forall i \in [\![2, k-1]\!]$$

In Minizinc, the shift function is written: if $p[i] \geq i$ then $p[i] - i$ else $p[i] + k - 1 - i$.

This model is very efficient and can compute GFN with k up to 2048 in an hour. We report in Table 3.13 the truncated differentials for all the GFN with $k = 8$ to compare them to TWINE and the optimal values. The red bold numbers indicate when the number of active S-Boxes is optimal. This table shows that this method gives good to optimal differential characteristics but not consistently. Moreover, the DR is in the worst case

	Number of active S-Box																
Permutation	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22	R23	R24
[3 0 1 6 2 5 7 4]	10	12	15	19	24	25	27	29	30	32	35	37	39	42	43	45	48
[3 0 6 2 7 5 4 1]	11	14	16	18	22	25	27	30	31	33	35	37	39	42	43	45	48
[3 0 7 6 4 2 5 1]	11	13	15	17	20	24	26	28	30	33	36	38	39	41	44	47	51
[3 0 5 7 2 4 6 1]	11	14	15	16	18	20	23	26	28	31	36	38	40	42	44	46	48
[3 0 2 6 1 4 7 5]	10	12	15	18	22	26	28	30	32	34	36	38	40	42	44	46	48
[3 0 7 4 1 5 2 6]	11	14	17	20	23	26	28	30	32	34	36	38	40	42	44	46	48
[3 0 5 2 1 6 4 7]	11	14	17	20	**24**	26	29	31	32	33	35	37	40	43	46	49	52
[3 0 6 4 2 1 5 7]	11	14	15	18	22	25	27	30	31	33	36	37	39	42	43	45	48
[3 0 6 2 4 1 7 5]	11	14	**19**	21	**24**	25	27	30	31	33	36	37	39	42	43	45	48
[3 0 6 2 5 1 4 7]	11	14	16	19	22	25	28	30	32	33	35	37	40	43	47	**50**	**53**
[3 0 2 7 5 1 4 6]	10	12	15	18	22	25	27	30	32	34	36	38	40	42	44	46	48
[3 0 7 2 5 1 6 4]	11	14	16	20	22	26	28	30	33	35	36	38	40	43	45	48	51
[3 0 1 6 5 2 4 7]	11	14	**19**	21	**24**	25	27	30	31	33	36	37	39	42	43	45	48
[3 0 1 4 7 2 6 5]	11	14	17	20	**24**	25	27	30	31	33	36	37	39	42	43	45	48
[3 0 5 1 4 2 7 6]	10	12	15	16	18	20	23	26	29	33	36	38	40	41	43	45	48
[3 0 1 6 4 2 7 5]	11	14	16	19	22	24	27	30	31	33	35	37	39	42	43	45	48
[3 0 6 1 5 4 2 7]	10	12	15	19	22	24	27	30	32	34	36	37	39	41	44	47	49
[3 0 1 7 4 6 2 5]	11	13	15	17	19	21	24	27	30	33	36	37	39	42	43	45	48
[3 0 2 1 7 6 5 4]	10	12	14	16	18	20	22	25	29	33	36	37	39	42	44	46	48
TWINE	11	14	18	22	**24**	27	30	32	35	36	39	41	44	45	48	**50**	**53**

Table 3.13 – Differential characteristics of GFN with maximal paths

$k+2$. Similarly to the number of S-Boxes criteria, we conclude that having good paths or even the best paths is not sufficient to have the optimal number of active S-Boxes. This means that this property can be useful if combined with other criteria like the diffusion round.

3.6 Towards a lower bound proof for the general case

The proof that the non-even-odd permutations cannot lead to GFN with better diffusion has only been established empirically and we expanded it to 32 blocks, but a formal proof would be very much welcomed. In [CGT19], the authors started a reasoning on a lower bound for the non-even-odd case to compare it to the Fibonacci lower bound of the even-odd case. Unfortunately, there are a lot of possible *diffusion trees* in the general case. In the end, they were not able to reason on the diffusion trees because it is not clear which tree is the best at each depth.

Intuitively, a non-even-odd permutation should not reach a better diffusion round than the optimal even-odd one. Indeed, every time there are two consecutive odd nodes $u, v \in V_{\mathsf{o}}$ such that $(u, v) \in E_\pi$, there are also somewhere in the graph G_π two consecutive even nodes $x, y \in V_{\mathsf{e}}$ such that $(x, y) \in E_\pi$. We recall that each odd node has two outgoing edges (one in E_π and one in E_ϵ) whereas each even node has only one. Therefore, all the

paths starting from the node x have one edge less to achieve full diffusion and any path that passes through (u, v) will gain one edge. Since the number of even to even edges is the same as the number of odd to odd edges, one could think that they compensate.

3.6.1 An interesting example

One of our objectives during this work was to provide a formal proof that the diffusion round of the non-even-odd permutations is also bounded by the Fibonacci bound as for even-odd permutations. We recall that the Fibonacci suite gives an upper bound on the number of paths and thus a lower bound on the diffusion round in the even-odd case. Thus we made the following conjecture:

Conjecture: "The number of paths of length R in a non-even-odd GFN graph can not be greater than the number of paths of length R in an even-odd GFN graph and the same property holds for the inverse permutation too."

However, we found a non-even-odd permutation for which the number of odd nodes reached from the even nodes was in total, and with redundancy, greater than the even-odd Fibonacci bound, which suggests that an improvement of the diffusion round is possible by considering non-even-odd permutations. This permutation is given in Example 3.4.

Example 3.4 *We consider the permutation π =(3,2,1,5,0,6,7,4) depicted in the leftmost graph of Figure 3.20. The rightmost one represents π^{-1}.*

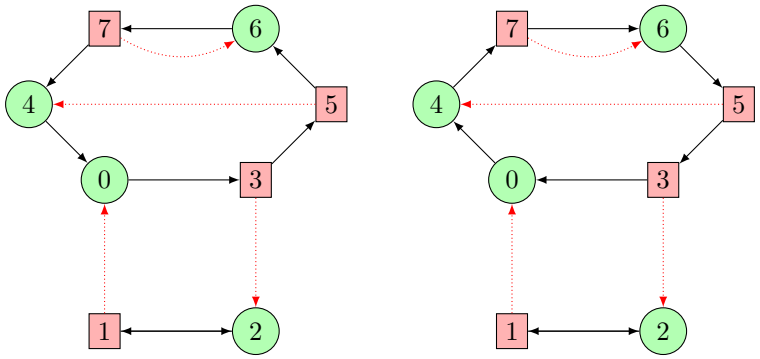


Figure 3.20 – Permutation graph of π and π^{-1}

On these two graphs, we give in Table 3.14 the number of paths of length $R = 5$ from even to odd nodes. There are 22 paths for π, and 21 paths for π^{-1}.

start node	0	2	4	6
number of paths	5	8	5	4

start node	0	2	4	6
number of paths	4	5	5	7

Table 3.14 – Number of paths in π and π^{-1}

When considering only the even-odd permutations, the maximum number of paths given by the Fibonacci suite is 5 for each node and thus $4 \times 5 = 20$ in total. This example shows that the diffusion round in the general case (i.e. considering both even-odd and non-even-odd permutations) cannot be bounded by the Fibonacci suite if we consider the sum of all paths on π and π^{-1}. However, we may note that there is one node (e.g. node 6 for π) having less paths than the Fibonacci suite. We always observe this phenomenon on the permutations we considered. We think that to establish a conjecture like stated before, we should focus on these nodes.

3.6.2 Decaying trees

We have the intuition that the lower bound on the diffusion round in the general case should not be better than the Fibonacci bound. An idea to find a proof of this would be to reason on the diffusion trees of the weak nodes observed in the previous section. We did not found a clean formulation of this idea, and we only give below an example to explain it.

Example 3.5 *We start by the even-odd diffusion tree of depth* 6 *(Figure 3.21).*

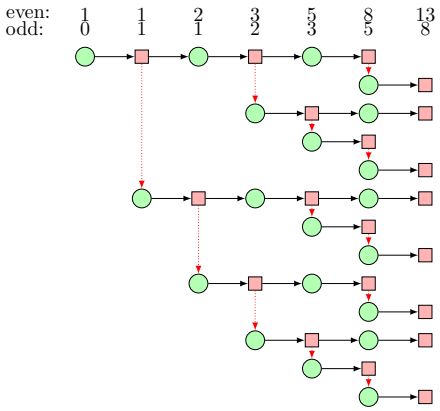


Figure 3.21 – Diffusion tree in the even-odd case

In order to have a better diffusion tree than the one depicted in Figure 3.21, more than 8 *odd nodes are needed in the leaves, and thus an extra red to red transition has to*

be added. For instance one option would be the tree of Figure 3.22:

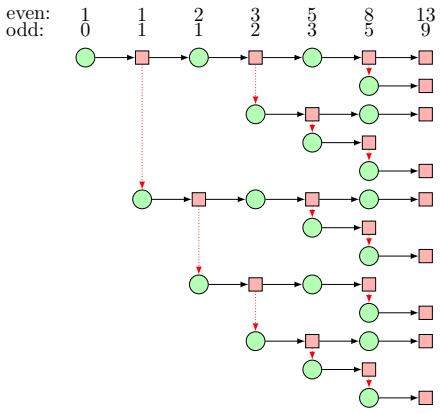


Figure 3.22 – Better diffusion tree than the even-odd case

With this non-even-odd edge we gained one extra odd node. However, the diffusion round must be at least equal for the inverse permutation too. In the graph representation, the inverse graph is the same graph but with inverted edges and the node colors swapped. So the red to red transition becomes a green to green transition in the inverse graph. The diffusion round also need to be better on the inverse permutation that starts by this green to green transition. To compensate this green to green transition, we need to add new red to red transitions to have more than 8 *odd nodes in the leaves. For example Figure 3.23.*

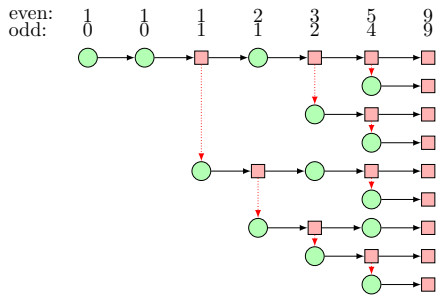


Figure 3.23 – Better diffusion tree than the even-odd case in the inverse

In this tree, we now have three consecutive red nodes that will be three consecutive green nodes in the inverse tree. Note that the upper path can also have two separated red to red transitions. In our example with depth 6, there is no problem because this path also lead to three consecutive red nodes in the next tree. However, we do not know if there will always be a worst path on trees of higher depth. In our case, the problem is that even if we

put red nodes everywhere else (Figure 3.24), we cannot get a better tree than the even-odd tree we started from.

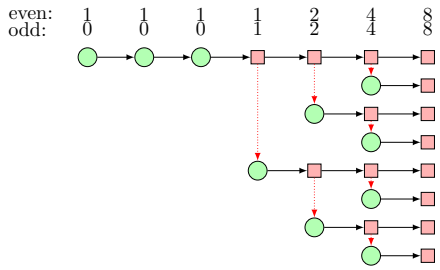


Figure 3.24 – Diffusion tree that cannot be better than the even-odd case

This suite of decaying trees gives a nice example but the generalization of this idea brings some open questions. What is the best way to add non-even-odd transitions to get a better tree ? Which part of the tree do we need to invert at each step ? Is there always a worst tree for higher depths ? Formalising this idea in order to obtain a form of proof is left as future work.

Chapter 4

Automatic generation of CP models

Introduction

In the previous chapters, we have seen that generic solvers can be valuable tools for solving cryptographic problems because they only require a model. However, modelling may not be straightforward and we still need to make a model for each cipher. In this chapter, we propose a CP model generator to solve the second step of differential cryptanalysis. We rely on the graph cipher representation of Tagada, a tool that generates MiniZinc models for the first step of differential cryptanalysis. Unlike the first step, we solely rely on the Choco CP solver because we need to implement new filtering algorithms and test their efficiency. Moreover, we propose to integrate implementation optimizations such as graph simplification and multi-core solving.

Takeaway

This contribution supports the following conclusions:

— CP models for differential analysis can be generated from the graph cipher representation of Tagada.

— We tested new filtering algorithms for cryptographic operators (XOR, multiplication in finite field, and LFSRs)

— The models can be automatically optimized.

4.1 Tagada

Tagada (Tool for Automatic Generation of Abstraction-based Differential Attack) is a tool proposed in [LDLS21] that can generate models for computing truncated differential

characteristics for any word-oriented cipher. It relies on a graph representation of the cipher and uses several techniques to optimize the models. In this background section, we first recall differential analysis and then explain the graph representation of Tagada and how it generates models for the first step of this problem. Moreover, we describe some of the optimizations Tagada proposed for the first step of this problem.

4.1.1 Differential cryptanalysis

Differential analysis is a method to analyze the effect of differences in plaintext pairs on the differences of the resultant ciphertexts. The difference is usually obtained with the bit-wise XOR; we will note it $+$. For a cipher function F and two plaintexts x and y where y is created by injecting an input difference δ_{in}, *i.e.,* $y = x + \delta_{in}$, the output difference δ_{out} is computed as $\delta_{out} = F(x) + F(y)$. There is a differential distinguisher for F if the probability that $\delta_{out} = F(x) + F(x + \delta_{in})$ is high *i.e.,* the input difference δ_{in} has a good probability of ending up in the output difference δ_{out}. Symmetric ciphers are iterated functions like $F(x) = f(f(\dots f(f(x))\dots))$. To see if the difference δ_{in} can end up by the difference δ_{out}, we study the propagation of the differences through all the rounds and all the cipher operators. We usually note δx_i the difference of an intermediate variable of the cipher x_i. The tracking of differences from δ_{in} to δ_{out} through the complete cipher is called a *differential trail* or *differential characteristic*.

Symmetric ciphers are generally composed of two types of operators: linear operators like XORs or permutations and non-linear operators like S-Boxes:

— *Linear operators* will always propagate differences with probability 1. Indeed a permutation will simply reorganize the differences. Another common linear operator is the XOR of three variables $y = x_2 + x_3$ were the differences will be propagated with another XOR: $\delta y = \delta x_2 + \delta x_3$. Note that if $\delta x_2 = \delta x_3$ then the output difference is cancelled $\delta y = 0$.

— However, *non-linear operators*, called S-Boxes, will propagate a difference with a probability that may be lower than one. For each S-Box, the propagation probabilities of the pair of input-output differences can be computed with their *Difference Distribution Table* (DDT). Since it is necessary to enumerate all the possible pairs of inputs to generate the DDTs they are cached and it is not possible to compute them if the input size of the associated operator is too big. Usually it is possible to compute DDTs for word oriented ciphers as they are working with 4-bit (nibbles)

and 8-bit (bytes) words. TAGADA is designed for word oriented ciphers, for which the propagation of differences in non-linear operators can be computed with DDTs. TAGADA will have very poor performances on bit-oriented ciphers (*e.g.* [BSS+13]), or on word-oriented ciphers that operate on words larger than bytes.

If a differential trail contains two or more S-Boxes, the probability of the trail is the multiplication of the probabilities given by the DDTs because we assume that the probabilities are independent [LMM91]. Therefore, the probability of a trail will be lower if we add more S-Boxes to it, and low-probability trails may not be useful for standard differential attacks (however, differential trails with zero probability can be used in impossible attacks for example). To make ciphers more resistant, we want to have S-Boxes with the most balanced probabilities possible. However, this is hard to achieve among all the other required properties of S-Boxes [Hey02].

As the search of differential trails is hard, it is usually split into two steps in the most recent works [BN10, FJP13, GLMS20].

First step: find truncated trails. The first step is the search for *truncated differential trails* [Knu94]. A truncated differential trail is an abstraction of a differential trail in which we only retain whether a difference exists or not, *i.e.,* each difference variable δx_i, associated to cipher's intermediate word x_i of size n, is abstracted by a Boolean variable Δx_i where

$$\Delta x_i = \begin{cases} 0 & \text{if} \quad \delta x_i = 0 \\ 1 & \text{if} \quad \delta x_i \in [1; 2^n - 1] \end{cases}$$

Because each Boolean variable Δx_i encodes the existence of the difference without tracking its value, several aspects of the problem change.

— For the probability of propagation through the S-Boxes, we cannot know which probability to pick in the DDT except when the Boolean variable associated with the input difference is equal to 0, in which case we know for sure that the Boolean variable associated with the output difference is also equal to 0. When the Boolean variable associated with the input variable is equal to 1, we only know that the S-Box is *active i.e.,* involved in the trail. Therefore, we will use the highest probability to have an upper bound on the probability of the differential trail.

— The second change is that we cannot capture the difference cancellations of the XOR operations. Therefore, there are usually a lot of false positive trails *i.e.,* trails that cannot be instantiated.

Truncated differential analysis can be enough to say that a cipher is secure in the case where the best truncated trail (the one with the fewest active S-Boxes) has a low enough probability. On the other hand, when a truncated trail has a high probability, we must successfully instantiate it with real difference values to have a differential trail.

Second step: instantiate the trails. In the second step, we enumerate all possible differential trails (starting from active S-boxes) to find the best one. The two steps can be done separately (find all truncated trails, then try to instantiate them all) or together like in Algorithm 9 that will be explained more in Section 4.3.

In [DDH+21], the authors compare a dedicated implementation (based on dynamic programming) with SAT, MILP, and CP models that are solved by generic solvers on the two steps of the differential cryptanalysis of the cipher SKINNY. In conclusion, they found that their hand-made algorithm is the fastest on the first step, followed by SAT and MILP. For the second step, they conclude that the CP solver is the most efficient solver by far. This work points out a key issue of cryptanalysis problems. It has to be done on every cipher, and the development time is not the same for a hand-made algorithm or a model for tools like SAT, MILP and CP.

4.1.2 How Tagada works

In [LDLS21], a generic graph representation of ciphers was proposed. This graph is encoded in a text format to be able to simplify the communication of the cipher definition. Tagada uses this graph to generate MiniZinc models to solve the first step of the differential analysis, and we will use the same graph to generate models for the second step too.

DAG: unifying description of ciphers. The input graph is a Directed Acyclic graph where the nodes correspond to all the cipher parameters (inputs, outputs, constants, ...) and operators (XOR, S-Box, permutations, ...). The edges of the DAG link the operators to the parameters. Hence the DAG is a bipartite graph. Figure 4.1 shows the DAG of a 2-round toy example Feistel cipher.

The text format used to define the DAG is a JSON composed of a list of three types of objects.

— The variables (parameters) with their domain ranges.

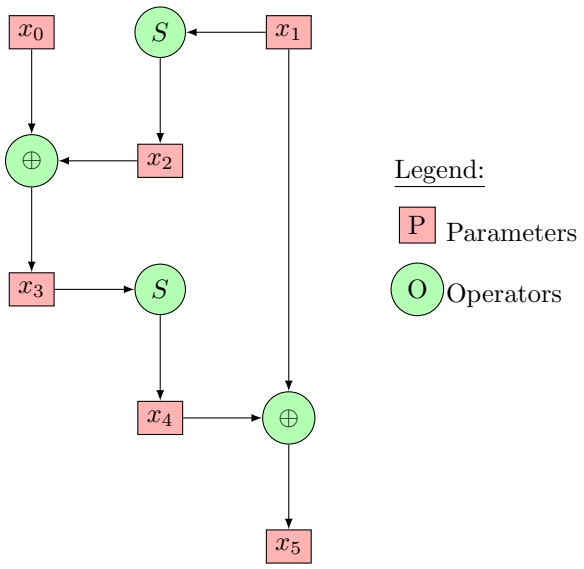


Figure 4.1 – DAG of a simple example 2-round toy Feistel cipher.

— The functions (operators) with input domains, output domains and specificities. For example, an S-Box will declare its lookup table, a LFSR will declare its length, shift direction and feedback polynomial, ...
— The transitions are triplets composed of a list of variables, a function and another list of variables. They describe the link between the operators and their input and output variables.

The advantage of a text format like JSON is that it is easy to generate and parse for any language. Moreover, the DAG has to be made only once for each cipher, thus saving a lot of time compared to the development of a hand-made algorithm or a solver-specific model. The difficulty of the DAG representation is that the DAG must be able to represent all the operators in order to be able to model any cipher. TAGADA currently handles the following operators: equality, bit-wise XOR, Galois field multiplication, LFSR, left shift register, right shift register, permutation, concatenation, split and S-Box. TAGADA also proposes to describe new operators by means of tables describing all the possible in/out tuples. However, this option is possible only when the table is not too large.

Truncated differential graph and optimizations. To solve the first step of the differential analysis, TAGADA first builds a truncated version of the graph of the input cipher. Then, this graph is optimized with a simplification of the useless parts of the graph. Indeed, differential analysis does not use constants nodes, and equality operators

can be removed from the graph by merging the equal nodes.

A second optimization is done to detect some inconsistent differential trails by adding constraints, as illustrated in Example 4.1.

Example 4.1 *Let $\delta_1, \delta_2, \delta_3$ be three differential variables, and $\Delta_1, \Delta_2, \Delta_3$ be their corresponding truncated differential variables. Let $\Delta_1 + \Delta_2 = 0$ and $\Delta_1 + \Delta_2 + \Delta_3 = 0$ be two equations generated from the DAG.*

If we look at the first equation, it is satisfied if $\Delta_1 = 0$ and $\Delta_2 = 0$. However, the difference can also be cancelled if they have the same value ($\delta_1 = \delta_2$). Therefore, the equation is also satisfied if $\Delta_1 = 1$ and $\Delta_2 = 1$.

For the same reason, the second equation accepts the solution $\Delta_1 = 1, \Delta_2 = 1, \Delta_3 = 1$. In the truncated model, there is no problem. However, in the second step model, the first equation implies that $\delta_1 = \delta_2$ and that they are equals to a non-zero difference, so $\delta_3 \neq 0$ would never be a valid assignment.

To detect this kind of inconsistencies, Tagada combines XOR equations to generate new equations. Generating new equations is a key point for an efficient model. These equations were hand-made in [GMS16, GLMS20], whereas they are automatically derived from the DAG in Tagada. In [RS20], an abstract-XOR constraint has been designed to better propagate XOR constraint in CP solvers.

Once optimized, a mathematical model is automatically generated from the truncated graph. This model is expressed using the MiniZinc language, which is a high-level language for defining constraint satisfaction problems [NSB+07]. Many different solvers are able to solve problems defined in MiniZinc such as, for example, Choco, Chuffed or Picat. We may also use Picat to automatically generate SAT or MILP models from a MiniZinc model, thus allowing one to use SAT or MILP solvers. In many cases, the best solver is Picat-SAT.

4.1.3 First step results

In [LDLS21] the Tagada models were able to recover the state-of-the-art truncated trails of the ciphers `AES`, `Midori`, `SKINNY`, and `Craft` in either single-key or related-key scenarios.

However, truncated trails may not lead to valid differential trails. To be able to make a strong statement on the differential characteristics of a cipher, we must try to instantiate

these trails with the second step. Therefore, another program or model has to be made to solve the second step, and we propose to generate it from the DAG representation of the cipher like Tagada generated the first step models.

4.2 Model generation for the second step

In this section, we present our contribution to the Tagada project, focusing on the modelling of the second step of the differential analysis, the instantiation of truncated characteristics. Contrary to the first step, we rely only on a CP solver, namely Choco [PF22]. There are two reasons for that. Firstly, in both [GMS16] and [DDH+21], the CP solver was said to perform very well for this particular problem. Secondly, we wanted to develop dedicated filtering algorithms for operators like the bit-wise XOR in a CP solver to improve the overall efficiency of the solving process.

In the second step of the differential analysis, we use the solutions of the first step and we try to instantiate them. More precisely, we make a complete model of the cipher, and we constrain the S-Box variables according to the truncated trail *i.e.,* the active S-Boxes in the truncated trail have a full domain, and the inactive S-Boxes be set to zero. To generate models for the second step, we need constraints for each operator and the most important one is the S-Box.

4.2.1 Modelling DDT with table constraints

The probability of the propagation of a differences through an S-Box is described in a *Difference Distribution Table* (DDT). For example, the DDT of the first S-Box of the F function of DES is given in Table 4.1. This S-Box has six input bits and four output bits. Therefore, the DDT is a table with $2^6 \times 2^4$ entries. In this table, we can see that the first difference (0) always propagates to 0 (64 times over the 64 possible input pairs of difference 0). In the second line, we can see that the input difference 1 propagates to the output difference 3 six times over 64 possible input pairs of difference 1. Therefore, the probability of this propagation is 6×2^{-6}. In the truncated model, we would use only the best probability of this table which is 16×2^{-6}, but this probability holds only for one transition $(34 \rightarrow 2)$.

DDT to table constraint. The main advantage of the CP solver is that we can directly model the DDT with a table constraint. A table constraint constrains a list of n variables

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	**64**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	**6**	0	2	4	4	0	10	12	4	10	**6**	2	4
2	0	0	0	8	0	4	4	4	0	6	8	6	12	6	4	2
3	14	4	2	2	10	6	4	2	6	4	4	0	2	2	2	0
4	0	0	0	6	0	10	10	6	0	4	6	4	2	8	6	2
5	4	8	6	2	2	4	4	2	0	4	4	0	12	2	4	6
6	0	4	2	4	8	2	6	2	8	4	4	2	4	2	0	12
7	2	4	10	4	0	4	8	4	2	4	8	2	2	2	4	4
8	0	0	0	12	0	8	8	4	0	6	2	8	8	2	2	4
9	10	2	4	0	2	4	6	0	2	2	8	0	10	0	2	12
A	0	8	6	2	2	8	6	0	6	4	6	0	4	0	2	10
B	2	4	0	10	2	2	4	0	2	6	2	6	6	4	2	12
C	0	0	0	8	0	6	6	0	0	6	6	4	6	6	14	2
D	6	6	4	8	4	8	2	6	0	6	4	6	0	2	0	2
E	0	4	8	8	6	6	4	0	6	6	4	0	0	4	0	8
F	2	0	2	4	4	6	4	2	4	8	2	2	2	6	8	8

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
10	0	0	0	0	0	0	2	14	0	6	6	12	4	6	8	6
11	6	8	2	4	6	4	8	6	4	0	6	6	0	4	0	0
12	0	8	4	2	6	6	4	6	6	4	2	6	6	0	4	0
13	2	4	4	6	2	0	4	6	2	0	6	8	4	6	4	6
14	0	8	8	0	10	0	4	2	8	2	2	4	4	8	4	0
15	0	4	6	4	2	2	4	10	6	2	0	10	0	4	6	4
16	0	8	10	8	0	2	2	6	10	2	0	2	0	6	2	6
17	4	4	6	0	10	6	0	2	4	4	4	6	6	6	2	0
18	0	6	6	0	8	4	2	2	2	4	6	8	6	6	2	2
19	2	6	2	4	0	8	4	6	10	4	0	4	2	8	4	0
1A	0	6	4	0	4	6	6	6	6	2	2	0	4	4	6	8
1B	4	4	2	4	10	6	6	4	6	2	2	4	2	2	4	2
1C	0	10	10	6	6	0	0	12	6	4	0	0	2	4	4	0
1D	4	2	4	0	8	0	0	2	10	0	2	6	6	6	14	0
1E	0	2	6	0	14	2	0	0	6	4	10	8	2	2	6	2
1F	2	4	10	6	2	2	2	8	6	8	0	0	0	4	6	4

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
20	0	0	0	10	0	12	8	2	0	6	4	4	4	2	0	12
21	0	4	2	4	4	8	10	0	4	4	10	0	4	0	2	8
22	10	4	6	2	2	8	2	2	2	2	6	0	4	0	4	10
23	0	4	4	8	0	2	6	0	6	6	2	10	2	4	0	10
24	12	0	0	2	2	2	2	0	14	14	2	0	2	6	2	4
25	6	4	4	12	4	4	4	10	2	2	2	0	4	2	2	2
26	0	0	4	10	10	10	2	4	0	4	6	4	4	4	2	0
27	10	4	2	0	2	4	2	0	4	8	0	4	8	8	4	4
28	12	2	2	8	2	6	12	0	0	2	6	0	4	0	6	2
29	4	2	2	10	0	2	4	0	0	14	10	2	4	6	0	4
2A	4	2	4	6	0	2	8	2	2	14	2	6	2	6	2	2
2B	12	2	2	2	4	6	6	2	0	2	6	2	6	0	8	4
2C	4	2	2	4	0	2	10	4	2	2	4	8	8	4	2	6
2D	6	2	6	2	8	4	4	4	2	4	6	0	8	2	0	6
2E	6	6	2	2	0	2	4	6	4	0	6	2	12	2	6	4
2F	2	2	2	2	2	6	8	8	2	4	4	6	8	2	4	2

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
30	0	4	6	0	12	6	2	2	8	2	4	4	6	2	2	4
31	4	8	2	10	2	2	2	2	6	0	0	2	2	4	10	8
32	4	2	6	4	4	2	2	4	6	6	4	8	2	2	8	0
33	4	4	6	2	10	8	4	2	4	0	2	2	4	6	2	4
34	0	8	**16**	6	2	0	0	12	6	0	0	0	0	8	0	6
35	2	2	4	0	8	0	0	0	14	4	6	8	0	2	14	0
36	2	6	2	2	8	0	2	2	4	2	6	8	6	4	10	0
37	2	2	12	4	2	4	4	10	4	4	2	6	0	2	2	4
38	0	6	2	2	2	0	2	2	4	6	4	4	4	6	10	10
39	6	2	2	4	12	6	4	8	4	0	2	4	2	4	4	0
3A	6	4	6	4	6	8	0	6	2	2	6	2	2	6	4	0
3B	2	6	4	0	0	2	4	6	4	6	8	6	4	4	6	2
3C	0	10	4	0	12	0	4	2	6	0	4	12	4	4	2	0
3D	0	8	6	2	2	6	0	8	4	4	0	4	0	12	4	4
3E	4	8	2	2	2	4	4	14	4	2	0	2	0	8	4	4
3F	4	8	4	2	4	0	2	4	4	2	4	8	8	6	2	2

Table 4.1 – DDT of the first S-Box of DES

to be instantiated to an n-tuple chosen within a collection of all valid n-tuples. Any constraint can be declared in *extension i.e.,* declared as a table constraint. However, this might not be the best way to model a constraint, especially if there are a lot of tuples. For example, to constrain three Boolean variables a, b, and c to have a sum equal to 2, we may define the table constraint $(a, b, c) \in \{(1, 1, 0), (1, 0, 1), (0, 1, 1)\}$ but the same sum with integer variables (with domains like $[\![-1000, 1000]\!]$) would require too many tuples. This is why constraints are often best declared in *intention i.e.,* with a dedicated filtering algorithm. However, an efficient filtering algorithm may not always be available.

Table filtering. The table constraint not only requires a filtering algorithm but also needs to be cautious of the data structure used to store and manipulate the table. This is because the memory used by the table depends on the number of tuples. In the literature [LS06, GJMN07, Ull07, Lec11, LLY12, MHD14], a lot of work has been done to optimize this algorithm for various situations (positive tables, binary tables, big or small tables, ...). The general idea is to pre-compute and try to maintain a set of indexes of valid tuples. When a tuple is no longer valid, an efficient search method will search for another valid tuple in the table. If there are no more valid tuples, the constraint is violated.

There are a lot of variations depending on the situation.

The DDT is an extensive definition of all the possible transitions of a difference through an S-Box with its probability and it is usually not possible to define these transitions in intention, by means of a small number of arithmetic constraints. Therefore, the table constraint is the most suited way to model it. In CP, this takes the form of a table T composed of a list of tuples $(\delta x_{in}, \delta x_{out}, p) \in T$ where δx_{in} and δx_{out} are the input and output variables of the S-Box, respectively, and p is a variable which corresponds to the probability of observing the output difference δx_{out} given the input difference δx_{in}. To avoid rounding errors, the probability is replaced by the negation of its base 2 logarithm (and probability multiplications are replaced with additions).

For MILP and SAT, this table would require a lot more intermediate variables and constraints [Udo21].

4.2.2 Modelling other operators

Unfortunately, the other operators are not available in CP solvers except for some exceptions, like the modular addition. To model the other operators, we could also use table constraints. However, tables are often too large to be efficient. Therefore, we will develop new filtering algorithms. In particular for the bit-wise XOR operator because it is used everywhere.

Bit-wise XOR. We first consider the bit-wise XOR in the case of three variables: $a + b = c$. Note that if there is less than one variable among a, b, c then the XOR is constant. If there are two variables (for example if c is constant), the XOR can be replaced with an equality operator. For the three variable case, we used some previous work from [DDH+21]. The filtering algorithm is not very smart. To filter the values of D_c, the domain of c, the algorithm computes a set that contains all the possible XORs between the values of the domains of a and b. This set is then used to remove the inconsistent values from the domain of c. The algorithm is given in Algorithm 7.

This algorithm has two weaknesses. First, the computing of the set is time-consuming but most of all, the set can reach the maximum size very fast. Indeed, if a, b and c have the same domain sizes, for example, 8 bits, then this algorithm will compute $2^8 \times 2^8$ XORs but will fill a set of maximum 2^8 values. In practice, this filtering algorithm takes a lot of time to build the set that does not filter anything in most cases. Therefore, this filtering is only performed in some cases decided by a chosen condition. In [DDH+21], this condition

Algorithm 7: 3-variable XOR filtering algorithm

```
   Input: IntVar a, IntVar b, IntVar c: the target domain to filter
1  set ← ∅;
   // Loop through possible values
2  for all v1 ∈ D_a do
3  |  for all v2 ∈ D_b do
4  |  |  set ← set ∪ {v1 ⊕ v2};
5  |  |  if set contains all possible values in variable domains then
6  |  |  |  return ;
7  D_c ← D_c ∩ set;
```

is that the sum of the domains of a and b is lower than the maximal domain size of c. We will see later that this condition is not the best one.

Bit-wise XOR of arbitrary arity. To model a XOR of higher arity with only three variable XOR constraints, we need to introduce intermediate variables and declare additional XORs. For Tagada, we wanted to avoid the introduction of new variables, so we extended the idea of this algorithm to a XOR of arbitrary arity. The algorithm uses a recursive loop to compute the set of all the possible XORs between the values of $n-1$ domains to filter the target domain. The algorithm is depicted in Algorithm 8. This algorithm is less efficient to constrain a 3-variable XOR than the previous propagator but the more variables we have, the more efficient this algorithm becomes compared to a decomposition with 3-variable XORs constraints and new intermediate variables (more than five variables XORs decompositions gives slower models). However, as we will see at the end of this section, the chosen condition to activate the filtering was also a problem.

Operations in the Galois Field. In cryptography, we sometimes use addition and multiplication in some fields. For CP modelling, operations like the MixColumns of `AES` have often been modelled with table constraints [MSR14]. For the modular addition, the modulo constraint exists in CP solvers. To model the modular addition $a \boxplus b = c \mod m$, we need one intermediate variable x and the two constraints $a + b = x$ and $x \mod m = c$. Unfortunately, modelling the multiplication is not possible with existing constraints. Like the XOR, we must make a new filtering algorithm for multiplication and division in a finite field. The filtering algorithm we made follows the same idea of Algorithm 7. We create a set of all the possible values of c from the domains of a and b except that in

Algorithm 8: n-variable XOR filtering algorithm

```
   Input: int target: index in vars table of the domain to filter
 1 Function combiXor(target, current, x):
       // skip target
 2     if current == target then
 3         combiXor(target, current + 1, x);
 4     else
           // add the value x to the set of values
 5         if current == vars.length - 1
 6               or (current + 1 == target
 7               and current + 1 == vars.length - 1) then
 8             for all v ∈ D_current do
 9                 set ← set ∪ {x ⊕ v};
10         else
               // Loop through domain with recursion
11             for all v ∈ D_current do
12                 combiXor(target, current + 1, x ⊕ v);
13 set ← ∅;
14 combiXor(target, 0, 0);
15 D_target ← D_target ∩ set;
```

the line $\texttt{set} \leftarrow \texttt{set} \cup \{v1 \oplus v2\}$, the $\oplus$ is replaced by ProdGF($v1$,$v2$) or DivGF($v1$,$v2$) where ProdGF and DivGF are algorithms to perform the modular product and division depending on the context. In real ciphers, the product is usually between a variable and a constant, so the algorithm is more likely to filter properly.

LFSR. The last operator we added is the LFSR. Similarly to the product, we replace the line 4 of Algorithm 7 with a function that can compute the next step of the LFSR. The LFSR could be decomposed into several variables and a XOR, but we would need to use *concat* and *split* constraints to divide an integer variable into the corresponding array of bit variables. The constraints *concat* and *split* are currently modelled with tables in Tagada.

Filtering efficiency. As stated before, the filtering of bit-wise XOR constraints needs a parameter to avoid useless computations. During our tests, we observed that the trade-off between the time gained from filtering and the time taken by the filtering algorithm is in

favour of less filtering. To be efficient, we must find at which domain size we would have a good chance to filter. For the XOR with two variables, we can write it as follows. Let a, b be two variables, D_a and D_b, their domain of max sizes n. Let c be a constant in the constraint $a + b = c$. In this case, we can filter the values in D_b if this set of values is not contained in D_a. For two variables, this condition is very probable. However, if now c is also a variable with a domain D_c of max size n, then the filtering is possible if the XORs of all the possible values of D_a and D_b is a set that does not include D_c. The number of values from the XORs is $\#D_a \times \#D_b$, and each value is in the domain range of D_c. In the end, it is like if we pick at random $\#D_a \times \#D_b$ values in the range of $[\![0, n]\!]$ and hope that we did not pick all the values of D_c. For the XOR with more variables ($\bigoplus_i x_i$), the number of values is $\prod_{i \neq j} \#D_{x_i}$. As a consequence, the probability of being able to filter some values of D_j is very low. To test the filtering efficiency of our XOR constraints, we implemented a *forward-checking* version of the filtering algorithms. The forward-checking (FC) method only filters when all the variables are fixed except one. In some early tests, we saw that the FC version was performing nearly as well as the full filtering algorithms. This means that the filtering algorithms for the 3-variable and n-variable XOR constraints are not helping the solving process that much. Therefore, we deduce that the CP model's strength is the DDT's table constraint. Moreover, the new dedicated filtering algorithms are still helpful because we would have to use table constraints instead, so they at least reduce the memory used by the model. A list of the operators we added to the model generator is depicted in Table 4.2.

4.3 Connect the two steps

At the beginning, the first step of Tagada was not designed to work with the second step but only to find the best truncated differential trail. While it can be possible to only use truncated differential trails optimizing, the whole process can be more efficient than only optimizing the two steps separately. To do so we improve the linking algorithm of [RGMS22] by splitting the first step search in three parts.

Step1-opt. The aim of Step1-opt is to find the optimal truncated differential trail. We define its signature with Signature 1.

Linear Operators			
Operator	Name	First step support	Second step Implementation
$=$	Equal	✓	Native support
`LFSR`	Linear Feedback Shift Register	✓	Custom filtering algorithm
$AB \to (A, B)$	Split	✓	Constraint table (native)
$(A, B) \to AB$	Concat	✓	Constraint table (native)
$\ll$ or $\gg$	Left (Right) Shift	✓	Custom filtering algorithm
$\lll$ or $\ggg$	Left (Right) Circular Shift	✓	Custom filtering algorithm
$\&_K$	Bitwise AND with Constant	✓	Constraint table (native)
$\|_K$	Bitwise OR with Constant	✓	Constraint table (native)
$\oplus$	N-ary Bitwise XOR	✓	Custom filtering algorithm or decomposition (for n-ary equations)
$\otimes_K$	Galois Field Multiplication with Constant	✓	Custom filtering algorithm
$\odot_K$	Galois Field Matrix Multiplication with Constant Matrix	✓	Decomposition and delegation to the $\otimes_K$ and $\oplus$ operators
`T`	Linear Lookup Table	✓	Constraint table (native)
Non-linear Operators			
DDT	Differential Distribution Table	✓	Constraint table (native)

Table 4.2 – List of supported operators in Tagada (both first and second steps).

Signature 1 Step1-Opt

Input:

G_Δ: the Differential Graph of the cipher

seen: the set of all the already found solutions

UB: the current upper bound

Output:

sol: the Truncated Differential Trail with the highest probability such as $P(sol) \leq UB$ and *sol* not in *seen*. If no such solution exists, returns `null`.

Step1-next. Instead of looking for an optimal solution Step1-next (Signature 2) is designed to find one truncated differential trail with a given upper bound (UB). While Step1-opt solves an optimization problem, Step1-next solves a satisfaction problem.

Usually solving optimization problems is more complicated than solving a satisfaction problem. Indeed for both optimization and satisfaction problems we have to find a solution but for optimization problems we also need to find the best one which is generally done by finding a sequence of solutions of increasing quality and proving that there is no better solution than the last one found.

To improve the overall time of the two steps algorithm (Algorithm 9) we try to use the Step1-next method as much as possible instead of the Step1-opt method. Step1-opt is only called at the beginning of the function when we have to compute the upper bound. The second call of Step1-opt is done when we have iterated over all the solutions that

Signature 2 STEP1-NEXT

Input:

G_Δ: the Differential Graph of the cipher
seen: the set of all the already found solutions
UB: the current upper bound

Output:

sol: a Truncated Differential Trail such as $P(sol) \leq UB$ and *sol* not in *seen*. If no such solution exists, returns `null`.

reach the current upper bound.

Step1-next-possible-UB (Signature 3). As said previously, step1-opt is the function that consumes the most calculation time. When we have iterated over all the solutions of a current upper bound we need to find the next upper bound. The search of the next upper bound is performed by another call to Step1-opt. However it is possible in some cases to bypass the function by using a new function Step1-next-possible-UB. The purpose of Step1-next-possible-UB is to find very quickly a lower approximation of the next upper bound. If this approximation is equal or lower than the current lower bound then we can stop the search without doing any more computations.

Example 4.2 *Let us take the example of the 4-round* `Rijndael-128-128` *instance. The step1-opt finds a truncated differential trail with an upper bound probability of* 2^{-72}*. For this trail the second step will find a valid differential trail of probability* 2^{-75} *and set it as the current lower bound. As* $2^{-72} > 2^{-75}$ *we need to find another truncated differential trail that matches* 2^{-72}*. For this cipher we only have one truncated differential trail of this probability. As we have a gap between the two probabilities we need to tighten the bounds which is done by decreasing the upper bound. For* `Rijndael` *all the S-Boxes are the same and their maximum probability is* 2^{-6} *(a trail of* 2^{-72} *is composed of* 12×2^{-6} *S-Boxes). In our case, the only way to find a new trail is to activate another S-Box, in that case the probability will be at least of* $13 \times 2^{-6} = 2^{-78}$ *which is lower than* 2^{-75}*. This simple computation saves one call of Step1-opt.*

4.4 Second Step Optimizations

To gain some generic solving efficiency, we propose three optimizations.

Signature 3 STEP1-NEXT-POSSIBLE-UB
Input:
G_Δ: the Differential Graph of the cipher
UB: the current upper bound
Output:
UB': an approximation of the next reachable UB.

4.4.1 Heuristics

Another way to improve the search speed is to use heuristics. Constraint Programming solvers usually use two kind of heuristics, a value heuristic and a variable heuristic. As their names suggest, the value heuristic is responsible of selecting the next value to test for a variable and the variable heuristic is used to select the next variable on which to branch. By default, generic solvers propose known general purpose search heuristics for both value and variable heuristics, but when we have extra information on a specific problem we can help the solver by designing custom heuristics.

We propose a custom value heuristic adapted to our second step. In this step, we want to maximize the probability of the differential trail. As the probability only depends on non-linear operators we can focus our heuristics on those operators. For each non-linear operator we have three available variables:

— δx which is the input differential variable
— δsx which is the output differential variable
— p which is the probability of the transition $\delta x \to \delta sx$

If the solver branches on a p variable, we only have to select the highest probability available. When the solver branches on a δx variable, if its δsx variable is instantiated, *i.e.,* it has only one possible value, then we can select the value that maximizes the transition to δsx, more formally:

$$\texttt{next-value}(\delta x) = \underset{v \in \texttt{dom}(\delta x)}{\operatorname{argmax}} P(v \to \texttt{value}(\delta sx))$$

When the solver branches on a δsx we can perform the same computation with its corresponding δx variable.

Algorithm 9: TWOSTEP(G_Δ, G_δ)

```
Input:
  G_Δ: step1 model
  G_δ: step2 model
1  LB ← 0
2  UB ← 1
3  best ← null
4  sol_1 ← STEP1-OPT(G_Δ, seen, UB)
5  seen ← {}
6  UB ← P(sol_1)
7  while LB < UB do
8      seen ← seen ∪ {sol_1}
9      sol_2 ← STEP2(G_δ, sol_1,LB)
10     if LB < UB then
11         sol_1 ← STEP1-NEXT(G_Δ, seen, UB)
12         if sol_1 is null then
13             UB ← STEP1-NEXT-POSSIBLE-UB(G_Δ, seen, UB)
14             if LB ≥ UB then
15                 break
16             sol_1 ← STEP1-OPT(G_Δ, seen, UB)
17         UB ← P(sol_1)
18 return best
```

4.4.2 DAG simplification

To make the model more efficient, we would like to add a graph simplification algorithm. From the first step solution, we know which S-Boxes are active and which are not. Therefore, we can simplify the model by removing the variables in the inactive part of the graph and all the related constraints. For example, let G be a graph composed of two S-Boxes $S1$ and $S2$, their output variables out_1 and out_2 and one XOR operation $out_1 \oplus out_2 = out_3$ (see Figure 4.2). If the truncated trail says that only the first S-Box is active, then instead of fixing the domain of out_2 to 0, we can remove the $S2$ and out_2 nodes from the graph. After this, we can further simplify the graph by removing the XOR node and the out_1 variable. In the end, we only need to keep $S1$ and out_3.

The simplification is split into two parts. The first part propagates from the active S-Boxes, all the nodes that can or will have a difference. In some cases, we can be more precise. In particular, we know that the difference always propagates through unary op-

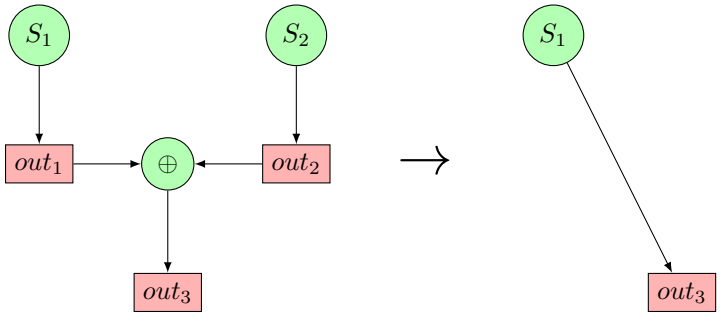


Figure 4.2 – Step2 graph shaving example

erators, so we can propagate this information in the graph. This information can then be used in the CP model. When we know that a variable necessarily has a difference, we can remove the value 0 from the variable domain when we declare it. We start with a graph containing all nodes with an "uncertain" marker. For each input and output variable of the S-Boxes, we use the information of the truncated trail to fix them to "active" or "inactive". For the linear operators, we can deduce the following *status propagation rules*:

— If there is only one "uncertain" variable in its input or output variables and all the other variables are inactive, it can be set to "inactive".
— If there is only one variable "active" and all the others are inactive, this is an error. This is not possible if the truncated trail is correct.
— If there is only one "active" variable and one "uncertain" variable, they both must be "active".

The "active" and "inactive" information is propagated iteratively in the graph until reaching a fixed point where no further propagation is possible. After these statuses are propagated, we can reduce the graph by removing all the "inactive" variables and all the operators only linked to them. We can also replace the XOR operators that have one inactive variable with equality operators, and finally, we can remove the equality operators and merge their input and output nodes. The simplified graph is then transformed into the CP model.

If we give the complete graph to the CP solver, it would eventually reach the same conclusion and set all the inactive variables to 0. However, this simplification is easy to do in advance and removing useless variables and constraints in a model is always a good idea.

4.4.3 Competitive parallel solving

In the previous CP model for the second step on SKINNY [DDH+21], a parallel method was used. Each model from the first step was launched on a separate thread, and the best solution found was shared with all the other threads. This new bound is added to the models that remain to be solved, and this information can be used to cut a large part of their search space.

In Tagada, we added a similar parallel competition between the models to solve the second step models from a list of truncated trails. In this setup, Tagada was able to recover the same results with similar solving times than the hand-made model of [DDH+21]. The generated models were only two to three times slower than the hand-made models.

An interesting future work would be to study the feasibility of parallel computing in the two-step method (Algorithm 9).

4.5 Results

We have implemented the model generator in Java and Kotlin to communicate more easily with the Choco solver. This generator can parse the JSON file of the Tagada DAGs and an associated file for the truncated trails. The generator then builds a CP model and calls the Choco solver to find the differential characteristics. The second step model generator can be used with the first step of Tagada in the two-step solving algorithm depicted in Algorithm 9, or it can also be used alone in a parallel setup if we can give a truncated trails list as input.

To compare the generated models, we reproduced the results of ad-hoc models with Tagada on the two-step differential analysis. We set a time limit of one day. The computation was done on a Debian GNU/Linux 11 (bullseye) x86_64 sever with two Intel Xeon Gold 6254 (3.10 to 4.00 GHz) processors and 64MB of RAM. Each instance was launched on a single core. We were able to reproduce the results shown in Table 4.3.

The second step code is available in [1] and will be available in the Tagada library [2]

1. https://drive.google.com/file/d/1EJGRmMZuAgZ4OznmHca6jkzQamA3ZMzp/view?usp=drive_link
2. https://gitlab.limos.fr/iia_lorouque/tadaga-step2

4.6 Perspectives

We have presented a model generator for the second step of the differential analysis that relies on the DAG representation of the model generator of the first step Tagada. We have shown that these generated models can recover the results of state-of-the-art ad-hoc models in reasonable times.

The idea of a simple tool to perform differential analysis is interesting, and other recent works are also working in this direction [RR22, BGG+23a]. Moreover, we think that a unified cipher format would help the comparison and development of these tools. Tagada could be improved by integrating previous solving methods dedicated to ARX ciphers [Leu12] and bit-based ciphers [Köl]. In these cases, more work is needed to determine if the CP solver is still the best solving tool. In the end, Tagada could be extended to search for the variants of differential analysis (boomerangs, impossible differentials, ...).

Cipher	Max Round	Probability	Reference
Midori-64	16	2^{-16}	[Gér18]
Midori-128	20	2^{-40}	[Gér18]
Warp	41	2^{-40}	[TB22]
Twine-80	18	2^{-64}	[SMS+20]
Twine-128	16	2^{-52}	[SMS+20]
Skinny-64-TK1	11	2^{-64}	[DDH+21]
Skinny-128-TK1	11	2^{-74}	[DDH+21]
Rijndael-128-128	5	2^{-105}	[GLMS20]
Rijndael-128-160	7	2^{-120}	[RGMS22]
Rijndael-128-192	9	2^{-146}	[GLMS20]
Rijndael-128-224	12	2^{-212}	[RGMS22]
Rijndael-128-256	14	2^{-146}	[GLMS20]
Rijndael-160-128	4	2^{-112}	[RGMS22]
Rijndael-160-160	6	2^{-138}	[RGMS22]
Rijndael-160-192	8	2^{-141}	[RGMS22]
Rijndael-160-224	9	2^{-190}	[RGMS22]
Rijndael-160-256	11	2^{-204}	[RGMS22]
Rijndael-192-128	3	2^{-54}	[RGMS22]
Rijndael-192-160	5	2^{-118}	[RGMS22]
Rijndael-192-192	7	2^{-153}	[RGMS22]
Rijndael-192-224	8	2^{-205}	[RGMS22]
Rijndael-192-256	9	2^{-179}	[RGMS22]
Rijndael-224-128	3	2^{-54}	[RGMS22]
Rijndael-224-160	4	2^{-122}	[RGMS22]
Rijndael-224-192	5	2^{-124}	[RGMS22]
Rijndael-224-224	7	2^{-196}	[RGMS22]
Rijndael-224-256	8	2^{-182}	[RGMS22]
Rijndael-256-128	3	2^{-54}	[RGMS22]
Rijndael-256-160	4	2^{-130}	[RGMS22]
Rijndael-256-192	5	2^{-148}	[RGMS22]
Rijndael-256-224	4	2^{-115}	[RGMS22]
Rijndael-256-256	6	2^{-128}	[RGMS22]

Table 4.3 – Best differential trails recovered with Tagada (time limit of one day).

Chapter 5

Explanations in Constraint Programming

Introduction

From the Latin word *explano*, explanation means to make plain and clear. However, the work described in this chapter uses a more specific definition. In CP solvers, an explanation is a set of events that describes the behaviour of a sub-part of a solver. This explanation can then be used by the solver to make better reasoning. Therefore explanations can increase the solving efficiency which is a welcome upgrade for solving the complex problems we encounter in cryptanalysis. We will discuss in the conclusion how the solver explanations can be related to human-readable explanations or machine-verifiable proofs.

Constraint programming (CP) is a flexible declarative language used to model and solve combinatorial problems. The advantage of CP over MILP and SAT is that it comes with a large collection of constraints. All the constraints can work together in a single model, and we can even define new ones if needed. This is useful when a problem can hardly be modelled with only linear equations or Boolean clauses. To solve a problem, CP solvers use the *filtering algorithms* of each constraint to reduce the search space. These algorithms are paired with a Depth-First Search (DFS) algorithm. The DFS builds a decision tree. Each leaf of this tree denotes either a solution or a conflict. When every variable of the problem has a value and all the constraints are satisfied, we have solution. When a variable has no more possible values, we have a conflict. If there is a conflict, the DFS algorithm backtracks to the previous decision to make a new branch and continues the search. One way to speed up the search is to reduce the number of conflicts. This idea was first introduced in the SAT solvers as the Conflict Driven Clause Learning algorithm (CDCL) [SS99, MLM09]. With this algorithm, the solver can learn from a conflict by searching for all the *events* that triggered it. This knowledge can be turned into a new constraint to add to the problem. The solving time to find the conflict may already be

spent, but this new constraint will hopefully prevent searching for a similar conflict. This idea was first implemented in SAT solvers with excellent results [SS96, MMZ+01, ES03], so the adaptation of this idea to CP solvers is interesting. However, SAT solvers use only one type of constraint, the Boolean clause. One first attempt to use CDCL in a CP solver was to make a Lazy Clause Generation solver. This is a CP solver paired with a SAT solver. The CP solver uses filtering algorithms, and the SAT solver only handles the CDCL algorithm. Later on, the CDCL algorithm was implemented in pure CP solvers. In both cases, each constraint of the solver must be *explained i.e.,* an explanation algorithm must be able to describe the filtering algorithms, which might not be straightforward for all of them. In addition, the global constraint catalog [BCDP07] references several hundreds of constraints. Therefore the key to the development of conflict analysis in CP solvers relies on explaining constraints.

In this chapter, we explore automatic explanation generation. We rely on the fact that most constraints can be reformulated as a conjunction of constraints called a decomposition. We can use it to generate an explanation of global constraints from the explanations of the decomposition.

Takeaway

This contribution supports the following conclusions:

- Constraint decomposition can be used to generate explanations for global constraints
- Our prototype can automatically define explanations for 14 global constraints.
- When added to CP solvers, the generated explanations are competitive with state-of-the-art explanations.

5.1 Background

Constraint programming was invented in the mid-70s ([Mon74, Lau78, Ros88]) to solve combinatorial problems. The first idea is very simple “find a value for each variable of a problem where constraints specify which values that cannot be used together”. Therefore, CP has been used to find solutions to various problems with various search algorithms and heuristics. Even some cryptographic problems were tackled in [Bur69]. The CP community

began to share these methods in the early 90s with the CP conference and the constraints journal. In 2005, an extensive list of constraints was proposed in the first version of the constraint catalog. In 2006, CP had an extensible definition in the Handbook of Constraint Programming [RvBW06].

5.1.1 CP modelling

A constraint satisfaction problem is defined with three items.

Variables. The variables $X = \{x_1, \ldots, x_n\}$ represent the unknowns of a problem for which we must find a value. They can be of any type, Boolean, integer, real [CDR98], set variables [Ger94], or even graph variables [DDD05], ...

For example, suppose the problem is how to plant vegetables in a garden. In that case, integer variables can represent the number of tomatoes, potatoes, and carrots, ... Real variables can represent their positions in the garden, and a graph variable can represent the watering pipe system.

Domains. The variable domains $D = \{D_{x_1}, \ldots, D_{x_n}\}$ contains all the possible values of each variable. They can be sets of values or bounds. In our example, the integer domains contain all the values of the available number of vegetables. The real domains contain the bounds of the gardens. The graph domain contains all the possible pipe configurations.

Constraints. The constraints $C = \{c_1, \ldots, c_m\}$ define the relations between variables. There can be simple constraints like arithmetic constraints (*e.g.,* $x_1 < x_2, x_3 \neq x_4$), or more complicated ones called global constraints [BH03, BCDP07]. The best known global constraint is AllDifferent($\{x_1, \ldots, x_n\}$). This constraint states that all the variables in $\{x_1, \ldots, x_n\}$ must have a different value. Each constraint must be paired with a filtering algorithm (also called a propagator) to remove the values of its variables that do not respect the constraint. The most basic filtering algorithm is to define every combination of valid values using a table. However, these tables have some limitations. When the tables are large, memory consumption issues may arrive. Therefore, it is interesting to define constraints with dedicated filtering algorithms. A very simple constraint like $x_1 < x_2$ will use a simple filtering algorithm (remove all the values of D_{x_1} that are greater than the largest value of D_{x_2} and conversely with the lowest values). On the contrary, a global constraint like AllDifferent has much more complex filtering algorithms [vH01]. There

is a large number of constraints (423 referenced with their filtering algorithms in the constraint catalog [1] in 2023).

In our example, if we want to have enough vegetables to eat, we can add an AtLeast [2] constraint on the number of vegetable variables. If we want to make sure that every plant has enough space, we can use a DiffN [3] constraint on the position variables *i.e.,* a constraint that prevents shapes from overlapping in a space. To control the proximity of certain vegetables that want to grow side by side or not, we can use Distance [4] constraints on the positions variables. The pipe system shape can be constrained with a graphCrossing [5] constraint to prevent pipes from crossing each other. Finally, we can adapt the garden with the seasons with a slidingTimeWindow [6] constraint that can prevent planting vegetables at the wrong season. All these very different constraints and their filtering algorithms will be used together in a model to solve our example garden problem.

5.1.2 CP solving

Filtering and backtracking. The idea of the backtracking algorithm originates in [Leh57, GB65, FW74, BR75]. This algorithm builds a search tree where each node contains a decision (a domain reduction for example) and verifies if the constraints can be satisfied. When a constraint is not satisfied, the algorithm backtracks to the previous decision to make a new branch by choosing another value. The algorithm is complete because it continues until it finds a solution or shows that there are none when all the branches are exhausted. Early backtracking algorithms spent a lot of time trying all combinations of inconsistent values (trashing). Constraint Programming was invented when filtering algorithms were merged within a backtracking algorithm to obtain consistency *i.e.,* at each decision, the domains only contain values that respect the constraints [Mac77]. The idea is to use filtering algorithms to remove the inconsistent values of the domains of the variables. The filtering algorithms of each constraint are called until no more values can be removed *i.e.,* it reaches a fixed point. Then a new decision is taken, and the filtering algorithms are called again. There are several levels of consistency. The most important one is called

1. https://sofdem.github.io/gccat/
2. https://sofdem.github.io/gccat/gccat/Catleast.html
3. https://sofdem.github.io/gccat/gccat/Cdiffn.html
4. https://sofdem.github.io/gccat/gccat/Call_min_dist.html
5. https://sofdem.github.io/gccat/gccat/Cgraph_crossing.html
6. https://sofdem.github.io/gccat/gccat/Csliding_time_window_sum.html

arc consistency. This consistency ensures that the filtering algorithms remove all the inconsistent values in the domains. Another important one is bound consistency, where the filtering process only ensures that the bounds of the domains respect the constraints. In some extreme cases, consistency does not need to be that strong. For example, the forward checking method (FC) does not call the filtering algorithms at every decision to save some computation time [HE79].

The complexity of a constraint programming solving process depends on the filtering algorithms complexity, the order of the decisions (the strategy) and the consistency. The filtering algorithm complexity can be computed, but the other two can only be estimated on specific problems. Therefore, it is impossible to give an exact complexity in general. However, several researches were conducted to speed up the search. For example, studies on the search showed that failing early is generally a good way to reduce the search space. Most importantly, backtracking can be smarter than simply undoing the previous decision. In [SS77, Gas78, Bru81, Dec90, Pro93, Gin93], backtracking was replaced by a backjumping method. When it reaches a conflict, the algorithm searches for a "culprit" decision in the tree and backjumps to it to explore the new branch. This idea went further with the introduction of nogoods in [SS77, SV94, RJL03].

A *nogood* is a set of decisions that is inconsistent with any solution. It is a redundant constraint that can be deduced when the decision tree fails. The cost of this failure is already paid, but the deduced nogood will hopefully prune the future search space. The nogoods were only compatible with a simple backtracking algorithm until [JDB00, Jus03] introduced the filtering consideration in the nogood generation. The idea is to record each filtering. When a failure occurs, we can then look at the records to generate the nogood. In this work, the authors noticed that the nogoods were useful except for the randomly generated problems. However, real problems are never random. In [KB05], the filtering is not recorded to make the nogoods. Instead, some rules are defined on each constraint to generate the nogood and we may call them explanation rules in the following. In this work, explanation rules were proposed for the AllDifferent, Range, and Roots constraints. Note that sometimes, there are multiple possible explanations to explain a conflict. In general, using the smallest one is most efficient for the solver. Note also that the explanation generation can be problematic because we can end up with an overflowing number of new constraints. To solve this issue, the old explanations are forgotten. Even if there were multiple definitions of nogoods in the literature, we can consider that they are early ideas of the recent explanations in [Stu10, VS10].

Of course, the idea of learning from the conflicts is not restricted to SAT and CP, and some work tried to implement it in MILP solvers with promising results [Ach07, BFS10, WBH17].

Alongside backjumping, Constraint Programming was upgraded over the years with better constraint propagation [SS08], branching strategy studies [LSHS13], views [ST13, HM14], ... One of the most important events for CP is probably the development of MiniZinc [NSB+07], a standard modelling language for CP that links several solvers. This common language and the MiniZinc challenges that provide a solver comparison on multiple problems made the use of CP easy even for non-CP users.

5.2 Related work on explanations for CP

Learning the reasons of a conflict has recently gained a rising interest in Constraint Programming (CP) and has been adapted to CP in the last decade [Stu10, VS10]. To explain a conflict during the solving algorithm, we use two important notions.

Event. An event stores a domain modification. It can be expressed by a unary constraint on the variable. For example $x = 3$ or $y < 7$. This modification is always a domain reduction. It may have been triggered by a filtering algorithm, or it can be a decision (for example a variable is assigned to a value).

Implication graph. The implication graph is a DAG that stores the events and their relationships regarding the constraints during the solving process.

Example 5.1 *Let x, y, z be three variables and c_1, c_2, c_3, three constraints. An implication graph can be seen in Figure 5.1. In this figure, the red square nodes are decision events, and the green circle ones are events caused by a constraint filtering algorithm. The constraint is labelled on the edges that led to the event.*

If an event of the implication graph is a conflict (for example, D'''_x), the clause learning algorithm will search for the responsible decisions in the implication graph. To do so, we go backwards in the implication graph until our backward tree has only decision events as leaves (initial domains are considered as decision events). A naive way to find the events is to consider that all the events involved are responsible [SS96]. In our example, a naive list of responsible events is (D_x,D_y,D'''_y,D'''_z). However, further research in [MMZ+01] showed

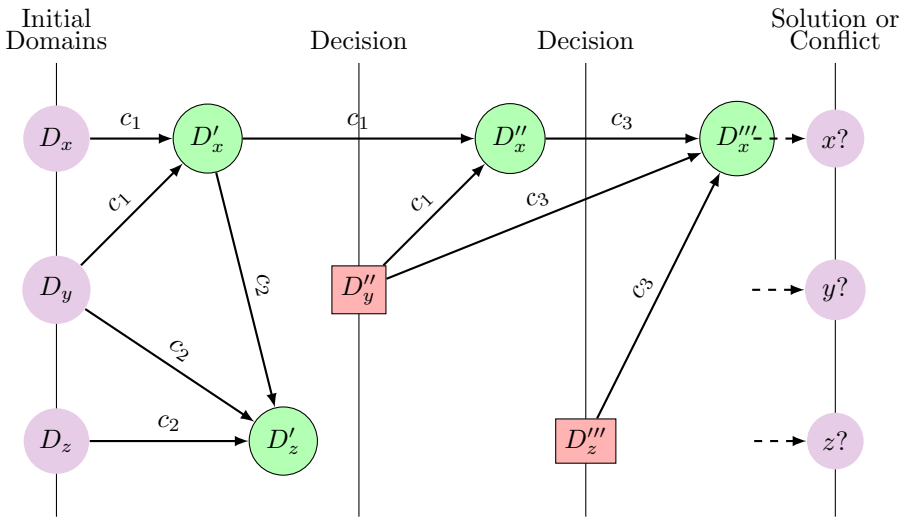


Figure 5.1 – Implication graph

that smaller clauses were more efficient. To find a smaller valid clause, the Conflict learning algorithm will search for an Unique Implication Points (UIP). UIP are mandatory nodes between the last decision and the conflict. Each UIP node can generate a clause, but we will usually use the last decision (1-UIP). In our example, the clause will be (D''_x, D''_y, D'''_z). This last decision is called the *articulation*, and it is usually the backjump node. In our example, it is D'''_z. From our naive list of responsible events, we have that $(D''_x \wedge D''_y \wedge D'''_z) \implies D'''_x$. Since we want to prevent this conflict, we will add the constraint $c_4 = \neg D''_x \vee \neg D''_y \vee \neg D'''_z$. This reasoning is correct if c_1, c_2, c_3 are Boolean clauses because all the variables of a Boolean clause are responsible for every event in the clause. However, CP has much more complex constraints, and the events may have more complex explanations. The naive clause (every event involved is responsible) is not incorrect for CP conflict learning, but it introduces useless events that make the constraint weak, *i.e.,* useless events will prevent the learnt clause from filtering efficiently. To find the responsible and only these ones, we need to explain how the constraints filtering algorithms did produce the events.

Explanation. An explanation is a Horn clause of events *i.e.,* it can be written as a disjunction of events that imply an event. A default explanation contains all the incoming events in the disjunction, but as we said earlier, it is not very helpful. Ideally, a stronger explanation will not mention irrelevant events in the disjunction. For example, the explanation of D'''_x by the constraint c_3 could be $D'''_z \implies D'''_x$, thus producing a much stronger constraint $c_4 = \neg D'''_z$ from this conflict. With this example, we can see the advantage

of conflict analysis over the backtracking algorithm alone. In another context, *i.e.,* another sub-part of the decision tree, the decision D_z''' will be tried again. With the learning algorithm, we learned that D_z''' will never be a valid event regardless of the rest of the problem.

Explanations are usually noted as follows:

$$\mathtt{R}\ \frac{e}{e_1 \vee \ldots \vee e_l}$$

Where R is the rule's name, e is an event, and $e_1 \vee \ldots \vee e_l$ is the clause of events that explains e. Note that explanations can be considered as the inverse implications of the filtering algorithms.

In CP, global constraints [BCDP07] are often used, either for efficiency or for modelling reasons. Global constraints are constraints with an arbitrary number of variables. They are more formally defined in [BH03]. Explaining global constraints requires integrating the sophisticated filtering rules of the constraint into a dedicated explanation. Some important constraints have been explained in [SFSW11, DFS12, FS14, SS16, GS20]. Considering that there are now hundreds of global constraints [BCDP07], providing dedicated explanations for each of them is a long and tedious task. Unfortunately, to apply conflict analysis, the solver needs *all* the constraints of the problem to be equipped with an explanation algorithm. In practice, having to explain global constraints restricts the development of explanation frameworks.

Lazy Clause Generation (LCG). LCG is a method for generating SAT clauses for CP solvers only when needed during the search [OSC07, OSC09, Stu10, Chu11, FSS13]. The idea is to use both a SAT solver and a CP solver with integer variables to combine the strength of learning and filtering. However, to model integer variables, the SAT solver needs a Boolean variable for each possible pair of integer variable and integer value. This would introduce too many Boolean variables in the SAT solver, so the lazy clause generation will generate only the needed Boolean variables on the fly. For example, if we have an integer variable with the domain $D_x = [\![1, 2000]\!]$, a SAT solver would need 2000 independent Boolean variables. The LCG solver will generate a Boolean variable only when the CP filtering algorithms reduce the domain or when a decision is taken. In our example, if the decision $x < 700$ is taken, a corresponding Boolean variable will be generated in the SAT solver. Therefore the SAT solver is light and has all the variables needed to perform CDCL on conflicts. This idea was implemented with the solver Chuffed. To make it as

efficient as possible, LCG is paired with a minimalist CP solver handling integer variables and ten commonly used global constraints and their explanations. In [DFS12], the LCG solver is compared with several standard CP solvers on the constraint AllDifferent with different propagators. In conclusion, the LCG solver is hugely beneficial and highlights the efficiency of explanations. However, if new constraints are added in Chuffed, they must also be explained.

Like every previous conflict learning algorithm, there is an issue when many clauses are learned. The solution for CP is usually to forget the oldest or the clauses that performed less filtering. One other advantage of conflict learning is that it can be helpful in a parallel-solving process [RM18]. The problem is run with a different strategy on each thread, and the learnt clauses are shared between them.

Constraint decomposition. Constraint decomposition is a good way to express a global constraint when a solver does not implement it. A constraint decomposition is a set of constraints that expresses a global constraint [BKNW09, BKN+09, Nar11]. For example, on the three integer variables x_1, x_2, x_3, the constraint AllDifferent(x_1, x_2, x_3) can be replaced by three inequality constraints, $x_1 \neq x_2$, $x_1 \neq x_3$, $x_2 \neq x_3$. Some decompositions may need to add intermediate variables. A common way to do this is to use constraint reification. A reified constraint is a constraint with is equivalent to a Boolean variable. If the constraint holds, the Boolean is true and conversely, for example, in $x < 7 \iff b$.

In [BH03], the authors defined three categories of global constraints:

— A constraint R is **semantically** global if there exists no constraint decomposition for R.

— A constraint R is **operationally** global if there exists no constraint decomposition with the same filtering quality.

— A constraint R is **algorithmically** global if there exists no constraint decomposition with the same filtering complexity.

For constraint programming solvers, constraint decomposition is very useful because this means that solvers does not need to implement all the constraints of the catalog [BCDP07]. The main example of this is the modelling language MiniZinc. When a solver is called to solve a MiniZinc model, all the constraints unknown to the solver will be decomposed, thus making a generic interface with most solvers. Most of the time, decompositions are slower than algorithmically, operationally and semantically global constraints. However, it was shown in LCG solvers that less efficient decompositions can be

more useful than global constraints without explanations [FS09].

5.3 New explanations from decompositions

In this section, we present a rewriting system to generate explanations from decompositions. First, we chose a minimalist decomposition language inspired by [Nar11]. It can also be easily expanded with new constraints.

Definition 5.1 (Decomposition language) *Let x be integer variables, b be Boolean literals, v be an integer value, and i be an integer index. The constraints of the decomposition language are:*

$$x = v \iff b, \qquad x \geq v \iff b,$$

$$\bigwedge_i b_i \iff b, \quad \sum_i b_i < v \iff b, \quad \bigvee_i b_i \iff b$$

Our goal is to use this decomposition language to generate explanation formulas for global constraints. This is a preliminary step for the development of a solver. The formula completes the filtering algorithm during the implementation of a constraint in a solver. In the end, the decomposition language will not be used in the solver, and the explanation formula should not be confused with an explanation of a conflict produced by the solver.

The explanation generation method is the following:

— Let C be a global constraint and D be its decomposition in our language. (multiple decompositions may be possible)

— For each possible type of event of C, deduce an explanation formula with D and a rewriting system.

— Finally, implement the formula as an explanation algorithm of C in a solver.

The formula operates on two types of terms. The first type of term represents the events of the global constraints. ($\mathbb{M}$ is a special set of indices defined later)

$$E = \{x_i = t,\ x_i \neq t,\ x_i < t,\ x_i \geq t \ \mid i \in \mathbb{M}, t \in \mathbb{M}\}$$

The second type of term represents the additional Boolean variables that are introduced with the decomposition *i.e.,* they do not exist in the global constraint.

$$\mathcal{B} = \{b_i^{\diamond} \mid i \in \mathbb{M},\ \diamond \in \{\texttt{optional name}\}\}$$

Each of these terms can be negated. For example $\neg b_i$ and $\neg(x_i = v) \iff (x_i \neq v)$. To make them work together, the decomposition language uses the same basic terms as the formula $(E,\ \mathcal{B})$. Our formula is defined in the following definition.

Definition 5.2 (formula) *A formula F is defined by the grammar,*

$$F ::= E \quad | \quad \mathcal{B} \quad | \quad F \wedge F \quad | \quad F \vee F \quad | \quad \top \quad | \quad \bot$$

In our formula, the term $\bot$ represents the inexplicable event, and $\top$ represents the always true event. From a starting event term E_0, we will use a rewriting system to find a formula F that contains the responsible events. We will then simplify the formula to find rules like $\mathtt{R}\ \frac{E_0}{\bigwedge_i E_i}$. For example, if we simplify the formula $(\top \vee E_1) \wedge (E_2 \vee E_3) \wedge E_4 \vee (E_5 \wedge \bot)$ we will produce two explanations: $\mathtt{R}^1\ \frac{E_0}{E_2 \wedge E_4}$, and $\mathtt{R}^2\ \frac{E_0}{E_3 \wedge E_4}$.

To see why we need a special set of indexes $\mathbb{M}$, we can look at the decomposition of the constraint Increasing. This constraint ensures that the variables $X = x_1, \ldots, x_n$ have strictly increasing values.

$$\textsc{Increasing}(X) \iff \begin{cases} (x_i \geq t) \iff b_{i,t} & \forall i \in [\![1, n]\!] \\ \neg b_{i-1,t} \vee b_{i,t} & \forall i \in [\![2, n]\!] \end{cases}$$

In the second equation, there is an $i - 1$ in the index. Therefore, the formula terms must be able to represent these indices relations. A more complex case is the sum where multiple events are responsible together. In this case, the formula may need a new index with a quantifier like $\forall j,\ j \neq i$. In the end, we identified the following index relations:

— $i \in S$, The index i belongs to the set of index value S
— $j \neq i,\ j < i,\ j > i$, j has a relation with another index i
— $j = i + n,\ j = i - n,\ j = i + d_i,\ j = i - d_i$,
j has an additive relation with i (n is an integer and d is an integer array).
— $\exists i, \forall i$, A new index is introduced with a quantifier.

There might be more relations to handle, and finding the most efficient way to represent the indices is left as future work.

5.4 Rewriting system

We propose an inference system that computes, from a logical expression representing a constraint in our decomposition language and an event type as defined in E, an explanation. The explanation will be a formula, but only on the event terms E appearing in the initial constraint. The additional Boolean terms $\mathcal{B}$ that may appear in the decomposition are removed in the process. Our method is purely symbolic. It is made off solver and only needs to be done once for each new constraint.

5.4.1 Intuition

Given an event, the rules are meant to deduce what happened that provoked this event. We describe the idea in three examples.

From the event term $E_0 = (x = t)$ in the constraint $x = t \iff b$, the only explanation for the event E_0 is b thus we can deduce the rewriting rule $\mathtt{R}\ \frac{b}{x=t}$.

Now consider the constraint $b_1 \vee b_2 \iff b_3$, and we want to explain b_1 *i.e.,* we want to know in which situation the variable b_1 will be set to true. The only possible reason why this event has been generated is the following situation: b_3 is true, and b_2 is false. Thus, we can deduce the rewriting rule: $\mathtt{R}\ \frac{b_1}{b_3 \wedge \neg b_2}$.

Finally, consider the sum constraint $b_1 + b_2 + b_3 \leq 2 \iff b_4$ and assume that we want to explain $\neg b_1$. The only situation where such an event can be generated is: b_4 is true, and both b_2 and b_3 are true. Thus we can deduce the rewriting rule: $\mathtt{R}\ \frac{\neg b_1}{b_4 \wedge b_2 \wedge b_3}$.

5.4.2 Rewriting algorithm

The rewriting rules explaining an event are derived from pseudo-Boolean logic. For each case, we deduce from the event the situation that made it happen, similarly as above.

Definition 5.3 (Rewriting rules) *For clarity purposes, the reified terms are labelled with the name of the constraint (*`eq`*,* `sum`*, ...). The first* 8 *rules are only for the reified events* $(x_i = t) \iff b_{it}^{\texttt{eq}}$ *and* $(x_i \geq t) \iff b_{it}^{\texttt{geq}}$*. They describe all the possible implications.*

$$R_{=}^{1}\ \frac{(x_i = t)}{b_{it}^{\texttt{eq}}} \quad R_{=}^{2}\ \frac{(x_i \neq t)}{\neg b_{it}^{\texttt{eq}}} \quad R_{=}^{3}\ \frac{b_{it}^{\texttt{eq}}}{(x_i = t)} \quad R_{=}^{4}\ \frac{\neg b_{it}^{\texttt{eq}}}{(x_i \neq t)}$$

$$R_{\geq}^{1}\ \frac{(x_i \geq t)}{b_{it}^{\texttt{geq}}} \quad R_{\geq}^{2}\ \frac{(x_i < t)}{\neg b_{it}^{\texttt{geq}}} \quad R_{\geq}^{3}\ \frac{b_{it}^{\texttt{geq}}}{(x_i \geq t)} \quad R_{\geq}^{4}\ \frac{\neg b_{it}^{\texttt{geq}}}{(x_i < t)}$$

In the following, the index i is implicitly in a set of indices I that describes the variables involved in the disjunction, conjunction or sum. Moreover, every $\forall i, \forall j$ and $\exists i$ are in the same set I.

Rules for the disjunction $\bigvee_i b_i \iff b^{or}$ are:

$$R^1_\vee \frac{b_i}{b_j | \forall j \wedge b^{or}} \qquad R^2_\vee \frac{\neg b_i}{\neg b^{or}} \qquad R^3_\vee \frac{b^{or}}{b_i | \exists i} \qquad R^4_\vee \frac{\neg b^{or}}{\neg b_i | \forall i}$$

Rules for the conjunction $\bigwedge_i b_i \iff b^{and}$ are:

$$R^1_\wedge \frac{b_i}{b^{and}} \qquad R^2_\wedge \frac{\neg b_i}{b_j | \forall j \wedge \neg b^{and}} \qquad R^3_\wedge \frac{b^{and}}{b_i | \forall i} \qquad R^4_\wedge \frac{\neg b^{and}}{\neg b_i | \exists i}$$

Rules for the sum $\sum_i b_i < v \iff b^{sum}$ are:

$$R^1_\Sigma \frac{b_i}{\neg b_j | \forall j \wedge \neg b^{sum}} \qquad R^2_\Sigma \frac{\neg b_i}{b_j | \forall j \wedge b^{sum}} \qquad R^3_\Sigma \frac{b^{sum}}{\neg b_i | \forall i} \qquad R^4_\Sigma \frac{\neg b^{sum}}{b_i | \forall i}$$

These rules are then used in a rewriting algorithm described in Algorithm 10. The algorithm starts with an event of the global constraint to explain and rewrites it to a formula. The algorithm stops when the formula is composed of terms involving only variables that are in the scope of the global constraint. Indeed, when the decomposition introduces additional variables, they must not appear in the formula since they have no existence within the global constraint. A call to PossibleRules(v, D) returns a set of rules such

Algorithm 10: Rewrite(v: term, F: formula, D: decomposition)

```
1  F_1 ← empty formula
2  for all r ∈ PossibleRules(v, D) do
3  |   F_2 ← empty formula
4  |   S_v ← set of terms replacing v according to rule r
5  |   for all v' ∈ S_v do
6  |   |   F_2 ← F_2 ∧ v'
7  |   |   if v' is not an event of the global constraint then
8  |   |   |   Rewrite(v', F_2)
9  |   F_1 ← F_1 ∨ F_2
10 Replace v by F_1 in F
```

that: v is the leftmost term of any of them, and no rule provides a term previously used to write the current term (v). The second condition ensures the termination of the algo-

rithm, enforcing that a modification event cannot be explained by itself. This resulting formula of Algorithm 10 is composed of terms, conjunctions and disjunctions. Depending on the rewriting rules, many terms may be created (lines 4-5). All of them are needed for the explanation and included in the formula with a logical conjunction ($\wedge$, line 6). When more than one rewriting rule is available (line 2), each of them may be part of the explanation. These sub-formulas are included in the formula with a logical disjunction ($\vee$, line 9). When there is no rewriting rule available, a *false* term ($\bot$) is written in replacement of v. A post-processing phase described in Algorithm 11 transforms the formula into a list of explanation clauses.

Algorithm 11: EXTRACT(F: formula)

```
1 if F is a term then
2 |   return a list containing a list containing F
3 else if F is a disjunction of formulas then
4 |   return FLATTEN(MAP(EXTRACT, list of sub-Formulas))
5 else if F is a conjunction of formulas then
6 |   return CONCAT(MAP(EXTRACT, list of sub-Formulas))
```

Where MAP applies a function on each element of a list, FLATTEN transforms a list of lists into a list, and CONCAT transforms a conjunction of disjunction of formulas into a disjunction of conjunction of formulas with the De Morgan laws. Finally, a simplification algorithm removes all the $\top$ and $\bot$ terms.

Example 5.2 presents an execution of Algorithm 10 on the simple constraint ATMOST.

Example 5.2 (Inference rule generation) *Consider the* ATMOST*(u,X,t) global constraint. It takes an integer u, an array of $n > 0$ variables $X = [x_1, \cdots, x_n]$ and an integer t as input. The constraint ensures that at most u variables from X are assigned to t. Its filtering algorithm can filter values from X. In particular, when exactly u variables from X are instantiated to t, this value is removed from the other variables' domain. Based on the following decomposition of* ATMOST*, it is then possible to generate the explanation formula of a value removal.*

$$\text{ATMOST}(u, X, t) \iff \begin{cases} (x_i = t) \iff b^{eq}_{i,t} & \forall i \in [\![1, \#X]\!] \\ \sum_{i \in [\![1,\#X]\!]} b^{eq}_{i,v} \leq u \iff \top \end{cases}$$

Algorithm 10 proceeds as follows with the value removal event term $v = (x_i \neq t)$ as input:

1. $R^2_{=}$ *rewrites* v *as* $\neg b^{eq}_{i,t}$;
2. R^2_{Σ} *rewrites* $\neg b_{i,t}$ *as* $b_{j,t} | \forall j \in [\![1, \#X]\!] \wedge b^{sum}$. *All should be considered;*
3. b^{sum} *is rewritten* $\top$ *because the sum constraint reified variable is* $\top$ *in our decomposition. This is a terminal term;*
4. $R^3_{=}$ *rewrites* $b_{j,t}$ *as* $(x_j = t)$;
5. *Finally, all terms are terminal and* $F = (\top \wedge ((x_j = t) | \forall j \in [\![1, \#X]\!]))$.

The post-processing phase simplifies F *to* $((x_j = t) | \forall j \in [\![1, \#X]\!])$. *This explanation states that the removal of the value* t *from* x_i *is explained by all the other variables instantiated to the value* t. *The explanation* F *can then be implemented in a CP Solver for the* Atmost *constraint.*

Now we can use Algorithm 10 with the value affectation event term $v = (x_i = t)$ *as input:*

1. $R^1_{=}$ *rewrites* v *as* $b^{eq}_{i,t}$;
2. R^1_{Σ} *rewrites* $b_{i,t}$ *as* $\neg b_{j,t} | \forall j \in [\![1, \#X]\!] \wedge \neg b^{sum}$. *All should be considered;*
3. $\neg b^{sum}$ *is rewritten* $\bot$ *because the sum constraint reified variable is* $\top$ *in our decomposition. This is a terminal term;*
4. $R^2_{=}$ *rewrites* $\neg b_{j,t}$ *as* $(x_j \geq t)$;
5. *Finally, all terms are terminal and* $F = (\bot \wedge ((x_j \geq t) | \forall j \in [\![1, \#X]\!]))$.

The post-processing phase simplifies F *to* $\bot$. *We cannot find an explanation for this event, and if we look at the filtering algorithm of the* Atmost *constraint, this event cannot be triggered.*

Example 5.3 presents a generated explanation on the more complex constraint Cumulative with a graph representation of the formula.

Example 5.3 *Consider the* Cumulative(X, d, t) *constraint. This constraint expresses a capacity constraint in a scheduling problem. The variables* $X = [x_1, \cdots, x_n]$ *represent the starting time of each task. Each task has a fixed duration* d_i, *and there is a maximal task capacity* c. *We use here the* Cumulative*'s variant with fixed durations and a fixed task height of* 1. *For this constraint, we have the following decomposition [SFS13]:*

$$\text{Cumulative}(X, d, t) \iff \begin{cases} x_i \geq t \iff b^{geq}_{i,t} & \forall i \in [\![1, \#X]\!] \\ (b^{geq}_{i,(t-d_i)} \wedge \neg b^{geq}_{i,t}) \iff b^{and}_{i,t} & \forall i \in [\![1, \#X]\!] \\ \sum_{i \in [\![1, \#X]\!]} b^{and}_{i,t} < c \iff \top & \end{cases}$$

The first constraint (5.3) reifies a domain modification of the cumulative variables. The second constraint (5.3) and its reified variable $b^{\texttt{and}}$ encodes the overlap of the i^{th} task on time t. The last equation (5.3) constrains the number of overlapping tasks not to exceed the integer c.

Algorithm 10 on this decomposition with the lower bound modification event $x_i \geq t$ will result in the formula described in Figure 5.2.

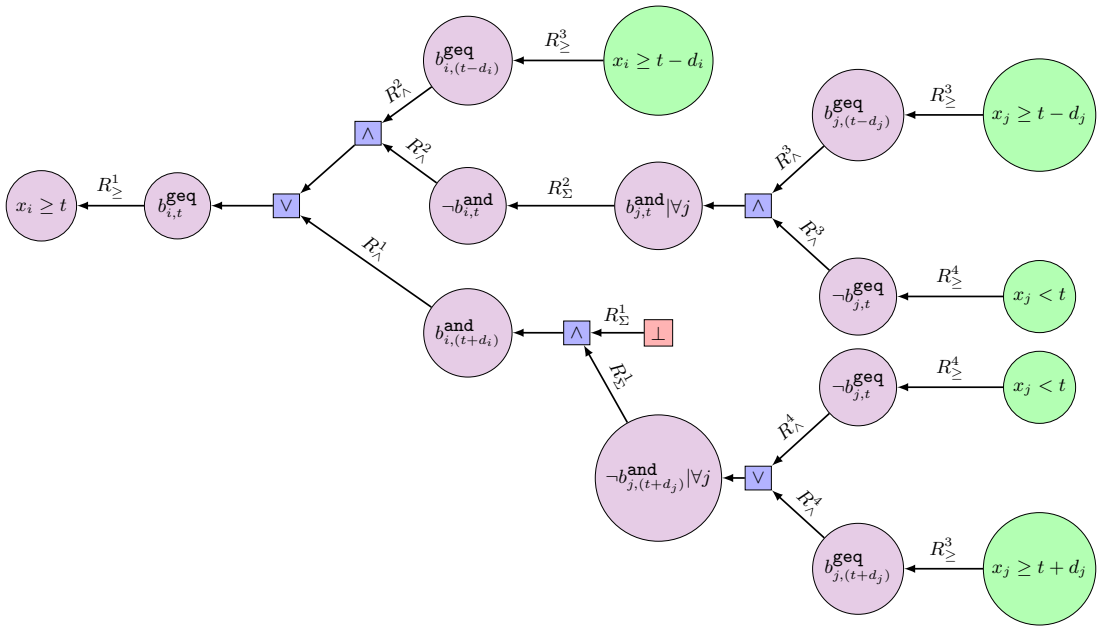


Figure 5.2 – Generated Formula graph representation of the explanation of the lower bound event of the cumulative constraint.

After simplification, the resulting explanation is:

$$(x_i \geq t - d_i) \wedge (x_j \geq t - d_j) \wedge (x_j < t) | \forall j \in [\![1, \#X]\!]$$

In this explanation, we can see that the two first terms ensure that the task x_i overlaps the time t, and the last term represents all the other tasks that saturate this time value.

5.5 Implementation and results

Algorithm 10 has been implemented in OCaml. We chose this programming language because the static type system is helpful in defining terms and formulas and the functional paradigm with pattern matching are great tools for building formulas. The code is publicly

available [7]. The complexity of Algorithm 10 is exponential in the number of constraints and variables of the decomposition. Moreover, the formula simplification of Algorithm 11 also has an exponential complexity. However, constraint decompositions are usually rather small, and the program always terminated in less than a second in our tests.

Some early tests were conducted in the Choco solver with our generated explanations of the CUMULATIVE constraint. We solved the ten rcpsp (resource-constrained project scheduling problem) instances of the MiniZinc Challenge 2008. We compared the number of nodes of the Choco solver with and without explanation, and we also ran the LCG solver Chuffed on the same problem. Chuffed uses the cumulative constraint with hand-made explanations. The results are shown in Figure 5.3. We observe a similar number of nodes for the LCG and Choco with the generated explanations. The problem without explanations is much harder. These results correspond to the conclusions of previous works on the rcpsp problem with LCG techniques [SFSW13].

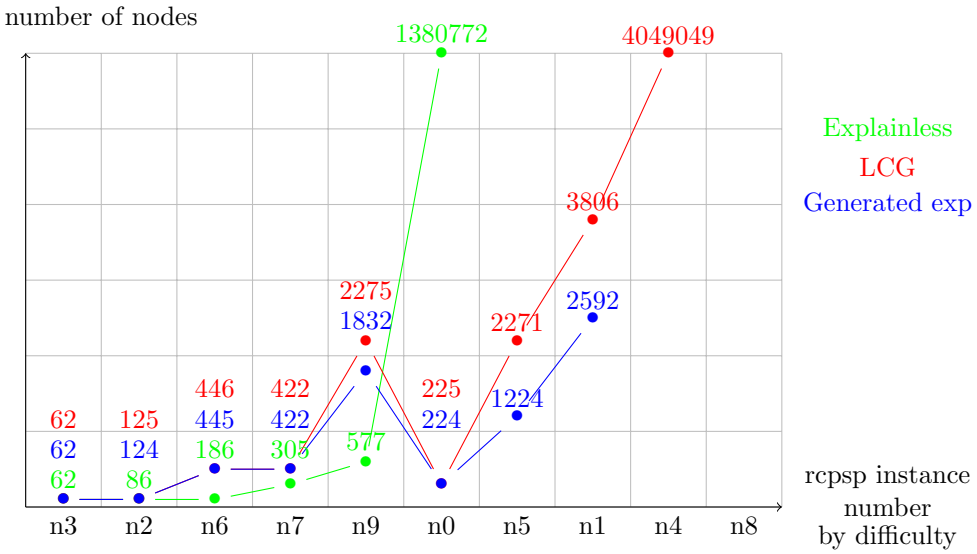


Figure 5.3 – Explanation comparison of cumulative on rcpsp problems

To test the validity of the entire approach, we ran our prototype and integrated some of the generated inference rules in the well-known LCG solver Chuffed. The recommended way to use this solver is to run it with MiniZinc [NSB+07] models. Among the set of constraints defined by MiniZinc, Chuffed only supports a few and relies on decompositions for the others, in particular for the constraints COUNT and INCREASING. We illustrate the feasibility of our approach by implementing the two constraints into Chuffed, and we

7. https://github.com/ArthurGontierPro/Explaining-Global-Constraints-from-Decompositions

instance	explanation	nodes	fails	backjumps	time (ms)
league (*model15-4-3*)	chu	10487	9717	639	393
	def	51383	51067	187	2641
	gen	4501	4038	330	189
oocsp (*racks_030_mii8*)	chu	3724887	3616895	107972	179025
	def	3951358	3875074	76263	219893
	gen	3807940	3713037	94882	186959
oc-rooster (*4s-23d*)	chu	71044	54342	16367	4432
	def	792690	774296	18055	24617
	gen	82421	62145	19939	5530

Table 5.1 – Three example instances comparing Chuffed, the default explanation and the generated explanations.

add their generated explanations. Table 5.1 shows the results on three MiniZinc instances using these constraints, with the following three configurations: *chu* (Chuffed based on the decompositions), *def* (Chuffed implementing Count and Increasing with the default explanations) and *gen* (Chuffed with the new constraints and generated explanations). We only report three interesting instances because most of the other instances do not take great advantage of Count and Increasing. Therefore, most of them do not change with the new constraints. All the other instances with raw data can be found online alongside the generator and the chuffed constraints.

Our results show that generated explanations are always more efficient than the default ones when global constraints are implemented (*def* and *gen)*. On the other hand, the interest in implementing or not explaining global constraints, as opposed to decomposed global constraints, is debatable: the performances vary slightly from one problem to another for Count and Increasing. In the end, both approaches stay close and still remain better than *def*.

5.6 Future work

In this chapter, we presented a rewriting rule system to generate a constraint explanation from its decomposition. The framework is made of atomic constraints and can be extended to any constraint as long as its rewriting rules are provided. Note that the generated explanations can be added as new rules. We implemented this rewriting algorithm and showed that it can produce explanations without effort. Finally, we tested our

generated explanation in conflict analysis solvers, firstly against previously made explanations on cumulative and then on two examples of global constraints with no explanation algorithms.

Explaining a global constraint with its decomposition raises two questions. First, there can be several decompositions for a given constraint. Therefore, the accuracy of the generated explanations may depend on the selected decomposition. Second, operationally global constraints [BH03] reach stronger filtering than their decompositions. This can lead to situations where the resulting formulas cannot be used directly because the generated explanation would not be able to explain all the events. In this case, they may be combined with the default explanation to obtain completeness. In the case of multiple explanations, we can reason like the CDCL solvers and take the smaller one to gain efficiency [SS96]. More work is needed to understand the impact of explanations on conflict analysis. Some early comparisons were conducted in [SMTdlB16]. Note that our work can also be used outside the CDCL context. For instance, in [HS17], the authors introduce a way to use explanations to better weigh the variables appearing in a constraint that fails to improve the search efficiency.

If we think further, we can try to use the explanations to build a machine-checkable proof of failure like in [VS10]. The CDCL algorithm was first designed for solver efficiency, but some work in the SAT and CP communities adapted it for proofs [HJW13, GMN22]. These solvers are slower than classical solvers for now, but the solution verifiability they provide can be appreciated in communities that need proof of security. The final upgrade for explanations would be to make them human-readable. For now, explanations are just extensive event clauses, but by combining them to find the shortest constraint that causes the problem contradiction, we could provide the user useful feedback. Similar to readable proof generation for geometry [CG96].

Chapter 6

Conclusion

In this thesis, we have studied symmetric cryptanalysis problems with solvers. We proposed new models based on graph representations of these problems. This allows us to solve them more efficiently and to obtain new results on several instances. Moreover, we proposed a model generator for differential cryptanalysis, and we also worked to improve the solving methods of CP solvers. We first recall below the main results we obtained before discussing some open line of research.

6.1 Summary of results

In the first two chapters, we have modelled cryptographic problems relying on new graph representations. We studied the search for the superpoly of Trivium and the search for the permutation with the best diffusion in the Generalized Feistel Networks (GFN). In both cases, the new representations based on graphs brought new properties and ideas to model and solve these problems with generic solvers or dedicated algorithms. For the superpoly of Trivium, we found new constraints and gave a smart strategy to the MILP solver to find the superpoly 10 to 60 times faster than previous methods. For the permutations of GFN, we were able to exploit the structures of the graph to design an efficient algorithm that was able to find new optimal permutations. We also used a graph representation of ciphers to generate CP models to solve a well-known cryptographic problem, the instantiation of truncated differential trails. We provide a tool that takes as input a graph representation of any cipher and some truncated trails to generate and solve CP models to find the best differential characteristic of the cipher. To make this tool, we added new constraints in a CP solver (Choco) and studied their efficiency. Moreover, we proposed an automatic model simplification and a parallel-solving method. Finally, we were interested in improving the CP solvers with solver explanations that allow them to learn from search conflicts. We designed a tool to generate constraint explanations to improve CP solver's efficiency. These explanations can then be added to the solving

algorithms of any conflict-learning CP solver. The explanation generator uses a rewriting rule system and a constraint decomposition to generate explanations of the constraint to improve their performances. The generated explanations were tested in CP solvers on public instances of the MiniZinc challenge with promising results.

6.2 Perspective for future work

Specific perspectives. In this thesis, we have seen that new representation can help finding interesting properties useful to enhance problems solving. However, we have not used all the forbidden sub-graphs of Trivium, and there may be other properties to be used in the model (for example, the ternary property of non-doubling edges). During this thesis, the representation of a problem using a graph has proved interesting on several occasions, and could be applied to more ciphers. The first target that comes to mind is Grain since it is quite similar to Trivium. Moreover, our graph representation of the diffusion in the Generalized Feistel Networks left two open questions. The first question is the existence of a formal proof that the even-odd property is optimal for the diffusion property. We provided some examples and new ideas that rely on the graph representation to start this proof. The second question is the study of new properties to replace or complement the diffusion. We proposed some properties and we modified our algorithm to study them, but this part deserves further study.

General perspectives. A broader question is whether it is possible to make a tool to find distinguishers on any cipher automatically. Such a tool would save development time for early cryptanalysis of new ciphers. The existing automatic tools for differential analysis are a good start. They show that cryptanalysis could be done more easily. Indeed, we can use a simple graph representation to model any cipher and use back-end solvers to solve cryptanalysis problems on them. However, much more work is needed to have a generic tool for all cryptographic problems. A good starting extension for differential automatic tools would be to solve the variants of differential cryptanalysis (boomerangs, impossible differentials, ...). Many cryptographic problems we studied require a lot of variables and constraints to be modeled. However, these models represent ciphers composed of one round function applied multiple times. Therefore, the model has a lot of similar constraints and variables in a similar context. One way to exploit this is to learn from the structure of the problem during the solving process. The idea of explaining failures was

ground-breaking for SAT solvers' efficiency. This idea has been transposed in CP solvers, and they might be very useful for cryptographic problems. However, conflict learning CP solvers need to develop an extensive explanation framework. When all constraints will be explained, further work could be done to use explanations in the similar sub-parts of a model. For instance, we think that the learnt explanations on a variable may be useful on other variables that are constrained in the same way. We could add multiple constraints from one conflict explanation. Finally, explanations may be upgraded to human readable explanations for the users to better understand the conflicts in their problems. Explanations may also be used to generate a proof certificate of the solver's reasoning. Indeed, there are several cases of cryptographic problems where we do not want to trust a black box solver, and having a proof certificate that can be verified by some other tool would help us to trust the obtained results. There is some work on proof generation in the SAT community [HJW13], and adapting this idea to make CP solvers would be an interesting research topic.

BIBLIOGRAPHY

[AAG+19] Mark Aagaard, Riham AlTawy, Guang Gong, Kalikinkar Mandal, and Raghvendra Rohit. ACE: An authenticated encryption and hash algorithm. *Submission to NIST-LWC (announced as round 2 candidate on August 30, 2019)*, 2019.

[ABCC11] David L Applegate, Robert E Bixby, Vašek Chvátal, and William J Cook. The traveling salesman problem. In *The Traveling Salesman Problem*. Princeton university press, 2011.

[Ach07] Tobias Achterberg. Conflict analysis in mixed integer programming. *Discret. Optim.*, 4(1):4–20, 2007.

[ADMS09] Jean-Philippe Aumasson, Itai Dinur, Willi Meier, and Adi Shamir. Cube Testers and Key Recovery Attacks on Reduced-Round MD6 and Trivium. In *16th International Workshop on Fast Software Encryption (FSE'09)*, volume 5665 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2009.

[AFI+04] Gwénolé Ars, Jean-Charles Faugère, Hideki Imai, Mitsuru Kawazoe, and Makoto Sugita. Comparison between XL and gröbner basis algorithms. In Pil Joong Lee, editor, *Advances in Cryptology - ASIACRYPT 2004, 10th International Conference on the Theory and Application of Cryptology and Information Security, Jeju Island, Korea, December 5-9, 2004, Proceedings*, volume 3329 of *Lecture Notes in Computer Science*, pages 338–353. Springer, 2004.

[AG99] Carlisle Adams and Jeff Gilchrist. The CAST-256 encryption algorithm. *RFC*, 2612:1–19, 1999.

[AGH+] Riham AlTawy, Guang Gong, Morgan He, Ashwin Jha, Kalikinkar Mandal, Mridul Nandi, and Raghvendra Rohit. SpoC: an authenticated cipher submission to the NIST LWC competition (2019).

[AGH+19] Riham AlTawy, Guang Gong, Morgan He, Kalikinkar Mandal, and Raghvendra Rohit. Spix: An authenticated cipher submission to the NIST LWC competition. *Submitted to NIST Lightweight Standardization Process*, 2019.

[AGP+19] Martin R. Albrecht, Lorenzo Grassi, Léo Perrin, Sebastian Ramacher, Christian Rechberger, Dragos Rotaru, Arnab Roy, and Markus Schofnegger. Feistel structures for MPC, and more. In Kazue Sako, Steve A. Schneider, and Peter Y. A. Ryan, editors, *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part II*, volume 11736 of *Lecture Notes in Computer Science*, pages 151–171. Springer, 2019.

[ÅLHJ12] Martin Ågren, Carl Löndahl, Martin Hell, and Thomas Johansson. A survey on fast correlation attacks. *Cryptogr. Commun.*, 4(3-4):173–202, 2012.

[Ano97] Advanced Encryption Standard. In Eli Biham, editor, *Fast Software Encryption, 4th International Workshop, FSE '97, Haifa, Israel, January 20-22, 1997, Proceedings*, volume 1267 of *Lecture Notes in Computer Science*, pages 83–87. Springer, 1997.

[Arm06] Frederik Armknecht. *Algebraic attacks on certain stream ciphers*. PhD thesis, University of Mannheim, Germany, 2006.

[AS09] Gilles Audemard and Laurent Simon. Glucose: a solver that predicts learnt clauses quality. *SAT Competition*, pages 7–8, 2009.

[BBdS+19] Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großschädl, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, Qingju Wang, and Alex Biryukov. Schwaemm and esch: lightweight authenticated encryption and hashing using the sparkle permutation family. *NIST round*, 2, 2019.

[BBFL22] Christof Beierle, Tim Beyne, Patrick Felke, and Gregor Leander. Constructing and deconstructing intentional weaknesses in symmetric ciphers. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part III*, volume 13509 of *Lecture Notes in Computer Science*, pages 748–778. Springer, 2022.

[BBI+20] Subhadeep Banik, Zhenzhen Bao, Takanori Isobe, Hiroyasu Kubo, Fukang Liu, Kazuhiko Minematsu, Kosei Sakamoto, Nao Shibata, and Maki Shigeri. WARP : Revisiting GFN for lightweight 128-bit block cipher. In Orr Dunkelman, Michael J. Jacobson Jr., and Colin O'Flynn, editors, *Selected Areas in Cryptography - SAC 2020 - 27th International Conference, Halifax, NS,*

Canada (Virtual Event), October 21-23, 2020, Revised Selected Papers, volume 12804 of *Lecture Notes in Computer Science*, pages 535–564. Springer, 2020.

[BBS99] Eli Biham, Alex Biryukov, and Adi Shamir. Cryptanalysis of Skipjack reduced to 31 rounds using impossible differentials. In Jacques Stern, editor, *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, volume 1592 of *Lecture Notes in Computer Science*, pages 12–23. Springer, 1999.

[BC16] Christina Boura and Anne Canteaut. Another view of the division property. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part I*, volume 9814 of *Lecture Notes in Computer Science*, pages 654–682. Springer, 2016.

[BCD+98] Carolynn Burwick, Don Coppersmith, Edward D'Avignon, Rosario Gennaro, Shai Halevi, Charanjit Jutla, Stephen M Matyas Jr, Luke O'Connor, Mohammad Peyravian, David Safford, et al. MARS-a candidate cipher for AES. *NIST AES Proposal*, 268:80, 1998.

[BCDP07] Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit. Global constraint catalogue: Past, present and future. *Constraints An Int. J.*, 12(1):21–62, 2007.

[BDK01] Eli Biham, Orr Dunkelman, and Nathan Keller. The rectangle attack - rectangling the serpent. In Birgit Pfitzmann, editor, *Advances in Cryptology - EUROCRYPT 2001, International Conference on the Theory and Application of Cryptographic Techniques, Innsbruck, Austria, May 6-10, 2001, Proceeding*, volume 2045 of *Lecture Notes in Computer Science*, pages 340–357. Springer, 2001.

[BFMT15] Thierry P Berger, Julien Francq, Marine Minier, and Gaël Thomas. Extended generalized feistel networks using matrix representation to propose a new lightweight block cipher: Lilliput. *IEEE Transactions on Computers*, 65(7):2074–2089, 2015.

[BFS10] Timo Berthold, Thibaut Feydy, and Peter J. Stuckey. Rapid learning for binary programs. In Andrea Lodi, Michela Milano, and Paolo Toth, editors,

Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 7th International Conference, CPAIOR 2010, Bologna, Italy, June 14-18, 2010. Proceedings, volume 6140 of *Lecture Notes in Computer Science*, pages 51–55. Springer, 2010.

[BGG+23a] Emanuele Bellini, David Gérault, Juan Grados, Yun Ju Huang, Mohamed Rachidi, Sharwan K. Tiwari, and Rusydi H. Makarim. CLAASP: a cryptographic library for the automated analysis of symmetric primitives. *IACR Cryptol. ePrint Arch.*, page 622, 2023.

[BGG+23b] Emanuele Bellini, David Gérault, Juan Grados, Rusydi H. Makarim, and Thomas Peyrin. Fully automated differential-linear attacks against ARX ciphers. In Mike Rosulek, editor, *Topics in Cryptology - CT-RSA 2023 - Cryptographers' Track at the RSA Conference 2023, San Francisco, CA, USA, April 24-27, 2023, Proceedings*, volume 13871 of *Lecture Notes in Computer Science*, pages 252–276. Springer, 2023.

[BH03] Christian Bessière and Pascal Van Hentenryck. To be or not to be ... a global constraint. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, volume 2833 of *Lecture Notes in Computer Science*, pages 789–794. Springer, 2003.

[Bir04] Alex Biryukov. Block ciphers and stream ciphers: The state of the art. *IACR Cryptol. ePrint Arch.*, page 94, 2004.

[BJK+16] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The SKINNY family of block ciphers and its low-latency variant MANTIS. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II*, volume 9815 of *Lecture Notes in Computer Science*, pages 123–153. Springer, 2016.

[BKN+09] Christian Bessiere, George Katsirelos, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Decompositions of All Different, Global Cardinality and related constraints. In Craig Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 419–424, 2009.

[BKNW09] Christian Bessiere, George Katsirelos, Nina Narodytska, and Toby Walsh. Circuit complexity and decompositions of global constraints. In Craig Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 412–418, 2009.

[BL16] Karthikeyan Bhargavan and Gaëtan Leurent. On the practical (in-)security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 456–467. ACM, 2016.

[BLLN21] Arghya Bhattacharjee, Cuauhtemoc Mancillas López, Eik List, and Mridul Nandi. The Oribatida v1.3 family of lightweight authenticated encryption schemes. *J. Math. Cryptol.*, 15(1):305–344, 2021.

[BN10] Alex Biryukov and Ivica Nikolic. Automatic search for related-key differential characteristics in byte-oriented block ciphers: Application to AES, camellia, khazad and others. In Henri Gilbert, editor, *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Monaco / French Riviera, May 30 - June 3, 2010. Proceedings*, volume 6110 of *Lecture Notes in Computer Science*, pages 322–344. Springer, 2010.

[BP15] Alex Biryukov and Léo Perrin. Lightweight cryptography lounge. http://cryptolux.org/index.php/Lightweight_Cryptography, 2015.

[BP17] Alex Biryukov and Léo Perrin. State of the art in lightweight symmetric cryptography. *IACR Cryptol. ePrint Arch.*, page 511, 2017.

[BPW15] Céline Blondeau, Thomas Peyrin, and Lei Wang. Known-key distinguisher on full PRESENT. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 455–474. Springer, 2015.

[BR75] James R. Bitner and Edward M. Reingold. Backtrack programming techniques. *Commun. ACM*, 18(11):651–656, 1975.

[Bru81] Maurice Bruynooghe. Solving combinatorial search problems by intelligent backtracking. *Inf. Process. Lett.*, 12(1):36–39, 1981.

[BS90] Eli Biham and Adi Shamir. Differential cryptanalysis of DES-like cryptosystems. In Alfred Menezes and Scott A. Vanstone, editors, *Advances in Cryptology - CRYPTO '90, 10th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1990, Proceedings*, volume 537 of *Lecture Notes in Computer Science*, pages 2–21. Springer, 1990.

[BS93] Eli Biham and Adi Shamir. *Differential Cryptanalysis of the Data Encryption Standard*. Springer, 1993.

[BS94] Matt Blaze and Bruce Schneier. The MacGuffin block cipher algorithm. In Bart Preneel, editor, *Fast Software Encryption: Second International Workshop. Leuven, Belgium, 14-16 December 1994, Proceedings*, volume 1008 of *Lecture Notes in Computer Science*, pages 97–110. Springer, 1994.

[BSS+13] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK families of lightweight block ciphers. *IACR Cryptol. ePrint Arch.*, page 404, 2013.

[BSW00] Alex Biryukov, Adi Shamir, and David A. Wagner. Real time cryptanalysis of A5/1 on a PC. In Bruce Schneier, editor, *Fast Software Encryption, 7th International Workshop, FSE 2000, New York, NY, USA, April 10-12, 2000, Proceedings*, volume 1978 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2000.

[Bur69] Rod M. Burstall. A program for solving word sum puzzles. *Comput. J.*, 12(1):48–51, 1969.

[BW99] Alex Biryukov and David A. Wagner. Slide attacks. In Lars R. Knudsen, editor, *Fast Software Encryption, 6th International Workshop, FSE '99, Rome, Italy, March 24-26, 1999, Proceedings*, volume 1636 of *Lecture Notes in Computer Science*, pages 245–259. Springer, 1999.

[CDJ+19] Avik Chakraborti, Nilanjan Datta, Ashwin Jha, Cuauhtemoc Mancillas Lopez, Mridul Nandi, and Yu Sasaki. LOTUS-AEAD and LOCUS-AEAD. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[CDK09] Christophe De Cannière, Orr Dunkelman, and Miroslav Knezevic. KATAN and KTANTAN - A family of small and efficient hardware-oriented block

ciphers. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*, pages 272–288. Springer, 2009.

[CDL+20] Anne Canteaut, Sébastien Duval, Gaëtan Leurent, María Naya-Plasencia, Léo Perrin, Thomas Pornin, and André Schrottenloher. Saturnin: a suite of lightweight symmetric algorithms for post-quantum security. *IACR Trans. Symmetric Cryptol.*, 2020(S1):160–207, 2020.

[CDN15] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, 2015.

[CDR98] Hélène Collavizza, François Delobel, and Michel Rueher. A note on partial consistencies over continuous domains. In Michael J. Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming - CP98, 4th International Conference, Pisa, Italy, October 26-30, 1998, Proceedings*, volume 1520 of *Lecture Notes in Computer Science*, pages 147–161. Springer, 1998.

[CG96] Shang-Ching Chou and Xiao-Shan Gao. Automated generation of readable proofs with geometric invariants i. multiple and shortest proof generation. *J. Autom. Reason.*, 17(3):325–347, 1996.

[CGT19] Victor Cauchois, Clément Gomez, and Gaël Thomas. General diffusion analysis: How to find optimal permutations for Generalized Type-II Feistel Schemes. *IACR Trans. Symmetric Cryptol.*, 2019(1):264–301, 2019.

[Cha83] David Chaum. Blind signature system. In David Chaum, editor, *Advances in Cryptology, Proceedings of CRYPTO '83, Santa Barbara, California, USA, August 21-24, 1983*, page 153. Plenum Press, New York, 1983.

[Chu11] Geoffrey G Chu. *Improving combinatorial optimization*. University of Melbourne, Department of Computer Science and Software Engineering, 2011.

[Chv73] Vasek Chvátal. Edmonds polytopes and a hierarchy of combinatorial problems. *Discret. Math.*, 4(4):305–337, 1973.

[CKPS00] Nicolas T. Courtois, Alexander Klimov, Jacques Patarin, and Adi Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In Bart Preneel, editor, *Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of*

Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding, volume 1807 of *Lecture Notes in Computer Science*, pages 392–407. Springer, 2000.

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

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971.

[CP02] Nicolas T. Courtois and Josef Pieprzyk. Cryptanalysis of block ciphers with overdefined systems of equations. In Yuliang Zheng, editor, *Advances in Cryptology - ASIACRYPT 2002, 8th International Conference on the Theory and Application of Cryptology and Information Security, Queenstown, New Zealand, December 1-5, 2002, Proceedings*, volume 2501 of *Lecture Notes in Computer Science*, pages 267–287. Springer, 2002.

[CP03] Nicolas T. Courtois and Jacques Patarin. About the XL algorithm over GF(2). In Marc Joye, editor, *Topics in Cryptology - CT-RSA 2003, The Cryptographers' Track at the RSA Conference 2003, San Francisco, CA, USA, April 13-17, 2003, Proceedings*, volume 2612 of *Lecture Notes in Computer Science*, pages 141–157. Springer, 2003.

[CP08] Christophe De Cannière and Bart Preneel. Trivium. In Matthew J. B. Robshaw and Olivier Billet, editors, *New Stream Cipher Designs - The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*, pages 244–266. Springer, 2008.

[CQ23] Junjie Cheng and Kexin Qiao. Improved graph-based model for recovering superpoly on Trivium. In Mike Rosulek, editor, *Topics in Cryptology - CT-RSA 2023 - Cryptographers' Track at the RSA Conference 2023, San Francisco, CA, USA, April 24-27, 2023, Proceedings*, volume 13871 of *Lecture Notes in Computer Science*, pages 225–251. Springer, 2023.

[Dan02] George B. Dantzig. Linear programming. *Oper. Res.*, 50(1):42–47, 2002.

[DDD05] Grégoire Dooms, Yves Deville, and Pierre Dupont. CP(Graph): Introducing a graph computation domain in constraint programming. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, volume 3709 of *Lecture Notes in Computer Science*, pages 211–225. Springer, 2005.

[DDGP21] Stéphanie Delaune, Patrick Derbez, Arthur Gontier, and Charles Prud'homme. A simpler model for recovering superpoly on trivium. In Riham AlTawy and Andreas Hülsing, editors, *Selected Areas in Cryptography - 28th International Conference, SAC 2021, Virtual Event, September 29 - October 1, 2021, Revised Selected Papers*, volume 13203 of *Lecture Notes in Computer Science*, pages 266–285. Springer, 2021.

[DDGP22] Stéphanie Delaune, Patrick Derbez, Arthur Gontier, and Charles Prud'homme. New algorithm for exhausting optimal permutations for Generalized Feistel Networks. In Takanori Isobe and Santanu Sarkar, editors, *Progress in Cryptology - INDOCRYPT 2022 - 23rd International Conference on Cryptology in India, Kolkata, India, December 11-14, 2022, Proceedings*, volume 13774 of *Lecture Notes in Computer Science*, pages 103–124. Springer, 2022.

[DDH+21] Stéphanie Delaune, Patrick Derbez, Paul Huynh, Marine Minier, Victor Mollimard, and Charles Prud'homme. Efficient Methods to Search for Best Differential Characteristics on SKINNY. In Kazue Sako and Nils Ole Tippenhauer, editors, *19th International Conference on Applied Cryptography and Network Security, (ACNS'21)*, volume 12727 of *Lecture Notes in Computer Science*, pages 184–207. Springer, 2021.

[Dec90] Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artif. Intell.*, 41(3):273–312, 1990.

[DFLM19] Patrick Derbez, Pierre-Alain Fouque, Baptiste Lambin, and Victor Mollimard. Efficient search for optimal diffusion layers of Generalized Feistel Networks. *IACR Trans. Symmetric Cryptol.*, 2019(2):218–240, 2019.

[DFS12] Nicholas Downing, Thibaut Feydy, and Peter J. Stuckey. Explaining alldifferent. In Mark Reynolds and Bruce H. Thomas, editors, *Thirty-Fifth Aus-*

tralasian Computer Science Conference, ACSC 2012, Melbourne, Australia, January 2012, volume 122 of *CRPIT*, pages 115–124. Australian Computer Society, 2012.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theory*, 22(6):644–654, 1976.

[DH77] Whitfield Diffie and Martin E. Hellman. Special feature exhaustive cryptanalysis of the NBS data encryption standard. *Computer*, 10(6):74–84, 1977.

[DKM+17] Ashutosh Dhar Dwivedi, Milos Kloucek, Pawel Morawiecki, Ivica Nikolic, Josef Pieprzyk, and Sebastian Wójtowicz. SAT-based cryptanalysis of authenticated ciphers from the CAESAR competition. In Pierangela Samarati, Mohammad S. Obaidat, and Enrique Cabello, editors, *Proceedings of the 14th International Joint Conference on e-Business and Telecommunications (ICETE 2017) - Volume 4: SECRYPT, Madrid, Spain, July 24-26, 2017*, pages 237–246. SciTePress, 2017.

[DKR97] Joan Daemen, Lars R. Knudsen, and Vincent Rijmen. The block cipher square. In Eli Biham, editor, *Fast Software Encryption, 4th International Workshop, FSE '97, Haifa, Israel, January 20-22, 1997, Proceedings*, volume 1267 of *Lecture Notes in Computer Science*, pages 149–165. Springer, 1997.

[DKS10] Orr Dunkelman, Nathan Keller, and Adi Shamir. A practical-time related-key attack on the KASUMI cryptosystem used in GSM and 3g telephony. In Tal Rabin, editor, *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*, volume 6223 of *Lecture Notes in Computer Science*, pages 393–410. Springer, 2010.

[dMB08] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[dNX19] Eduardo Marsola do Nascimento and José Antônio Moreira Xexéo. FlexAEAD-A lightweight cipher with integrated authentication. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[DR99] Joan Daemen and Vincent Rijmen. AES proposal: Rijndael. 1999.

[DS09] Itai Dinur and Adi Shamir. Cube attacks on tweakable black box polynomials. In Antoine Joux, editor, *Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings*, volume 5479 of *Lecture Notes in Computer Science*, pages 278–299. Springer, 2009.

[DSD01] Morris Dworkin, NATIONAL INST OF STANDARDS, and TECHNOLOGY GAITHERSBURG MD COMPUTER SECURITY DIV. Recommendation for block cipher modes of operation. methods and techniques. 2001.

[EJMY19] Patrik Ekdahl, Thomas Johansson, Alexander Maximov, and Jing Yang. A new SNOW stream cipher called SNOW-V. *IACR Trans. Symmetric Cryptol.*, 2019(3):1–42, 2019.

[EJT07] Håkan Englund, Thomas Johansson, and Meltem Sönmez Turan. A framework for chosen IV statistical analysis of stream ciphers. In K. Srinathan, C. Pandu Rangan, and Moti Yung, editors, *Progress in Cryptology - INDOCRYPT 2007, 8th International Conference on Cryptology in India, Chennai, India, December 9-13, 2007, Proceedings*, volume 4859 of *Lecture Notes in Computer Science*, pages 268–281. Springer, 2007.

[ENP19] Maria Eichlseder, Marcel Nageler, and Robert Primas. Analyzing the linear keystream biases in AEGIS. *IACR Trans. Symmetric Cryptol.*, 2019(4):348–368, 2019.

[ES03] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[Fag15] Jean-Guillaume Fages. On the use of graphs within constraint-programming. *Constraints An Int. J.*, 20(4):498–499, 2015.

[Fei73] Horst Feistel. Cryptography and computer privacy. *Scientific american*, 228(5):15–23, 1973.

[FJP13] Pierre-Alain Fouque, Jérémy Jean, and Thomas Peyrin. Structural evaluation of AES and chosen-key distinguisher of 9-round AES-128. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 183–203. Springer, 2013.

[FS09] Thibaut Feydy and Peter J Stuckey. Lazy clause generation reengineered. In *International Conference on Principles and Practice of Constraint Programming*, pages 352–366. Springer, 2009.

[FS14] Kathryn Glenn Francis and Peter J. Stuckey. Explaining circuit propagation. *Constraints An Int. J.*, 19(1):1–29, 2014.

[FSS13] Thibaut Feydy, Andreas Schutt, and Peter Stuckey. Semantic learning for lazy clause generation. In *TRICS workshop, held alongside CP*. Citeseer, 2013.

[FV13] Pierre-Alain Fouque and Thomas Vannet. Improving Key Recovery to 784 and 799 Rounds of Trivium Using Optimized Cube Attacks. In *20th International Workshop on Fast Software Encryption (FSE'13)*, volume 8424 of *Lecture Notes in Computer Science*, pages 502–517. Springer, 2013.

[FW74] Jay P. Fillmore and S. G. Williamson. On backtracking: A combinatorial description of the algorithm. *SIAM J. Comput.*, 3(1):41–55, 1974.

[Gas78] John Gaschnig. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing assignment problems. In *Proceedings of the Second Canadian Conference on Artificial Intelligence*, pages 268–277, 1978.

[GB65] Solomon W. Golomb and Leonard D. Baumert. Backtrack programming. *J. ACM*, 12(4):516–524, 1965.

[Ger94] Carmen Gervet. Conjunto: Constraint logic programming with finite set domains. In Maurice Bruynooghe, editor, *Logic Programming, Proceedings of the 1994 International Symposium, Ithaca, New York, USA, November 13-17, 1994*, pages 339–358. MIT Press, 1994.

[Gér18] David Gérault. *Security analysis of contactless communication protocols. (Analyse de sécurité des protocoles de communication sans contact).* PhD thesis, University of Clermont Auvergne, Clermont-Ferrand, France, 2018.

[GHG+21] Ambros Gleixner, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp M. Christophel, Kati Jarck, Thorsten Koch, Jeff Linderoth, Marco Lübbecke, Hans D. Mittelmann, Derya Ozyurt, Ted K. Ralphs, Domenico Salvagnin, and Yuji Shinano. MIPLIB 2017: Data-driven compilation of the 6th mixed-integer programming library. *Mathematical Programming Computation*, 2021.

[Gil14] Henri Gilbert. A simplified representation of AES. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part I*, volume 8873 of *Lecture Notes in Computer Science*, pages 200–222. Springer, 2014.

[Gin93] Matthew L. Ginsberg. Dynamic backtracking. *J. Artif. Intell. Res.*, 1:25–46, 1993.

[GJMN07] Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Data structures for generalised arc consistency for extensional constraints. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada*, pages 191–197. AAAI Press, 2007.

[GLMS20] David Gérault, Pascal Lafourcade, Marine Minier, and Christine Solnon. Computing AES related-key differential characteristics with constraint programming. *Artif. Intell.*, 278, 2020.

[GLST22] Jian Guo, Guozhen Liu, Ling Song, and Yi Tu. Exploring SAT for cryptanalysis: (quantum) collision attacks against 6-round SHA-3. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology - ASIACRYPT 2022 - 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5-9, 2022, Proceedings, Part III*, volume 13793 of *Lecture Notes in Computer Science*, pages 645–674. Springer, 2022.

[GMN22] Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. An auditable constraint programming solver. In Christine Solnon, editor, *28th International Conference on Principles and Practice of Constraint Programming, CP 2022, July 31 to August 8, 2022, Haifa, Israel*, volume 235 of *LIPIcs*, pages 25:1–25:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[GMS16] David Gérault, Marine Minier, and Christine Solnon. Constraint programming models for chosen key differential cryptanalysis. In Michel Rueher, editor, *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, volume 9892 of *Lecture Notes in Computer Science*, pages 584–601. Springer, 2016.

[GMS17] David Gérault, Marine Minier, and Christine Solnon. Using Constraint Programming to solve a Cryptanalytic Problem. In Carles Sierra, editor, *26th International Joint Conference on Artificial Intelligence (IJCAI'17)*, pages 4844–4848. ijcai.org, 2017.

[Gom02] Ralph E. Gomory. Early integer programming. *Oper. Res.*, 50(1):78–81, 2002.

[Gom10] Ralph E. Gomory. Outline of an algorithm for integer solutions to linear programs *and* an algorithm for the mixed integer problem. In Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey, editors, *50 Years of Integer Programming 1958-2008 - From the Early Years to the State-of-the-Art*, pages 77–103. Springer, 2010.

[GP10] Henri Gilbert and Thomas Peyrin. Super-sbox cryptanalysis: Improved attacks for AES-like permutations. In Seokhie Hong and Tetsu Iwata, editors, *Fast Software Encryption, 17th International Workshop, FSE 2010, Seoul, Korea, February 7-10, 2010, Revised Selected Papers*, volume 6147 of *Lecture Notes in Computer Science*, pages 365–383. Springer, 2010.

[GR20] Lorenzo Grassi and Christian Rechberger. Revisiting Gilbert's known-key distinguisher. *Des. Codes Cryptogr.*, 88(7):1401–1445, 2020.

[GS20] Graeme Gange and Peter J. Stuckey. The argmax constraint. In Helmut Simonis, editor, *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September*

7-11, 2020, Proceedings, volume 12333 of *Lecture Notes in Computer Science*, pages 323–337. Springer, 2020.

[Gur21] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2021.

[HC99] Ivan Hamer and Paul Chow. DES cracking on the transmogrifier 2a. In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, volume 1717 of *Lecture Notes in Computer Science*, pages 13–24. Springer, 1999.

[HE79] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. In Bruce G. Buchanan, editor, *Proceedings of the Sixth International Joint Conference on Artificial Intelligence, IJCAI 79, Tokyo, Japan, August 20-23, 1979, 2 Volumes*, pages 356–364. William Kaufmann, 1979.

[Hey02] Howard M. Heys. A tutorial on linear and differential cryptanalysis. *Cryptologia*, 26(3):189–221, 2002.

[HHPW22] Jiahui He, Kai Hu, Bart Preneel, and Meiqin Wang. Stretching cube attacks: Improved methods to recover massive superpolies. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology - ASIACRYPT 2022 - 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5-9, 2022, Proceedings, Part IV*, volume 13794 of *Lecture Notes in Computer Science*, pages 537–566. Springer, 2022.

[HJM07] Martin Hell, Thomas Johansson, and Willi Meier. Grain: a stream cipher for constrained environments. *Int. J. Wirel. Mob. Comput.*, 2(1):86–93, 2007.

[HJW13] Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 181–188. IEEE, 2013.

[HLLT20] Phil Hebborn, Baptiste Lambin, Gregor Leander, and Yosuke Todo. Lower Bounds on the Degree of Block Ciphers. In *26th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT'20)*, volume 12491 of *Lecture Notes in Computer Science*, pages 537–566. Springer, 2020.

[HLM+20] Yonglin Hao, Gregor Leander, Willi Meier, Yosuke Todo, and Qingju Wang. Modeling for Three-Subset Division Property Without Unknown Subset - Improved Cube Attacks Against Trivium and Grain-128AEAD. In Anne Canteaut and Yuval Ishai, editors, *39th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT'20)*, volume 12105 of *Lecture Notes in Computer Science*, pages 466–495. Springer, 2020.

[HLM+21] Yonglin Hao, Gregor Leander, Willi Meier, Yosuke Todo, and Qingju Wang. Modeling for three-subset division property without unknown subset. *J. Cryptol.*, 34(3):22, 2021.

[HM14] Pascal Van Hentenryck and Laurent D. Michel. Domain views for constraint programming. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, volume 8656 of *Lecture Notes in Computer Science*, pages 705–720. Springer, 2014.

[HS17] Emmanuel Hebrard and Mohamed Siala. Explanation-based weighted degree. In Domenico Salvagnin and Michele Lombardi, editors, *Integration of AI and OR Techniques in Constraint Programming - 14th International Conference, CPAIOR 2017, Padua, Italy, June 5-8, 2017, Proceedings*, volume 10335 of *Lecture Notes in Computer Science*, pages 167–175. Springer, 2017.

[HSH+06] Deukjo Hong, Jaechul Sung, Seokhie Hong, Jongin Lim, Sangjin Lee, Bonseok Koo, Changhoon Lee, Donghoon Chang, Jesang Lee, Kitae Jeong, Hyun Kim, Jongsung Kim, and Seongtaek Chee. HIGHT: A new block cipher suitable for low-resource device. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*, volume 4249 of *Lecture Notes in Computer Science*, pages 46–59. Springer, 2006.

[HST+21] Kai Hu, Siwei Sun, Yosuke Todo, Meiqin Wang, and Qingju Wang. Massive superpoly recovery with nested monomial predictions. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6-10, 2021, Proceedings, Part I*, volume 13090 of *Lecture Notes in Computer Science*, pages 392–421. Springer, 2021.

[HSWW20] Kai Hu, Siwei Sun, Meiqin Wang, and Qingju Wang. An algebraic formulation of the division property: Revisiting degree evaluations, cube attacks, and key-independent sums. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part I*, volume 12491 of *Lecture Notes in Computer Science*, pages 446–476. Springer, 2020.

[IKM+19] Tetsu Iwata, Mustafa Khairallah, Kazuhiko Minematsu, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Ling Sun. Thank goodness it's friday (TGIF). *Submission to Round*, 1:157, 2019.

[JDB00] Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. Maintaining arc-consistency within dynamic backtracking. In Rina Dechter, editor, *Principles and Practice of Constraint Programming - CP 2000, 6th International Conference, Singapore, September 18-21, 2000, Proceedings*, volume 1894 of *Lecture Notes in Computer Science*, pages 249–261. Springer, 2000.

[JK97] Thomas Jakobsen and Lars R. Knudsen. The interpolation attack on block ciphers. In Eli Biham, editor, *Fast Software Encryption, 4th International Workshop, FSE '97, Haifa, Israel, January 20-22, 1997, Proceedings*, volume 1267 of *Lecture Notes in Computer Science*, pages 28–40. Springer, 1997.

[JP22] Dimitri Justeau-Allaire and Charles Prud'homme. Global domain views for expressive and cross-domain constraint programming. *Constraints An Int. J.*, 27(1-2):1–7, 2022.

[Jus03] Narendra Jussien. *The versatility of using explanations within constraint programming*. 2003.

[KB05] George Katsirelos and Fahiem Bacchus. Generalized nogoods in CSPs. In Manuela M. Veloso and Subbarao Kambhampati, editors, *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 390–396. AAAI Press / The MIT Press, 2005.

[Ker83] Auguste Kerckhoffs. La cryptographie militaire. *J. Sci. Militaires*, 9(4):5–38, 1883.

[KM04] Lars R. Knudsen and John Erik Mathiassen. On the role of key schedules in attacks on iterated ciphers. In Pierangela Samarati, Peter Y. A. Ryan, Dieter Gollmann, and Refik Molva, editors, *Computer Security - ESORICS 2004, 9th European Symposium on Research Computer Security, Sophia Antipolis, France, September 13-15, 2004, Proceedings*, volume 3193 of *Lecture Notes in Computer Science*, pages 322–334. Springer, 2004.

[Knu94] Lars R. Knudsen. Truncated and higher order differentials. In Bart Preneel, editor, *Fast Software Encryption: Second International Workshop. Leuven, Belgium, 14-16 December 1994, Proceedings*, volume 1008 of *Lecture Notes in Computer Science*, pages 196–211. Springer, 1994.

[Köl] Stefan Kölbl. CryptoSMT: An easy to use tool for cryptanalysis of symmetric primitives (2015). *URL: https://github. com/kste/cryptosmt*.

[Laf18] Frédéric Lafitte. CryptoSAT: a tool for SAT-based cryptanalysis. *IET Inf. Secur.*, 12(6):463–474, 2018.

[Lai94] Xuejia Lai. Higher order derivatives and differential cryptanalysis. *Communications and Cryptography: Two Sides of One Tapestry*, pages 227–233, 1994.

[Lau78] Jean-Louis Laurière. A language and a program for stating and solving combinatorial problems. *Artif. Intell.*, 10(1):29–127, 1978.

[LDLS21] Luc Libralesso, François Delobel, Pascal Lafourcade, and Christine Solnon. Automatic generation of declarative models for differential cryptanalysis. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, volume 210 of *LIPIcs*, pages 40:1–40:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[Lec11] Christophe Lecoutre. STR2: optimized simple tabular reduction for table constraints. *Constraints An Int. J.*, 16(4):341–371, 2011.

[Leh57] Derrick H Lehmer. Combinatorial problems with digital computers. In *Proc. of the Fourth Canadian Math. Congress*, pages 160–173, 1957.

[Leu12] Gaëtan Leurent. Analysis of differential attacks in ARX constructions. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December*

2-6, 2012. Proceedings, volume 7658 of *Lecture Notes in Computer Science*, pages 226–243. Springer, 2012.

[Liu17] Meicheng Liu. Degree Evaluation of NFSR-Based Cryptosystems. In *37th Annual International Cryptology Conference (CRYPTO'17)*, volume 10403 of *Lecture Notes in Computer Science*, pages 227–249. Springer, 2017.

[LLY12] Christophe Lecoutre, Chavalit Likitvivatanavong, and Roland H. C. Yap. A path-optimal GAC algorithm for table constraints. In Luc De Raedt, Christian Bessiere, Didier Dubois, Patrick Doherty, Paolo Frasconi, Fredrik Heintz, and Peter J. F. Lucas, editors, *ECAI 2012 - 20th European Conference on Artificial Intelligence. Including Prestigious Applications of Artificial Intelligence (PAIS-2012) System Demonstrations Track, Montpellier, France, August 27-31 , 2012*, volume 242 of *Frontiers in Artificial Intelligence and Applications*, pages 510–515. IOS Press, 2012.

[LMM91] Xuejia Lai, James L. Massey, and Sean Murphy. Markov ciphers and differential cryptanalysis. In Donald W. Davies, editor, *Advances in Cryptology - EUROCRYPT '91, Workshop on the Theory and Application of of Cryptographic Techniques, Brighton, UK, April 8-11, 1991, Proceedings*, volume 547 of *Lecture Notes in Computer Science*, pages 17–38. Springer, 1991.

[LR88] Michael Luby and Charles Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM J. Comput.*, 17(2):373–386, 1988.

[LS06] Christophe Lecoutre and Radoslaw Szymanek. Generalized arc consistency for positive table constraints. In Frédéric Benhamou, editor, *Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings*, volume 4204 of *Lecture Notes in Computer Science*, pages 284–298. Springer, 2006.

[LSHS13] Manuel Loth, Michèle Sebag, Youssef Hamadi, and Marc Schoenauer. Bandit-based search for constraint programming. In Christian Schulte, editor, *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, volume 8124 of *Lecture Notes in Computer Science*, pages 464–480. Springer, 2013.

[Mac77] Alan K. Mackworth. Consistency in networks of relations. *Artif. Intell.*, 8(1):99–118, 1977.

[Mas69] James L. Massey. Shift-register synthesis and BCH decoding. *IEEE Trans. Inf. Theory*, 15(1):122–127, 1969.

[Mas99] Fabio Massacci. Using Walk-SAT and Rel-Sat for cryptographic key search. In Thomas Dean, editor, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 pages*, pages 290–295. Morgan Kaufmann, 1999.

[Mat93] Mitsuru Matsui. Linear cryptanalysis method for DES cipher. In Tor Helleseth, editor, *Advances in Cryptology - EUROCRYPT '93, Workshop on the Theory and Application of of Cryptographic Techniques, Lofthus, Norway, May 23-27, 1993, Proceedings*, volume 765 of *Lecture Notes in Computer Science*, pages 386–397. Springer, 1993.

[Mat94] Mitsuru Matsui. The first experimental cryptanalysis of the data encryption standard. In Yvo Desmedt, editor, *Advances in Cryptology - CRYPTO '94, 14th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21-25, 1994, Proceedings*, volume 839 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 1994.

[MHD14] Jean-Baptiste Mairy, Pascal Van Hentenryck, and Yves Deville. Optimal and efficient filtering algorithms for table constraints. *Constraints An Int. J.*, 19(1):77–120, 2014.

[Mil82] Frank Miller. *Telegraphic code to insure privacy and secrecy in the transmission of telegrams*. CM Cornwell, 1882.

[Mir02] Ilya Mironov. (not so) random shuffles of RC4. *IACR Cryptol. ePrint Arch.*, page 67, 2002.

[MLM09] João Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 131–153. IOS Press, 2009.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of*

the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001, pages 530–535. ACM, 2001.

[Mon74] Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Inf. Sci.*, 7:95–132, 1974.

[MPP09] Marine Minier, Raphael C.-W. Phan, and Benjamin Pousse. Distinguishers for ciphers and known key attack against Rijndael with large blocks. In Bart Preneel, editor, *Progress in Cryptology - AFRICACRYPT 2009, Second International Conference on Cryptology in Africa, Gammarth, Tunisia, June 21-25, 2009. Proceedings*, volume 5580 of *Lecture Notes in Computer Science*, pages 60–76. Springer, 2009.

[MPRS09] Florian Mendel, Thomas Peyrin, Christian Rechberger, and Martin Schläffer. Improved cryptanalysis of the reduced grøstl compression function, ECHO permutation and AES block cipher. In Michael J. Jacobson Jr., Vincent Rijmen, and Reihaneh Safavi-Naini, editors, *Selected Areas in Cryptography, 16th Annual International Workshop, SAC 2009, Calgary, Alberta, Canada, August 13-14, 2009, Revised Selected Papers*, volume 5867 of *Lecture Notes in Computer Science*, pages 16–35. Springer, 2009.

[MRST09] Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. The rebound attack: Cryptanalysis of reduced whirlpool and grøstl. In Orr Dunkelman, editor, *Fast Software Encryption, 16th International Workshop, FSE 2009, Leuven, Belgium, February 22-25, 2009, Revised Selected Papers*, volume 5665 of *Lecture Notes in Computer Science*, pages 260–276. Springer, 2009.

[MS89] Willi Meier and Othmar Staffelbach. Fast correlation attacks on certain stream ciphers. *J. Cryptol.*, 1(3):159–176, 1989.

[MS01] Itsik Mantin and Adi Shamir. A practical attack on broadcast RC4. In Mitsuru Matsui, editor, *Fast Software Encryption, 8th International Workshop, FSE 2001 Yokohama, Japan, April 2-4, 2001, Revised Papers*, volume 2355 of *Lecture Notes in Computer Science*, pages 152–164. Springer, 2001.

[MSR14] Marine Minier, Christine Solnon, and Julia Reboul. Solving a symmetric key cryptographic problem with constraint programming. In *ModRef 2014, Workshop of the CP 2014 Conference*, page 13, 2014.

[MWGP11] Nicky Mouha, Qingju Wang, Dawu Gu, and Bart Preneel. Differential and linear cryptanalysis using mixed-integer linear programming. In Chuankun Wu, Moti Yung, and Dongdai Lin, editors, *Information Security and Cryptology - 7th International Conference, Inscrypt 2011, Beijing, China, November 30 - December 3, 2011. Revised Selected Papers*, volume 7537 of *Lecture Notes in Computer Science*, pages 57–76. Springer, 2011.

[MZ06] Ilya Mironov and Lintao Zhang. Applications of SAT solvers to cryptanalysis of hash functions. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121 of *Lecture Notes in Computer Science*, pages 102–115. Springer, 2006.

[Nar11] Nina Narodytska. *Reformulation of global constraints*. PhD thesis, University of New South Wales, Sydney, Australia, 2011.

[NPSS10] Ivica Nikolic, Josef Pieprzyk, Przemyslaw Sokolowski, and Ron Steinfeld. Known and chosen key differential distinguishers for block ciphers. In Kyung Hyune Rhee and DaeHun Nyang, editors, *Information Security and Cryptology - ICISC 2010 - 13th International Conference, Seoul, Korea, December 1-3, 2010, Revised Selected Papers*, volume 6829 of *Lecture Notes in Computer Science*, pages 29–48. Springer, 2010.

[NSB+07] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007.

[Nyb96] Kaisa Nyberg. Generalized Feistel Networks. In Kwangjo Kim and Tsutomu Matsumoto, editors, *Advances in Cryptology - ASIACRYPT '96, International Conference on the Theory and Applications of Cryptology and Information Security, Kyongju, Korea, November 3-7, 1996, Proceedings*, volume 1163 of *Lecture Notes in Computer Science*, pages 91–104. Springer, 1996.

[OSC07] Olga Ohrimenko, Peter J Stuckey, and Michael Codish. Propagation= lazy clause generation. In *Principles and Practice of Constraint Programming–*

CP 2007: 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007. Proceedings 13, pages 544–558. Springer, 2007.

[OSC09] Olga Ohrimenko, Peter J Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14:357–391, 2009.

[PF22] Charles Prud'homme and Jean-Guillaume Fages. Choco-solver: A java library for constraint programming. *Journal of Open Source Software*, 7(78):4708, 2022.

[Pie90] Josef Pieprzyk. How to construct pseudorandom permutations from single pseudorandom functions. In Ivan Damgård, editor, *Advances in Cryptology - EUROCRYPT '90, Workshop on the Theory and Application of of Cryptographic Techniques, Aarhus, Denmark, May 21-24, 1990, Proceedings*, volume 473 of *Lecture Notes in Computer Science*, pages 140–150. Springer, 1990.

[Pro93] Patrick Prosser. Domain filtering can degrade intelligent backtracking search. In Ruzena Bajcsy, editor, *Proceedings of the 13th International Joint Conference on Artificial Intelligence. Chambéry, France, August 28 - September 3, 1993*, pages 262–267. Morgan Kaufmann, 1993.

[PRS02] Sarvar Patel, Zulfikar Ramzan, and Ganapathy S. Sundaram. Luby-Rackoff ciphers: Why XOR is not so exclusive. 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 271–290. Springer, 2002.

[Raj90] Arvind Rajan. Theory of linear and integer programming, by alexander schrijver, wiley, new york, 1986, 471 pp. *Networks*, 20(6):801, 1990.

[RBH+09] Peter Y. A. Ryan, David Bismark, James Heather, Steve A. Schneider, and Zhe Xia. Prêt à voter: a voter-verifiable voting system. *IEEE Trans. Inf. Forensics Secur.*, 4(4):662–673, 2009.

[Rej81] Marian Rejewski. How Polish mathematicians deciphered the enigma. *Annals of the History of Computing*, 3(3):213–234, 1981.

[RGMS22] Loïc Rouquette, David Gérault, Marine Minier, and Christine Solnon. And Rijndael?: Automatic related-key differential analysis of Rijndael. In Lejla Batina and Joan Daemen, editors, *Progress in Cryptology -*

AFRICACRYPT 2022: 13th International Conference on Cryptology in Africa, AFRICACRYPT 2022, Fes, Morocco, July 18-20, 2022, Proceedings, Lecture Notes in Computer Science, pages 150–175. Springer Nature Switzerland, 2022.

[RJL03] Guillaume Rochart, Narendra Jussien, and François Laburthe. Challenging explanations for global constraints. In *CP03 Workshop on User-Interaction in Constraint Satisfaction (UICS'03)*, 2003.

[RM18] Jean-Charles Régin and Arnaud Malapert. Parallel constraint programming. In Youssef Hamadi and Lakhdar Sais, editors, *Handbook of Parallel Constraint Reasoning*, pages 337–379. Springer, 2018.

[Ros88] Francesca Rossi. Constraint satisfaction problems in logic programming. *SIGART Newsl.*, 106:24–28, 1988.

[RR22] Adrián Ranea and Vincent Rijmen. Characteristic automated search of cryptographic algorithms for distinguishing attacks (CASCADA). *IET Inf. Secur.*, 16(6):470–481, 2022.

[RS20] Loïc Rouquette and Christine Solnon. abstractxor: A global constraint dedicated to differential cryptanalysis. In Helmut Simonis, editor, *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings*, volume 12333 of *Lecture Notes in Computer Science*, pages 566–584. Springer, 2020.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[Rue84] Rainer A. Rueppel. *New approaches to stream ciphers.* PhD thesis, ETH Zurich, Zürich, Switzerland, 1984.

[RvBW06] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006.

[S+99] Data Encryption Standard et al. Data encryption standard. *Federal Information Processing Standards Publication*, 112, 1999.

[SFS13] Andreas Schutt, Thibaut Feydy, and Peter J. Stuckey. Explaining time-table-edge-finding propagation for the cumulative resource constraint. In Carla P.

Gomes and Meinolf Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings*, volume 7874 of *Lecture Notes in Computer Science*, pages 234–250. Springer, 2013.

[SFSW11] Andreas Schutt, Thibaut Feydy, Peter J. Stuckey, and Mark G. Wallace. Explaining the cumulative propagator. *Constraints An Int. J.*, 16(3):250–282, 2011.

[SFSW13] Andreas Schutt, Thibaut Feydy, Peter J. Stuckey, and Mark G. Wallace. Solving RCPSP/max by lazy clause generation. *J. Sched.*, 16(3):273–289, 2013.

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

[Sho94] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*, pages 124–134. IEEE Computer Society, 1994.

[SHW+14] Siwei Sun, Lei Hu, Peng Wang, Kexin Qiao, Xiaoshuang Ma, and Ling Song. Automatic security evaluation and (related-key) differential characteristic search: Application to SIMON, PRESENT, LBlock, DES(L) and other Bit-Oriented block ciphers. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part I*, volume 8873 of *Lecture Notes in Computer Science*, pages 158–178. Springer, 2014.

[SIH+11] Kyoji Shibutani, Takanori Isobe, Harunaga Hiwatari, Atsushi Mitsuda, Toru Akishita, and Taizo Shirai. Piccolo: An ultra-lightweight blockcipher. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, volume 6917 of *Lecture Notes in Computer Science*, pages 342–357. Springer, 2011.

[SM10] Tomoyasu Suzaki and Kazuhiko Minematsu. Improving the Generalized Feistel. In Seokhie Hong and Tetsu Iwata, editors, *Fast Software Encryption,*

17th International Workshop, FSE 2010, Seoul, Korea, February 7-10, 2010, Revised Selected Papers, volume 6147 of *Lecture Notes in Computer Science*, pages 19–39. Springer, 2010.

[Smi71] JL Smith. The design of lucifer, a cryptographic device for data communication. *Technical report, IBM T.J. Watson Research Center, Yorktown Heights, N.Y.*, 1971.

[SMMK12] Tomoyasu Suzaki, Kazuhiko Minematsu, Sumio Morioka, and Eita Kobayashi. TWINE : A lightweight block cipher for multiple platforms. In Lars R. Knudsen and Huapeng Wu, editors, *Selected Areas in Cryptography, 19th International Conference, SAC 2012, Windsor, ON, Canada, August 15-16, 2012, Revised Selected Papers*, volume 7707 of *Lecture Notes in Computer Science*, pages 339–354. Springer, 2012.

[SMS+20] Kosei Sakamoto, Kazuhiko Minematsu, Nao Shibata, Maki Shigeri, Hiroyasu Kubo, Yuki Funabiki, and Takanori Isobe. Security of related-key differential attacks on TWINE, revisited. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 103-A(1):212–214, 2020.

[SMTdlB16] Maxim Shishmarev, Christopher Mears, Guido Tack, and Maria Garcia de la Banda. Learning from learning solvers. In Michel Rueher, editor, *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, volume 9892 of *Lecture Notes in Computer Science*, pages 455–472. Springer, 2016.

[Soo16] Mate Soos. The CryptoMiniSat 5 set of solvers at SAT competition 2016. *Proceedings of SAT Competition*, page 28, 2016.

[SP03] P. Souradyuti and B. Preneel. Non-fortitous RC4 key stream generator. pages 318—-325, 2003.

[SPQ03] Francois-Xavier Standaert, Gilles Piret, and Jean-Jacques Quisquater. Cryptanalysis of block ciphers: A survey. *UCL Crypto Group*, 2003.

[SS77] Richard M. Stallman and Gerald J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artif. Intell.*, 9(2):135–196, 1977.

[SS96] João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In Rob A. Rutenbar and Ralph H. J. M. Otten, editors, *Proceedings of the 1996 IEEE/ACM International Conference on*

Computer-Aided Design, ICCAD 1996, San Jose, CA, USA, November 10-14, 1996, pages 220–227. IEEE Computer Society / ACM, 1996.

[SS99] João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.

[SS08] Christian Schulte and Peter J. Stuckey. Efficient constraint propagation engines. *ACM Trans. Program. Lang. Syst.*, 31(1):2:1–2:43, 2008.

[SS16] Andreas Schutt and Peter J. Stuckey. Explaining producer/consumer constraints. In Michel Rueher, editor, *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, volume 9892 of *Lecture Notes in Computer Science*, pages 438–454. Springer, 2016.

[SSA+07] Taizo Shirai, Kyoji Shibutani, Toru Akishita, Shiho Moriai, and Tetsu Iwata. The 128-bit blockcipher CLEFIA, fast software encryption-FSE 2007, LNCS 4593, pp: 181-195, 2007.

[ST13] Christian Schulte and Guido Tack. View-based propagator derivation. *Constraints An Int. J.*, 18(1):75–107, 2013.

[ST17] Yu Sasaki and Yosuke Todo. New impossible differential search tool from design and cryptanalysis aspects - revealing structural properties of several ciphers. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part III*, volume 10212 of *Lecture Notes in Computer Science*, pages 185–215, 2017.

[Sta14] Martin Stanek. Experimenting with shuffle block cipher and SMT solvers. *IACR Cryptol. ePrint Arch.*, page 919, 2014.

[Sto16] Ko Stoffelen. Optimizing S-Box implementations for several criteria using SAT solvers. In Thomas Peyrin, editor, *Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers*, volume 9783 of *Lecture Notes in Computer Science*, pages 140–160. Springer, 2016.

[Stu10] Peter J. Stuckey. Lazy clause generation: Combining the power of SAT and CP (and MIP?) solving. In Andrea Lodi, Michela Milano, and Paolo

Toth, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 7th International Conference, CPAIOR 2010, Bologna, Italy, June 14-18, 2010. Proceedings*, volume 6140 of *Lecture Notes in Computer Science*, pages 5–9. Springer, 2010.

[SV94] Thomas Schiex and Gérard Verfaillie. Nogood recording for static and dynamic constraint satisfaction problems. *Int. J. Artif. Intell. Tools*, 3(2):187–208, 1994.

[SY11] Yu Sasaki and Kan Yasuda. Known-key distinguishers on 11-round Feistel and collision attacks on its hashing modes. In Antoine Joux, editor, *Fast Software Encryption - 18th International Workshop, FSE 2011, Lyngby, Denmark, February 13-16, 2011, Revised Selected Papers*, volume 6733 of *Lecture Notes in Computer Science*, pages 397–415. Springer, 2011.

[TB22] Je Sen Teh and Alex Biryukov. Differential cryptanalysis of WARP. *J. Inf. Secur. Appl.*, 70:103316, 2022.

[TM16] Yosuke Todo and Masakatu Morii. Bit-based division property and application to simon family. In Thomas Peyrin, editor, *Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers*, volume 9783 of *Lecture Notes in Computer Science*, pages 357–377. Springer, 2016.

[Tod15] Yosuke Todo. Structural evaluation by generalized integral property. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 287–314. Springer, 2015.

[Tod17] Yosuke Todo. Integral cryptanalysis on full MISTY1. *J. Cryptol.*, 30(3):920–959, 2017.

[Udo21] Aleksei Udovenko. MILP modeling of boolean functions by minimum number of inequalities. *IACR Cryptol. ePrint Arch.*, page 1099, 2021.

[Ull07] Julian R. Ullmann. Partition search for non-binary constraint satisfaction. *Inf. Sci.*, 177(18):3639–3678, 2007.

[vH01] Willem Jan van Hoeve. The alldifferent constraint: A survey. *CoRR*, cs.PL/0105015, 2001.

[Vie07] Michael Vielhaber. Breaking ONE. FIVIUM by AIDA an algebraic IV differential attack. *Cryptology ePrint Archive*, 2007.

[VS10] Michael Veksler and Ofer Strichman. A proof-producing CSP solver. In Maria Fox and David Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press, 2010.

[Wag99] David A. Wagner. The boomerang attack. In Lars R. Knudsen, editor, *Fast Software Encryption, 6th International Workshop, FSE '99, Rome, Italy, March 24-26, 1999, Proceedings*, volume 1636 of *Lecture Notes in Computer Science*, pages 156–170. Springer, 1999.

[WBH17] Jakob Witzig, Timo Berthold, and Stefan Heinz. Experiments with conflict analysis in mixed integer programming. In Domenico Salvagnin and Michele Lombardi, editors, *Integration of AI and OR Techniques in Constraint Programming - 14th International Conference, CPAIOR 2017, Padua, Italy, June 5-8, 2017, Proceedings*, volume 10335 of *Lecture Notes in Computer Science*, pages 211–220. Springer, 2017.

[WZ11] Wenling Wu and Lei Zhang. LBlock: A lightweight block cipher. In Javier López and Gene Tsudik, editors, *Applied Cryptography and Network Security - 9th International Conference, ACNS 2011, Nerja, Spain, June 7-10, 2011. Proceedings*, volume 6715 of *Lecture Notes in Computer Science*, pages 327–344, 2011.

[YCC04] Bo-Yin Yang, Jiun-Ming Chen, and Nicolas T. Courtois. On asymptotic security estimates in XL and gröbner bases-related algebraic cryptanalysis. In Javier López, Sihan Qing, and Eiji Okamoto, editors, *Information and Communications Security, 6th International Conference, ICICS 2004, Malaga, Spain, October 27-29, 2004, Proceedings*, volume 3269 of *Lecture Notes in Computer Science*, pages 401–413. Springer, 2004.

[ZJR+19] Muhammad Reza Z'aba, Norziana Jamil, Mohd Saufy Rohmad, Hazlin Abdul Rani, and Solahuddin Shamsuddin. The cilipadi family of lightweight authenticated encryption. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[ZK16] Neng-Fa Zhou and Håkan Kjellerstrand. The Picat-SAT compiler. In Marco Gavanelli and John H. Reppy, editors, *Practical Aspects of Declarative Lan-*

guages - 18th International Symposium, PADL 2016, St. Petersburg, FL, USA, January 18-19, 2016. Proceedings, volume 9585 of *Lecture Notes in Computer Science*, pages 48–62. Springer, 2016.

[ZMI89] Yuliang Zheng, Tsutomu Matsumoto, and Hideki Imai. On the construction of block ciphers provably secure and not relying on any unproved hypotheses. In Gilles Brassard, editor, *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 461–480. Springer, 1989.

List of Figures

List of Tables

List of Algorithms

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

Université de Rennes

Titre : Utilisation de solveurs génériques pour la cryptanalyse de chiffrements symétriques

Mot clés : Cryptanalyse, Chiffrement Symétrique, Programmation par Contraintes

Résumé : La cryptographie est une science cruciale pour nos sociétés connectées. Elle implique la conception l'analyse et la mise en œuvre d'algorithmes de chiffrement. L'analyse des chiffrements est une étape obligatoire pour assurer leur sécurité, mais cette tâche est souvent fastidieuse. En cryptographie symétrique, cette analyse porte principalement sur la recherche de *distingueurs*, des propriétés qui distinguent un message chiffré d'un message aléatoire. Les solveurs génériques sont des outils créés à l'origine pour résoudre des problèmes comme la planification ou l'ordonnancement. Ils sont régulièrement améliorés et sont devenus des bons candidats pour faciliter la cryptanalyse.

Cette thèse s'intéresse à l'analyse et la conception des chiffrements symétriques avec l'aide des solveurs génériques, notamment de programmation par contraintes (CP). Nous avons exploré plusieurs pistes pour améliorer les techniques de cryptanalyse et les solveurs. Nous avons modélisé sous forme de graphe la recherche de distingueurs (plus précisément la recherche de superpoly dans le chiffrement TRIVIUM) et avons résolu ce problème plus efficacement. Nous avons étudié la propriété de diffusion dans les réseaux de Feistel généralisés améliorant ainsi leur conception. Ensuite, nous avons proposé un outil pour générer automatiquement des modèles CP utiles pour la cryptanalyse différentielle. Enfin, nous nous sommes intéressés aux solveurs CP eux-mêmes en proposant une technique pour générer des explications et ainsi améliorer leurs performances.

Title: Cryptanalysis of symmetric cipher using generic solvers

Keywords: Cryptanalysis, Symmetric Cipher, Constraint Programming

Abstract: Cryptography is a critical science for our connected societies. It involves the design, analysis and implementation of encryption algorithms. Analyzing ciphers is a mandatory step to ensure their security, but this task is often tedious. In symmetric cryptography, this analysis mainly focuses on finding *distinguishers*, properties that distinguish a ciphertext from a random message. Generic solvers are tools originally created to solve problems like planning or scheduling. They are regularly improved and have become good candidates to facilitate cryptanalysis.

This thesis focuses on the analysis and design of symmetric ciphers with the help of generic solvers, in particular constraint programming (CP) solvers. We have explored several leads to improve cryptanalysis techniques and solvers. We modeled as a graph the search for distinguishers (specifically the search for superpoly in the TRIVIUM cipher) and solved this problem more efficiently. We have studied the property of diffusion in generalized Feistel networks thus improving their design. Next, we proposed a tool to automatically generate CP models for differential cryptanalysis. Finally, we focused on CP solvers themselves by proposing a technique to generate explanations and thus improve their performance.