Cryptanalytic Time-Memory Trade-Off

Diane Leblanc-Albarel

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

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

THÈSE DE DOCTORAT DE

L'INSTITUT NATIONAL DES SCIENCES APPLIQUÉES DE RENNES

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

Par
Diane LEBLANC-ALBAREL

Cryptanalytic Time-Memory Trade-Off

Thèse présentée et soutenue à IRISA, le 12/10/2023
Unité de recherche : IRISA, CNRS
Thèse N° : 23ISAR 27 / D23 - 27

Rapporteurs avant soutenance :

Orr Dunkelman	Professeur, Haifa University, Israël
Orhun Kara	Associate Professor, IYTE Izmır Institute of Technology, Turquie

Composition du Jury :

Président :	Pierre-Alain Fouque	Professeur, Université de Rennes, IRISA, France
Examinateur·rice·s :	Orr Dunkelman	Professeur, Haifa University, Israël
	Orhun Kara	Associate Professor, IYTE Izmır Institute of Technology, Turquie
	Barbara Fila	Maître de Conférence, INSA Rennes, IRISA, France
	Bart Preneel	Professeur, KU Leuven, Belgique
	Philippe Oechslin	Docteur, Objectif-Sécurité, Suisse
Dir. de thèse :	Gildas Avoine	Professeur, INSA Rennes, IRISA, France
Co. encadrant :	Xavier Carpent	Assistant Professor, Nottingham University, Royaume-Uni

Thèse de doctorat

Diane LEBLANC-ALBAREL

Directeur de thèse: Gildas AVOINE

Co-encadrant: Xavier CARPENT

Version finale, 15 décembre 2023

Remerciements

I would like to thank all the members of my jury: Barbara, Bart, Orhun, Orr, Philippe, and Pierre-Alain for dedicating their time to my defense and reviewing my manuscript. Especially, thanks to the reviewers for the extra time they devoted to review and comment on it. Your valuable comments and suggestions have significantly improved the quality of this work.

Merci particulièrement à Xavier et Gildas. Vous êtes, chacun à votre manière, un exemple pour moi, et j'aimerais devenir une chercheuse aussi accomplie que vous.

Merci Xavier pour ton accueil chaleureux à Nottingham, pour ton soutien et tes conseils. Merci pour tous les cidres que tu m'as fait goûter et les discussions qui allaient avec. Merci pour tout le temps que tu m'as accordé. Désolée pour toutes les fois où je t'ai coupé la parole alors que tu essayais d'expliquer ton point de vue, et merci encore pour toutes ces après-midi de discussions et de réflexion.

Merci Gildas pour toute ton aide et ton soutien. Merci pour tes conseils et ton écoute. Merci pour ta bonne humeur. Merci énormément pour ton honnêteté et ta bienveillance. Merci de m'avoir accordé autant de liberté. Merci infiniment pour tout le temps que tu m'as donné. Ma thèse n'aurait pas été la même sans ton aide, tes conseils et ton investissement. Merci pour toutes les opportunités que tu m'as offertes, et pour toutes les après-midi de discussion que tu m'as accordées. Je te suis très reconnaissante et j'ai énormément apprécié travailler avec toi.

Merci Morgan pour ton soutien, ton aide, et ton honnêteté sans faille. Merci pour toutes les fois où, quand je passais le week-end à travailler, tu ne m'en voulais pas et me préparais de bons petits plats pour me remonter le moral.

Merci à toutes les personnes présentes à ma défense. Merci à ceux et celles qui auraient voulu venir, mais qui ont accepté de ne pas le faire. Merci à toutes les personnes qui m'ont soutenue, de près ou de loin, pendant ces 3 ans.

Contents

List of Contributions

[NLAB+21] Cyrius Nugier, Diane Leblanc - Albarel, Agathe Blaise, Simon Masson, Paul Huynh, Yris Brice Wandji Piugie. An Upcycling Tokenization Method for Credit Card Numbers. In Elisa Bertino, Haya Shulman, and Michael Waidner, editors, Computer Security Proceedings of the 18th International Conference on Security and Cryptography – *SECRYPT 2021*, pages 15–25. SCITEPRESS 2021.

[ACLA21] Gildas Avoine, Xavier Carpent, Diane Leblanc-Albarel. Precomputation for rainbow tables has never been so fast. In Elisa Bertino, Haya Shulman, and Michael Waidner, editors, Computer Security – *ESORICS 2021*, pages 215–234. Springer International Publishing.

[ACLA22] Gildas Avoine, Xavier Carpent, Diane Leblanc-Albarel. Rainbow Tables: How Far Can CPU Go? In *The Computer Journal*, pages bxac147. October 2022.

[ACLA23] Gildas Avoine, Xavier Carpent, Diane Leblanc-Albarel. Stairway To Rainbow. In Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security *ASIA CCS '23*, pages 215–234. page 286–299, New York, USA, 2023. Association for Computing Machinery.

In submission soon

Gildas Avoine, Xavier Carpent, Diane Leblanc-Albarel. Ascending Stepped Rainbow Tables.

Résumé en Français

1. Contexte

Aujourd'hui, toutes nos données personnelles - depuis nos photographies, nos dossiers médicaux, jusqu'à nos informations bancaires - sont traduites dans un langage complexe composé de '1' et de '0', un langage uniquement compréhensible par les machines et par les quelques experts qui le maîtrisent. Ce basculement du monde physique au monde numérique a transformé nos informations personnelles en flux de données. La protection de ces flux contre les menaces potentielles et les acteurs malveillants nécessite des solutions de sécurité sophistiquées.

Les cryptographes, acteurs principaux de cette protection, conçoivent et implémentent des algorithmes de *chiffrement cryptographique* conçus pour être extrêmement difficiles à briser, et qui sont le fondement de la protection de nos données numériques, y compris sur Internet.

De manière générale, les chiffrements cryptographiques reposent sur des clés, que l'on peut comparer à de très longs mots de passe aléatoires. Le terme "cryptographie symétrique" est utilisé, lorsqu'une même clé sert à *chiffrer* et à *déchiffrer* l'information. Quand on utilise une clé différente pour chiffrer et déchiffrer, on parle de cryptographie asymétrique. Les cryptographes font constamment face aux *cryptanalystes*, qui tentent de briser les chiffrements en trouvant des failles dans leur conception. Cette thèse se concentre sur un outil utilisé par les cryptanalystes : Les Compromis Temps-Mémoire Cryptanalytiques (*Time Memory Trade-Off, TMTO*).

Le dernier recours des cryptanalystes est l'attaque par *recherche exhaustive*, qui consiste à essayer toutes les clés possibles pour récupérer une information chiffrée. Cependant, cette approche se révèle souvent inefficace en raison du nombre important de possibilités. Les attaques par *dictionnaire* sont une alternative consistant à calculer un maximum de possibilité par avance pour pouvoir déchiffrer une information très rapidement le moment venu. Dans ce cas, le problème n'est plus le temps, mais la mémoire requise pour stocker toutes ces possibilités, rendant l'attaque irréaliste dans la plupart des scénarios.

Un bon chiffrement implique plus de 2^{128} clés possibles, quantité très importante, entravant l'attaque par recherche exhaustive, et le stockage des clés dans un dictionnaire. Les TMTO constituent un compromis se situant exactement entre la recherche exhaustive et l'attaque par dictionnaire. On peut définir les TMTO comme un ensemble d'algorithmes spécifiquement conçus pour attaquer des *fonctions à sens unique*. Ils sont principalement connus pour leur utilisation contre un sous-ensemble de ces fonctions : les fonctions de hachage, dont le résultat est appelé un *haché*. La nature à sens unique des fonctions de hachage est ce qui fait leur particularité : alors qu'il est simple de calculer le haché d'une entrée donnée, il est difficile de retrouver l'entrée d'origine uniquement à partir du haché.

Les TMTO constituent un outil permettant de retrouver l'entrée d'une fonction à sens

unique, plus rapidement que par la recherche exhaustive. Leur utilisation est divisée en deux phases :

1. Une *phase de précalcul* durant laquelle des entrées possibles de la fonction de hachage cible sont calculées. Cette phase est très longue, bien plus longue qu'une recherche exhaustive mais se fait en amont de l'attaque en elle-même.

2. Une *phase d'attaque* qui consiste à utiliser les données précalculées pour retrouver rapidement l'entrée correspondant au haché.

2. Compromis Temps-Mémoire Cryptanalytiques

2.1. Problème traité

Une fonction de hachage, notée h, prend une entrée de taille variable et renvoie une chaîne d'octets de taille fixe. La transformation peut être exprimée ainsi :

$$h : \{0,1\}^* \to \{0,1\}^n$$

L'ensemble $\{0,1\}^*$ représente l'ensemble de toutes les entrées possibles et $\{0,1\}^n$ est l'ensemble des hachés de n bits possibles. Le symbole $*$ indique que la longueur du message d'entrée est arbitraire, tandis que n est la longueur fixe du haché.

Il est important de noter que h est déterministe, ce qui signifie que pour une entrée donnée, elle produit toujours le même résultat. Cependant, en raison de sa nature *à sens unique*, étant donné un haché, il est impossible, dans un temps raisonnable, de retrouver l'entrée correspondante. Cette propriété est connue sous le nom de *résistance à la pré-image*.

Pourtant, l'objectif d'une cryptanalyse consiste souvent à trouver la pré-image d'une fonction à sens unique. C'est dans ce but que les TMTO sont utilisés. Leur finalité est de trouver la pré-image d'un haché donné sans effectuer de recherche exhaustive.

Les TMTO utilisent de la mémoire pour stocker un ensemble d'entrées possibles sous la forme de *tables*. Cela réduit considérablement le temps nécessaire pour trouver la pré-image pendant la phase d'attaque. Plus on alloue de mémoire pour stocker les données précalculées, moins la phase d'attaque est longue, d'où le concept de compromis temps-mémoire.

2.2. Objectif

Étant donné un espace de recherche A et un espace d'images B d'une fonction à sens unique h, le problème que les TMTO visent à résoudre est de trouver x^* avec $x^* \in A$, tel que $h(x^*) = y$, pour la fonction h, et une image y.

En notant $N = |A|$, l'attaque par force brute coûte $O(N)$ opérations de hachage pendant la phase d'attaque sans nécessiter aucun précalcul ni stockage. L'utilisation d'un dictionnaire, elle, coûte $O(N)$ opérations de hachage pendant une phase de précalcul ainsi que $O(N)$ en stockage, la phase d'attaque étant elle, seulement en $O(1)$ (instantanée).

Un TMTO est un compromis entre les deux : avec M mémoire, la phase d'attaque a une complexité de $O(N^2/M^2)$ pour les cas typiques [BBS06] et la phase de précalcul a une complexité de $O(N)$. Cela signifie qu'utiliser un TMTO a du sens dans les scénarios suivants :

- L'attaque doit être effectuée un grand nombre de fois.

- L'attaque doit être très rapide mais l'attaquant dispose d'autant de temps qu'il le souhaite pour se préparer (cas de la "lunch time attack"[1]).
- La puissance de calcul dont dispose l'attaquant n'est pas suffisante pour effectuer une recherche exhaustive, mais il a accès au résultat d'un précalcul et peut l'utiliser pour une attaque.

Bien que les TMTO soient généralement perçus comme un compromis entre le temps d'attaque et la mémoire, d'autres caractéristiques sont importantes, notamment la *couverture* et le *temps de précalcul*.

La *couverture* est la proportion d'entrées de A représentée dans les données générées lors de la phase de précalcul. Une couverture élevée implique une forte probabilité de succès pour l'attaque. À l'inverse, une faible couverture conduira à une probabilité de succès inférieure. Plus la couverture est élevée, plus la phase de précalcul est longue, car davantage d'applications de la fonction de hachage doivent être effectuées pour générer des tables avec un nombre plus important de pré-images.

2.3. Utilisation d'un TMTO

2.3.1. Phase de précalcul

Lors de la phase de précalcul, une ou plusieurs matrices composées d'éléments de l'espace A sont créées. À la fin de cette phase, seules les premières et dernières colonnes de chaque matrice sont conservées, tous les autres éléments sont éliminés. On appelle *table*, les éléments de la première et dernière colonne conservés à la fin de la phase de précalcul. Ce sont ces tables qui sont stockées puis utilisées lors de l'attaque. Le stockage des tables précalculées présente un coût en mémoire appelé M.

Les différentes variantes des TMTO se distinguent par les méthodes de calcul et les formes des matrices générées. Selon la variante utilisée, les matrices peuvent avoir des lignes avec un nombre variable de colonnes. Dans ce cas, seuls les premiers et derniers éléments de chaque ligne sont stockés une fois la phase de précalcul achevée.

Le temps requis pour la phase de précalcul est dépendant de la variante utilisée et de la couverture ciblée. Par exemple, générer un TMTO qui couvre seulement 70% de l'espace est moins coûteux qu'un TMTO qui couvre 90%. Au-delà d'un certain taux de couverture, une augmentation de celui-ci augmente fortement le coût de précalcul. Par exemple, augmenter le taux de couverture de 99% à 99.9% requiert beaucoup plus de ressources que de 71% à 71.9%.

En définitive, le temps requis pour le précalcul dépasse N, allant de 3 à 4 fois N pour les variantes améliorées (pour une couverture faible) à des centaines, voir des milliers, de N pour d'autres variantes.

2.3.2. Phase d'attaque

Les tables générées durant la phase de précalcul sont utilisées pour l'attaque. Indépendamment de la variante considérée, cette phase suppose que, l'élément recherché x^*, se trouve dans la matrice précalculée. Si cette hypothèse est vérifiée, il devient possible de transformer y jusqu'à ce qu'il corresponde à un élément stocké dans la table précalculée, c'est à dire un

[1] https://en.wikipedia.org/wiki/Chosen-ciphertext_attack#Lunchtime_attacks

élément de la dernière colonne de la matrice. En récupérant l'élément correspondant de la première colonne, la ligne est recalculée jusqu'à ce que x^* soit atteint (voir le chapitre 2 pour plus de détails).

L'aspect le plus chronophage de cette phase est d'éliminer les faux positifs. Cependant, le temps moyen de l'attaque, T est généralement de l'ordre de $T = N^2/M^2$ (voir le chapitre 8 pour une discussion à ce sujet). Une configuration typique est de choisir $M = N^{2/3}$, ce qui implique un temps d'attaque T de l'ordre de $T = N^{2/3}$, considérablement plus efficace que la recherche exhaustive.

3. Cas d'utilisation

3.1. Cassage de mots de passe

Les TMTO sont, et ont été, largement utilisés pour le cassage de mots de passe, par exemple, dans les cassages des mots de passe Windows LAN Manager (LM) [Oec03] ou Unix [MBPV06]. En réponse à ces dernières attaques, diverses contre-mesures ont été introduites pour compliquer l'utilisation des attaques précalculées dont font partie les TMTO.

La contre-mesure la plus impactante a été l'ajout d'un *sel* ("salt" en anglais) aléatoire ajouté au mot de passe avant le hachage. C'est une chaîne de bits aléatoire, ayant une taille généralement comprise entre 8 et 64 bits. Cette chaîne est concaténée avec le mot de passe avant d'être haché. Cela garantit que même si deux utilisateurs ont le même mot de passe, leurs hachés correspondants seront différents si deux sels différents ont été utilisés. Si l'utilisation du sel est bien implémentée, il n'est révélé qu'au cours de la phase d'attaque et est donc inconnu lors de la phase de précalcul. Par conséquent, pour tenir compte des sels, un TMTO doit être généré pour chaque sel possible, ce qui entraîne une augmentation du temps de précalcul jusqu'à un facteur pouvant aller jusqu'à 2^s, où s est la longueur du sel. Dans la plupart des cas, l'utilisation de sel rend donc les TMTO inutilisables en raison de son impact sur la phase de précalcul.

Malgré l'introduction du sel pour protéger les services contre les attaques précalculées, et notamment contre les TMTO, une partie importante des mots de passe reste non salées. Les TMTO continuent donc d'être utilisés. En particulier, ils sont fréquemment utilisés par des testeurs d'intrusion ("pentester" en anglais) afin d'identifier les mots de passe potentiellement faibles dans les bases de données compromises. Leur application s'étend également à la cybercriminalité, lorsqu'il est nécessaire d'inverser des données hachées pour récupérer des informations.

3.2. Autres Utilisations

Depuis l'introduction des TMTO par Hellman en 1980 [Hel80], des variantes de la construction originale de TMTO ont été développées. Certaines ont été conçues pour être appliquées à des chiffrements par flux [BS00, Bab95, Gol97], ou généralisées pour se servir de données supplémentaires (ces variantes sont appelées Time-Memory-Data Trade-Off [BMS05]). De plus, des vulnérabilités ont été découvertes dans d'autres chiffrements tels que LILI-128 [SRQL02] et Toyocrypt, destinés à être utilisés par le gouvernement japonais, grâce à des attaques réussies de TMTO [KGL06, KCGL09]. Par ailleurs, trois candidats eSTREAM, Grain [HJMM08], Lex [DK08a], et MICKEY [HK05], ont également été attaqués à l'aide de TMTO, provoquant des modifications considérables dans la conception de ces chiffrements. Les TMTO sont

également utilisés pour d'autres applications spécifiques, notamment lorsque l'espace d'entrée n'est pas uniformément distribué [ACL15, Hoc09], en utilisant une mémoire externe [ACKT17, KHP13], un FPGA [MBPV06, SRQL02] ou un GPU [KSH+15, LLH15].

4. Problématique

Ces dernières années, divers facteurs, tels que l'utilisation de sel ou de fonctions de hachage lentes, ont rendu les TMTO plus difficiles à utiliser. Toutefois, l'utilisation et la compréhension des TMTO pourraient également ne pas avoir suffisamment évolué pour augmenter suffisamment leur efficacité. En particulier, le temps de précalcul requis pour gérer les espaces utilisés aujourd'hui est devenu excessivement coûteux.

Le grand défi de cette thèse est de mettre en évidence les limites actuelles des TMTO, de suggérer des solutions pratiques et des variantes alternatives pour repousser ces limites, et enfin de fournir une compréhension plus approfondie des TMTO.

Cette thèse vise donc trois objectifs :

1. Identifier les facteurs limitant une application plus large des TMTO.

2. Proposer des améliorations et de nouvelles variantes de TMTO qui pourraient efficacement effacer ou repousser ces limites.

3. Introduire une nouvelle perspective sur le problème des TMTO. Jusqu'à présent, l'accent principal des TMTO a été mis sur la mémoire et le temps d'attaque. Ce travail propose cependant une stratégie plus complète, prenant également en compte la couverture et le temps de précalcul. Ces facteurs doivent être considérés comme des composantes intégrales des TMTO, de la même manière que la mémoire et le temps d'attaque.

5. Structure de la thèse

Le chapitre 2 présente les bases nécessaires à la compréhension de la thèse. Les différentes variantes les plus connues des TMTO sont présentées, notamment les Tables de Hellman, les Tables de Points Distingués, les Tables Arc-en-ciel et les Tables Arc-en-ciel Brumeuses. Le choix d'étudier particulièrement la variante des Tables Arc-en-ciel est justifié, et une analyse approfondie des variantes de celles-ci ainsi que leurs améliorations est proposée.

Le chapitre 3 présente les résultats préliminaires utilisés dans tous les chapitres ultérieurs de la thèse. Certains de ces résultats ont probablement été utilisés officieusement avant cette thèse, mais ils ont été officiellement introduits dans un premier article publié [ACLA21] durant cette thèse.

Le chapitre 4 présente la méthode de filtration distribuée. Cette technique est spécifiquement conçue pour diminuer de manière significative le temps de précalcul requis pour générer des Tables Arc-en-ciel. Ce chapitre présente les complexités de la distribution du précalcul lors de la mise en place de la filtration, et introduit une manière efficace de gérer cette complexité. Des expériences réalisées sur différents environnements sont présentées pour vérifier l'efficacité de cette méthode.

Le chapitre 5 identifie la limite actuelle empêchant une utilisation plus large des Tables Arc-en-ciel, à savoir le temps de précalcul. Ce chapitre propose une analyse comparative

de différents scénarios utilisant différents environnements, en examinant l'espace maximal atteignable dans un laps de temps donné ou pour un investissement monétaire donné.

Le chapitre 6 présente ensuite une nouvelle variante des Tables Arc-en-ciel appelée *Descending Stepped Rainbow Tables* (Tables Escaliers Arc-en-ciel Descendantes). Cette variante repousse la limite identifiée dans le chapitre 5 en introduisant une variante qui surpasse les Tables Arc-en-ciel classiques. De plus, ce chapitre propose de comparer différentes variantes non seulement en tenant principalement compte de leur temps d'attaque et de leur mémoire, mais aussi en considérant leur couverture et temps de précalcul dans le compromis global.

Le chapitre 7 introduit une seconde nouvelle variante : les *Ascending Stepped Rainbow Tables* (Tables Escaliers Arc-en-ciel Ascendantes). Cette variante surpasse les Tables Escaliers Arc-en-ciel Descendantes dans certains cas, en particulier lorsque des couvertures élevées sont visés. Tout comme pour les Tables Escaliers Arc-en-ciel Descendantes, une comparaison prenant en compte le temps de précalcul et la couverture dans l'évaluation des variantes est proposée.

Le chapitre 8, conclue sur les contributions de la thèse, aborde les problèmes ouverts identifiés ainsi que les travaux futurs à réaliser.

1 Introduction

Why did the TMTO algorithm leave the password party buffet immediately after tasting the food? ***Because it was too salty!***

1.1. Context

Today, all our personal data — from our photographs and health records to our banking details — are translated into a complex language of ones and zeros, a language that can only be understood by machines and the few who understand it. This shift from the physical to the digital world has transformed our personal information into binary data streams. Protecting these data flows against potential threats and malicious actors requires sophisticated security solutions.

Cryptography is one of the preferred solutions against these threats. By designing and implementing cryptographic algorithms, or *ciphers*, cryptographers play a key role in this protection. These ciphers, designed to be extremely difficult to break, are the foundation of our digital data protection, especially on the Internet.

Generally speaking, cryptographic ciphers are based on keys, which can be compared to very long random passwords. The term symmetrical cryptography is used when the same key is used to (*encrypt*) and (*decrypt*) information. When a different key is used to encrypt and decrypt, we speak of asymmetric cryptography.

Cryptographers constantly have to contend with *cryptanalysts*, who try to break ciphers by finding breaches in their design.

This thesis focuses on a tool used by cryptanalysts: Cryptanalytic Time Memory Trade-Offs (TMTOs). Cryptanalysts last resort is the brute-force attack (an *exhaustive search*), which involves trying every possible key to recover encrypted information. However, this approach often proves ineffective due to the sheer number of possibilities. Dictionary attacks are an alternative, in which as many possibilities as possible are computed in advance, so that information can be decrypted very quickly when needed. In this case, the problem is no longer time, but the memory required to store all these possibilities, making the attack unrealistic in most scenarios.

Good encryption involves more than 2^{128} possible keys. This is very time-consuming to attack by brute force, and very difficult to store in a dictionary. TMTOs are an effective compromise between exhaustive search and dictionary attack.

1.1.1. Concept

TMTOs are a set of algorithms specifically designed to attack one-way functions. They are most known for their use against a particular category of one-way functions: the hash

functions. A hash function is a function that transforms the input into a new form, called a *hash value*. This transformation is unique such that any minor alteration to the input results in a drastically different hash value.

The one-way nature of hash functions is what makes them distinctive. While it is simple to compute the hash of a given input, it is computationally challenging, or even impossible, to retrieve the original input based solely on the hash.

TMTOs offer a more efficient alternative than to brute-force these one-way functions. They provide a method to balance the computational time needed to reverse the hash function and the memory required to store potential original inputs. The use of TMTOs involves creating a matrix of possible inputs during a precomputation phase, which is subsequently used to quickly find the correct input during an actual attack phase.

1.1.2. Problem Addressed

A hash function, denoted as h, is a function that takes an input and returns a fixed-size string of bytes. The transformation can be expressed as:

$$h : \{0,1\}^* \rightarrow \{0,1\}^n$$

Where $\{0,1\}^*$ represents the set of all possible inputs , and $\{0,1\}^n$ is the set of all possible n-bit strings (the hash values). The $*$ denotes that the length of the input message can be arbitrary long, while n is the fixed length of the hash value.

It is important to note that the hash function h is deterministic, meaning that for a given input, it always produces the same hash value. However, due to its *one-way* nature, given a hash value, it is computationally infeasible to retrieve the original input. This property is known as *preimage resistance*.

In the context of cryptanalysis, the goal often becomes to find the preimage of a one way function. It is for this end that TMTOs are used. The purpose of TMTOs is to find the preimage (the original input) of a given hash value without exhaustively trying all possible inputs.

TMTOs use memory to store a certain amount of precomputed inputs, which drastically reduces the time needed to find the preimage during the attack phase. The more memory is allocated for storing precomputed inputs, the less time took the attack phase, hence the name Time-Memory Trade-Off.

1.2. Time Memory Trade-Off

1.2.1. Purpose

Given a searched space A and the one way function images space B, the problem that TMTO aims to solve is defined as: given a one way function $h : A \rightarrow B$, and an image $y \in B$, find $x^* \in A$ such that $h(x^*) = y$.

As mentioned before, this problem has two extreme solutions: (1) the *brute force attack*, where all preimages in A are tested sequentially; and (2) the *dictionary* or *precomputed attack*, where preimage-image pairs are computed and saved, and finding the preimage consists in a simple lookup. Denoting $N = |A|$, the former attack costs $O(N)$ hash operations during the attack phase but requires no precomputation or storage. The latter costs $O(N)$ hash

operations during a precomputation phase as well as $O(N)$ in storage, but costs nothing in the attack phase (barring the cost of the lookup).

A TMTO is a trade-off between the two: with M memory, the attack phase costs in the order of $O(N^2/M^2)$ for typical cases [BBS06] and the precomputation phase is in the order of $O(N)$. This means that using a TMTO makes sense in specific scenarios: the attack has to be performed several times, the attack itself has to last for a short period of time ("lunch time" attack[1]), or the attacker is not powerful enough to perform an exhaustive search but can download the result of a precomputed phase, stored in what is called *tables* and use them to perform the attack.

Although TMTO is usually perceived as a trade-off between attack time and memory, other characteristics are important, namely: coverage and precomputation time. The *coverage* refers to the proportion of A that is represented in the matrices generated during the precomputation phase. High coverage will result in a high success probability of the attack, while low coverage will lead to a lower success probability. The higher the coverage, the more time-consuming the precomputation phase becomes because more applications of hash function must be performed to generate tables with a larger number of represented preimages.

1.2.2. Overview

1.2.2.1. Precomputation Phase

In the precomputation phase, matrices composed of elements from space A are created. At the end of this phase, only the first and last columns of the matrix are stored for the attack phase, while all other matrix elements are discarded. The elements stored for the attack phase form the *tables*. A *table* is store for each matrix and is formed of the first and last columns of the corresponding matrix. At the end of the precomputation phase, the tables storage requires M memory.

The various TMTO variants are distinguished by their methods of matrix computation and the resulting matrix shapes. Depending on the variant used, rows may have varying numbers of columns. In such cases, the first and last elements of each row are stored at the end of the precomputation phase.

The time required for the precomputation phase depends on the specific variant used and the targeted coverage. For instance, generating a TMTO that covers only 70% of the space costs significantly less than generating a TMTO that covers 90%. Beyond a certain point, every incremental increase in coverage incurs a higher increase of the computation cost. For instance, increasing coverage from 99% to 99.9% requires substantially more resources than raising it from 71% to 71.9%.

Ultimately, the time required for precomputation exceeds N, ranging from 3 to 4 times N for improved variants (with no extremely high coverage) to hundreds or even thousands of times N for other variants. Coverage is not the only factor affecting precomputation time. The attack time resulting from the precomputed matrix also influences it: tables that allow faster attacks tend to require more precomputation. Conversely, tables that are quickly precomputed might demand more memory or result in slower attack phases.

[1] https://en.wikipedia.org/wiki/Chosen-ciphertext_attack#Lunchtime_attacks

1.2.2.2. Attack Phase

During the attack phase, tables generated during the precomputation phase are used. For all variants, the attack assumes for each table that the searched element x^* is in the matrix corresponding to the table. If this assumption holds, it becomes possible to transform y until it matches an element in the last column of the matrix — i.e., an element that has been stored in a table. From this point, by retrieving the corresponding element from the first column (also stored in the table), the row is precomputed until x^* is reached (See Chapter 2 for more details).

The most time-consuming aspect of this phase is ruling out the false positives. However, despite this, the average time for the attack phase, T, is typically in the order of $T = N^2/M^2$ (refer to Chapter 8 for a discussion on this point). A typical configuration is $M = N^{2/3}$, which results in an attack time T in the order of $T = N^{2/3}$. This is considerably faster than performing an exhaustive search.

1.3. Use Cases

1.3.1. Password Cracking

TMTOs have a significant role in password cracking applications, such as in the breaching of Windows LAN Manager (LM) passwords [Oec03] and Unix passwords [MBPV06]. In response to such attacks, various countermeasures have been introduced to complicate the use of precomputed attacks.

The most substantial countermeasure has been the implementation of a random *salt* added to the password prior to hashing. The salt, a random bit string typically ranging from 8 to 64 bits in size, is concatenated with the password before it is hashed. This ensures that even if two users have the same password, their hashes will differ due to the unique salts. If well implemented, the salt is only revealed during the attack phase, and is unknown during the precomputation phase. Therefore, to account for the salts, a TMTO must be generated for every possible salt, leading to an increase in precomputation time by a factor up to 2^s, where s is the length of the salt, which leads, most of the time, to unusable TMTOs due to the extra cost in precomputation.

Despite the introduction of salt to protect services against precomputed attacks, notably TMTO attacks, a considerable portion of passwords remains unsalted. Hence, TMTOs continue to be used. In particular, they are frequently used by penetration testers for identifying potentially weak passwords within client databases. Their application also extends to digital forensics, where there may be a need to invert hashed data to retrieve information from data dumps.

1.3.2. Other Cases

Variants on Hellman's original construction have been developed over time. Some were designed to be applied to stream ciphers [BS00, Bab95, Gol97], or generalized to work with multiple data (so-called Time-Memory-Data Trade-Off [BMS05]). Furthermore, vulnerabilities were uncovered in other ciphers such as LILI-128 [SRQL02] and Toyocrypt, intended for use within the Japanese government, through successful TMTOs attacks [KGL06, KCGL09]. Three eSTREAM candidates, Grain [HJMM08], Lex [DK08a], and MICKEY [HK05], were also attacked using TMTOs, leading to substantial modifications in their design. TMTOs

are also use for others specific applications, such as when the input space is not uniformly distributed [ACL15, Hoc09], using external memory [ACKT17, KHP13], or FPGA [MBPV06, SRQL02] and GPU [KSH+15, LLH15].

Real-life attacks on Digital Signature Transponders (DSTs), used in electronic payments and vehicle immobilizers, were made feasible with TMTOs. Two additional vehicle immobilizers, Hitag2 [VGB12], and Megamos Crypto [VMG+13], have been successfully attacked using TMTOs.

Recently, an attack on WPA3 using TMTOs have been proposed [Van22], and a proposition of quantic adapted attack phase for various TMTOs variants have been proposed [DKRS21].

1.4. Time-Memory-Data Trade-Off

Time-Memory-Data Trade-Offs (TMDTOs) extend TMTOs by adding an additional factor D into the trade-off, representing supplementary data for attack execution. Mainly applied for stream cipher attacks, TMDTOs typically aim to retrieve one of the targeted elements within a set of D instances.

A notable implementation of TMDTO is the Babbage-and-Golic TMDTO [Bab95, Gol97]. Here, N stands for the total internal states of a bit generator, and D refers to the first pseudorandom bits generated. The algorithm is essentially about selecting, storing, and matching random states with their corresponding output prefixes during the attack phase, resulting in the identification of an internal state of the bit generator, thereby unlocking the remaining part of the key.

Subsequently of the Babbage-and-Golic TMDTO, a new TMDTO combining the Hellman and Babbage-and-Golic tradeoff attacks was proposed by Shamir and Biryukov [BMS05], yielding better bounds for stream cipher attacks. The precomputation time is in the order of $O(N/D)$, and the attack time in the order of $O(t^2)$, with t the number of columns in the matrix generated during the precomputation phase. This gives a general trade-off of $TM^2D^2 = N^2$ or $T = N^2/M^2D^2$ for $D^2 \leqslant T \leqslant N$.

TMDTOs represent an important extension of TMTOs and are useful in various cryptographic scenarios, particularly in attacking stream ciphers. However, while undoubtedly interesting, TMDTOs are out of scope of this thesis.

1.5. Problem Statement

Various factors, such as the use of salt or slow hash functions, have made TMTOs more challenging to use than in the past years. However, the approach to TMTOs may not have evolved sufficiently to increase their efficiency and tackle their current limitations. In particular, the precomputation time required for managing the spaces under consideration has become overly computationally demanding given today practical spaces.

The grand challenge of this thesis is to highlight the current limitations of TMTOs, suggest practical solutions and alternative variants to overcome these limitations, and provide a more comprehensive understanding of TMTOs.

To complete this challenge, the thesis targets three purposes.

The first is to identify the current bottlenecks that restrict broader application of TMTOs.

The second is to propose improvements and new TMTOs variants, which could effectively address these bottlenecks.

The third is to introduce a novel perspective on the TMTO problem. Indeed, until now, the primary emphasis on TMTOs has been on memory and attack time. This work, however, proposes a more comprehensive strategy, which also takes into account the coverage and precomputation time. These factors should be considered as integral components of TMTOs, just like memory and attack time.

1.6. Thesis Structure

This thesis is organized as follows. Chapter 2, presents the necessary background to the comprehension of the thesis. The most known different variants of TMTOs are presented, namely the Hellman Tables, Distinguished Points Tables, Rainbow Tables and Fuzzy Rainbow Tables. The choice of the study of the Rainbow Tables variant in particular is justified and a deeper analysis of the Rainbow Tables variants and their improvements are proposed.

In Chapter 3, preliminary results used in all the subsequent chapters of the thesis are presented. Some of these results were probably used unofficially before this thesis but were formally introduced in the first paper published during this thesis [ACLA21]. Concepts and quantifier values introduced in this chapter are then used in each chapter of the thesis.

Chapter 4 introduces the distributed filtration method. This technique is specifically designed to significantly decrease the precomputation time required for generating Rainbow Tables. This chapter presents the complexities of distributing precomputation when using the filtration method and introduces an efficient way to deal with it. It presents experiments performed in various environments to verify the efficiency of the method.

Chapter 5 identifies the current bottleneck limiting the use of Rainbow Tables, namely, the precomputation time. It states that it is the precomputation time that prevents attacks on larger spaces when CPUs are employed. This chapter offers a comparative analysis across different scenarios and environments, investigating the maximum achievable space within a set time frame or for a specified monetary investment.

Chapter 6 then presents a new Rainbow Table variant named Descending Stepped Rainbow Tables. This variant addresses the particular bottleneck identified in Chapter 5 by introducing a variant that outperforms Rainbow Tables. In addition, this chapter proposes to compare different variants not just by considering mainly their attack time and memory requirements but also their coverage and precomputation time as part of the whole trade-off.

Chapter 7 introduces another new variant: the Ascending Stepped Rainbow Tables. This variant outperforms the Descendig Stepped Rainbow Tables in some cases, in particular at high coverage. As for the Descending Stepped Rainbow Tables, we propose a comparison that takes also the precomputation time and coverage into account in the comparison of the variants.

In Chapter 8, we conclude on the contributions of the thesis, and delve into the open problems identified in the thesis and future works to carry out.

2 Background

A PhD student working on TMTO seems in the moon. Her supervisor, asks her "where are you at?" *Somewhere over the rainbow*

2.1. Overview

This chapter lays the foundation necessary for understanding the research conducted in this thesis. As previously mentioned in Chapter 1, Time-Memory-Data Trade-Offs (TMDTOs), though substantial in the TMTO literature and extensively discussed, analyzed, and employed in attacking various stream ciphers (as in [BS00, LLH15, BD00, BSW00, Gol97, MFI07, CK08, DK08b, KCGL09, BMS05, EK15, Saa02]), are not the focus of this thesis. Hence, their details would not be extended beyond what was already presented in Chapter 1.

The purpose of this chapter is to present and explain specific TMTO variant that has been examined in this thesis, as well as to clarify the reasoning behind its selection. In addition, it aims to compare various TMTO variants, thereby highlighting the motivations driving this research.

It is essential to stress that there is an ongoing debate in the scientific community concerning the most efficient TMTO variant. As a result, providing a clear-cut overview of the advantages and drawbacks of each variant proves to be a challenging task. Nevertheless, this chapter attempts to provide a preliminary assessment following the presentation of each variant. A more thorough comparison of all variants is proposed in Chapter 8, as a potential area for future research.

Takeaway:

- This chapter introduces various TMTO: Hellman tables, Distinguished Points tables, Rainbow tables, and Fuzzy Rainbow tables.
- While there is no consensus on the most efficient TMTO variant, the Rainbow tables variant has been selected for analysis in this thesis.
- Formulas for evaluating and characterizing the state of the art in Rainbow tables are provided.

2.2. Original TMTOs: Hellman Tables

Hellman tables are the first variant of TMTOs and were introduced by Martin Hellman in 1980 [Hel80]. The primary idea is to generate tables during a precomputation phase. These tables take M memory and are then used during the attack phase.

2.2.1. Precomputation Phase

> **Reminder:**
>
> - A is the space of the searched elements x^*.
> - B is the space of the considered hash function.

The precomputation phase consist of the generation of a *matrix* with $t+1$ columns and m rows. Only the first and last columns of the matrix are kept to form the *table* used during the attack. Thus, the term Hellman *matrix* refers to the full matrix containing m rows and $t+1$ columns, while Hellman *table* refers only to the first and last columns.

The generation of the matrix begins with the selection of m elements from the N possible elements in the search space A. These m elements form the first column of the matrix. For each of these elements, a so called *hash-reduction* function f is applied to compute the next element in the row. The hash-reduction function f is defined as follows:

$$\begin{array}{rccc} f & : & A & \rightarrow & A \\ & & x_{i+1} & \mapsto & R(h(x_i)) \end{array} \tag{2.1}$$

The function h denotes the one-way function[1] employed in the TMTO. The function R is called the *reduction function*. It maps an element from the hash space B into an element in A either uniformly or nearly uniformly (the reduction function typically executes a modulo operation over N [AC17, ACL15, HM13]. Hence, the function is classified as *nearly* uniform because if $|B| = r + dN$, then r elements in N will be represented once more than the others). The reduction function must be as fast as possible to minimize both precomputation and attack time. Typically, a modulo function is chosen as the reduction function.

Figure 2.1 illustrates a Hellman matrix.

$$\begin{array}{|c|} \hline \boldsymbol{x_{1,0}} \\ \boldsymbol{x_{2,0}} \\ \vdots \\ \boldsymbol{x_{m,0}} \\ \hline \end{array} \begin{array}{c} \xrightarrow{f} \\ \xrightarrow{f} \\ \\ \xrightarrow{f} \end{array} \begin{array}{c} x_{1,1} \\ x_{2,1} \\ \vdots \\ x_{m,1} \end{array} \begin{array}{c} \xrightarrow{f} \\ \xrightarrow{f} \\ \\ \xrightarrow{f} \end{array} \begin{array}{c} x_{1,2} \\ x_{2,2} \\ \vdots \\ x_{m,2} \end{array} \begin{array}{c} \cdots \\ \cdots \\ \\ \cdots \end{array} \begin{array}{c} \xrightarrow{f} \\ \xrightarrow{f} \\ \\ \xrightarrow{f} \end{array} \begin{array}{|c|} \hline \boldsymbol{x_{1,t}} \\ \boldsymbol{x_{2,t}} \\ \vdots \\ \boldsymbol{x_{m,t}} \\ \hline \end{array}$$

Figure 2.1. – Hellman Matrix

[1] In [FN91], the authors show that other functions than one-way functions can be used in TMTOs, however, this comes at the cost of significantly reduced performance.

The rows of the matrix are referred to as *chains*, the *length* of a chain is the number of applications of the hash-reduction function required to compute the chain. Thus, a chain i of length t is the list $\{x_{i,0}, x_{i,1}, x_{i,2}, ..., x_{i,t}\}$ that contains $t+1$ elements. Similarly, a *sub-chain* is a sub-set of consecutive elements of a chain. For instance, $\{x_{i,0}, x_{i,1}, x_{i,2}, ..., x_{i,j}\}$ is a sub-chain of length j, containing $j+1$ elements, with $j < t$.

> **Reminder:** The *coverage* refers to the part of A that is represented in the matrices generated during the precomputation phase.

The elements in the first column of the matrix are called *start points* (SPs), while the ones in the last column are referred to as *end points* (EPs). These terms are used for the other TMTO's variants as well. In literature *end points* are also referred to as *endpoints* or *ending points*. To increase the coverage, ℓ tables are generated instead of a single one (see Section 2.2.3 for more details).

The precomputation phase of Hellman matrices, or of TMTO's more generally, consumes a substantial amount of time on the order of $O(CN)$. Here, C varies according to the variant and the parameters used, ranging between 6 and several hundreds or thousands. As stated in [HKR83], it is impossible to generate a TMTO table that outperforms an exhaustive search in less than N operations.

2.2.2. Attack Phase

> **Reminder:**
>
> - Given a value y and a one-way function h, where $y = h(x)$, the goal of a TMTO attack is to efficiently retrieve x from y using the tables generated during the precomputation phase.
> - The value x that needs to be determined is referred to as the *searched element*.

The attack begins by assuming that the searched element x is located in the penultimate column of the first table. If this assumption is true, $R(y)$ will be equal to one of the EPs of the first table. Consequently, the attacker computes $R(y)$ and checks for a match with the EPs. If a match is found, the corresponding SP of the matched EP is retrieved and the chain starting from this SP is build again until column $t-1$. In this column, the attacker retrieves the element $x_{i,t-1}$, where i refers to the corresponding row of the chain and $t-1$ represents its column. The attacker then computes $h(x_{i,t-1})$ and checks if $h(x_{i,t-1}) = y$. If $h(x_{i,t-1}) = y$, the searched element is $x_{i,t-1}$. If $h(x_{i,t-1}) \neq y$ or if there is no match between $R(y)$ and any EPs of the first table, the attacker supposes that the searched element is in the second to last column of the table. Consequently, they compute $f(R(y))$ by applying the hash-reduction function f to the already computed value $R(y)$. If the searched element is indeed in the second to last column, a match will be found between $f(R(y))$ and one of the EPs of the table. If such a match occurs, the attacker repeats the previous process, reconstructing the corresponding chain up to column $t-2$.

The term *attack chain* is used to denote the chain that hypothetically begins at the pre-image of y and ends in column t, regardless of its length.

The process of assuming that the searched element is in a certain column c of the matrix and performing computations to verify this hypothesis is referred to as a *search* in column c.

The attack process continues by searching in all columns in a similar manner until the

searched element is found or until all tables have been explored. Once the end of the table is reached, the attacker proceeds to the next table.

2.2.2.1. False Alarm

A match between the attack chain $f(f(\dots R(y)))$ and an EP leads the attacker to rebuild the corresponding chain up to the column j, where j is the column of the search. However, if $h(x_{i,j}) \neq y$, where i is the row of the matched EP, it results in a *false alarm*.

False alarms are primarily due to collisions amongst elements of A when applying the surjective function f. False alarms are very common and unfortunately decrease the attack phase time significantly.

2.2.3. Coverage

The maximum coverage of a Hellman matrix, $p_H^{\max}$, is reached when all elements within the matrix are distinct, as depicted in Equation (2.2) borrowed from [AJO08].

$$p_H^{\max} = \frac{mt}{N} \tag{2.2}$$

Nonetheless, a primary challenge when using Hellman tables is that the actual coverage of a Hellman table, denoted by p_H, is significantly lower than the maximum coverage $p_H^{\max}$, because far less than mt distinct elements are, on average, present in the matrix. This arises due to a phenomenon called *merges*.

During the generation of a Hellman matrix, *merges* occur between chains. Merges are formally defined in Definition 2.1, they appears when two distinct chains become identical after some point. Given that the hash-reduction function is surjective and returns a random element in A, different elements across different columns and rows can be succeeded by the same element.

Definition 2.1. *For two chains i and i', with $x_{i,j} \neq x_{i',j'}$, a merge occurs when $x_{i,j+1} = x_{i',j'+1}$, where $j \leqslant j'$. The elements between $x_{i,j+1}$ and $x_{i,j+1-t-j'}$ are thus equal to the elements between $x_{i',j'+1}$ and $x_{i',t}$.*

In case of merge, instead of having $2t - j - j'$ distinct elements, only $t - j$ distinct elements are represented in the two sub-chains $\{x_{i,j}, ..., x_{i,t}\}$ and $\{x_{i',j'}, ..., x_{i',t}\}$.

Merges reduce the coverage of a Hellman matrix substantially, implying that the matrix will contain significantly fewer distinct elements than mt. Merges are so frequent that the coverage of a Hellman table is typically less than 1% [Hel80, KM96] for standard memory and parameter settings.

In order to increase the coverage, ℓ tables are used instead of one table. This is achieved by generating ℓ matrices, each employing a *distinct reduction function*, from the other matrices. If an element of a table ℓ_i equals an element from a different table ℓ_j, except in some extremely rare cases, this would not result in a merge chain situation[2], as the next applied reduction function will differ. Therefore, with the use of ℓ matrices, the maximum coverage of a Hellman table is provided by Equation (2.3).

In [KM96, MH09], the authors show that under the condition $mt^2 = N$ (i.e., generating $\ell = t$ Hellman table) the maximum coverage of Hellman trade-off is approximately 80%.

[2] A merge situation is possible with a probability $(\frac{1}{N})^{(t-c)}$, where c is the column of the first equality.

$$p_{\ell H}^{\max} = 1 - (1 - p_H^{\max})^{\ell} \tag{2.3}$$

2.2.3.1. Clean Hellman Table

As presented above, to achieve the maximum coverage of a Hellman table, it is necessary to generate tables with elements that are all distinct. This corresponds to deal with a matrix *free of merges*. Such a matrix is called a *clean* matrix. In such Hellman matrix, mt distinct elements are thus present in the matrix with no merged chains.

These types of matrices are significantly resource-intensive to generate as they require that all elements within the matrix are checked for merges. As suggested by the authors in [AJO08], a strategy for building clean Hellman tables could be to compute chains of maximal length without loop[3] and then truncate them to length t in order to assemble them as a clean Hellman matrix.

Depending on the specific situation, the utility of using clean Hellman tables as opposed to non-clean ones can be considered unjustified due to the precomputation extra cost it involves.

2.2.4. Precomputation Cost

The precomputation time or cost of Hellman tables can be estimated by the number of applications of the hash-reduction function, needed to generate ℓ matrices. Express the function necessary to precomputation time (or the attack time) by a number of hash-reduction function application instead of real word time allows to estimate a cost that will be independent of the hardware used. In the case of a Hellman table, the precomputation time can thus be estimated by Equation (2.4).

$$P = \ell mt \tag{2.4}$$

In the literature, the precomputation cost or time for TMTO is typically not considered worth researching. The underlying assumption is that since this precomputation is conducted only once for all, thus, it can take as much time as necessary.

2.2.5. Memory Cost

> **Reminder:** The elements in the first column of a TMTO or Hellman matrix are called *start points* (SPs), those in last column are called *end points* (EPs). Only SPs and EPs of a matrix are stored to be used during the attack phase.

In the naive way of storing SPs and EPs, when using ℓ tables of m chains, ℓm pairs (SP, EP) need to be stored for the attack phase, i.e., $2\ell m$ elements.

Each element can be stored on $\log_2(N)$ bits. The memory cost of storing a Hellman table can thus be estimated by Equation (2.5).

$$M = 2\ell m \log_2(N) \tag{2.5}$$

[3] A loop (or cycle) is a sub-chain where the last element of the sub-chain leads to the first element of the sub-chain when the hash-reduction function is applied, resulting in a loop.

2.2.6. Attack Cost

> **Reminder:** The process of assuming that the searched element is in a given column c of the matrix and performing computations to verify this hypothesis is referred to as a *search* in column c.

The average attack time, denoted as T, is typically evaluated by the average number of hash operations required to either find the searched element (success of the attack) or to perform a search through all columns of all tables (the attack fails).

The average attack time T is therefore strongly correlated with the coverage of the tables used during the attack. If the coverage is high, the probability of finding the searched element without having to search in all columns of all tables is lower than when the coverage is low. Conversely, with low coverage, there are generally fewer tables to search through, and each table can have fewer columns to search.

When using a Hellman table that is not clean, numerous false alarms can occur, significantly slowing down the attack: with each false alarm, an additional t operations are required to deal with it. Consequently, the probability of a false alarm must also be computed, which adds complexity to the computation.

For simplicity, we provide in Theorem 2.1 from [AJO08], the cost of an attack for a clean Hellman table. In [Hon10], the authors provide the attack time for non-clean Hellman tables and propose an analysis of the false alarms cost.

Theorem 2.1. *Given N, m, ℓ and t, the average cryptanalysis time of clean Hellman tables is:*

$$T = t\sum_{k=1}^{\ell} k\frac{mt}{N}\left(1-\frac{mt}{N}\right)^{k-1}$$

2.3. Other TMTOs Variants

In Section 2.4, we discuss the reasons behind the decision to focus on the Rainbow tables variant in this thesis. However, before delving into that, we briefly introduce the major variants of TMTO, namely the Distinguished Points tables, Rainbow tables, and Fuzzy Rainbow tables variants.

2.3.1. Distinguished Points Tables

The Distinguished Points (DP) tables variant can be traced back to an idea proposed by Rivest, as referenced by Denning in [Den82]. While it was not initially introduced in a standalone publication, the variant received further attention in subsequent works, in particular in [SRQL02, HJK+08a, HJK+08b, HLM11]. Different way of analyzing the variant were provided in [BPV98, AJO08, HM13]. A variant of DP tables was also proposed in [HJK+08b].

DP table is a variant built upon the foundation of Hellman's tables. Instead of using chains of fixed length t, the chains are computed until a so-called *distinguished point* (usually, element beginning or finishing by at least d zeros) is reached. The way of choosing the DP is called the DP *property*. As chains end when a DP is reached, DP tables have chains of variable lengths. This unpredictability affects both the precomputation and attack phases.

2.3.1.1. Precomputation Phase

The generation of a DP matrix shares similarities with the process used in Hellman tables. However, instead of having a fixed length t for the chains, these stop upon reaching a distinguished point, DP. If no DP is reached after computing a pre-defined number ν of elements, the chain computation stops, and the chain is discarded, as it is considered that the chain will never end or is blocked in a loop.

At the beginning of the precomputation phase, the DP property, i.e., the distinguished points, are firstly selected. The values ν, ℓ, and m are also set, where m represents the number of chains to compute, ℓ the number of matrices, and ν the maximum length computed per chain.

Subsequently, SPs that are not distinguished points are chosen, and the precomputation begins in a similar manner to Hellman tables. For each SP, the hash-reduction function f (defined as in Equation (2.1)) is applied until a DP is reached or after having reached ν iterations of the function. Instead of obtaining a matrix with t columns, chains have different lengths and the average length of chains is denoted by $\widehat{t}$. When reaching a DP, the latter is stored as an EP in addition to the corresponding SP. All other elements of the chain are discarded. The final table thus is formed of the SPs and their corresponding DPs. The evaluation of the average chain length is usually difficult but in [LH16a], the authors provide a formula to evaluate it. In certain versions [BPV98], the chain length is stored along with the EP, potentially increasing the memory cost but also significantly accelerating the attack phase.

Figure 2.2 represents a DP matrix, and Figure 2.3 shows the corresponding DP table.

> **Reminder:** A *clean* matrix is a matrix in which only one instance of merged chains has been kept.

Clean DP Tables Cleaning the matrix when using DP tables is a much simpler task compared to Hellman tables. This is because merged chains in DP tables end with the same DP, hence only one chain among those ending with the same DP needs to be kept to form a clean DP matrix.

If DP tables are not cleaned, it is possible to end up with EPs that have the same DP for different chains. This can significantly reduce the efficiency of the attack phase. Therefore, DPs matrices are cleaned before being stored by retaining only one instance of the chains ending with the same DP (usually, the longest chain is kept).

As highlighted in [LH16b], clean DP tables do not require additional precomputation time and they significantly outperform their non-clean counterparts in terms of efficiency. Hence, unless specified otherwise, we will use the term DP tables to refer to clean DP tables.

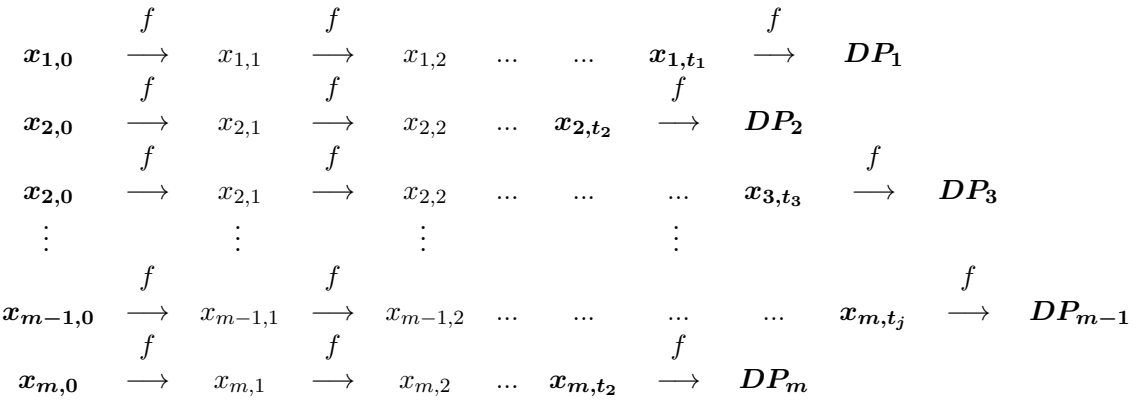


Figure 2.2. – DP Matrix

$\boldsymbol{x_{1,0}}$	$\boldsymbol{DP_1}$
$\boldsymbol{x_{2,0}}$	$\boldsymbol{DP_2}$
$\boldsymbol{x_{2,0}}$	$\boldsymbol{DP_3}$
⋮	⋮
$\boldsymbol{x_{m-1,0}}$	$\boldsymbol{DP_{m-1}}$
$\boldsymbol{x_{m,0}}$	$\boldsymbol{DP_m}$

Figure 2.3. – DP table

2.3.1.2. Attack Phase

In the case of DPs tables, the *attack chain* is the hypothetical chain that starts at the pre-image of y and ends not in column t but when a DP is reached or after ν operations.

Similarly to Hellman tables, the attacker initiates the attack phase by computing $R(y)$, followed by checking for a match with any of the stored EPs, i.e., checking if $R(y)$ is a DP and if yes, if this DP is among the EPs stored. If no match is found, the attacker applies the function f to $R(y)$, hence computing $f(R(y))$, and again checks if a match occurs with any of the EPs, i.e., if $f(R(y))$ is equal to one of the stored DPs. The attacker continues to apply the function f until a match is found between the attack chain and one of the stored DPs, or until ν applications of f are made. If ν applications of f are reached, this indicates that the preimage of y is not in the table, prompting the attacker to move on to the next table. In case the attack chain matches a DP at some point, the attacker reconstructs a chain starting from the corresponding SP until reaching the column where the searched element should be in case of success. In the latter column, the element is hashed and compared to y; if it is equals to y, the attack is successful, otherwise, it is a false alarm.

In case of a false alarm, the attacker immediately proceeds to the next table as, when using clean DP tables, a false alarm in a table leads to the conclusion that the searched element cannot be found in the given table.

2.3.1.3. Comparison with Hellman Tables

In this section, we discuss the advantages and disadvantages of DP tables. DP tables are generally considered superior to Hellman tables in all cases. However, to our knowledge, no studies have proposed a comparison of all variants for different coverages ranging from 1% to

99.99%. The focus of most papers is on the comparison for high coverages, typically higher than 90% coverage. Results for coverage around 50% to 70% are seldom mentioned [HM13, HJK+08b, KH13], and are not extensively discussed.

Advantages of DPs Tables

- **Handling Collisions**: A key advantage of using DP tables is their efficiency in building attack chains; only one attack chain needs to be computed per table, in contrast to Hellman tables, which require nearly one per column. Essentially, a single search in a DP table is enough to determine whether an element is in the table, unlike in Hellman tables where a search must be performed in every column. This significantly reduces the number of collisions encountered during the attack.

- **Memory Usage**: The use of DPs as EPs leads to more efficient memory usage and decreases computational overhead during the online phase as storing DPs that are usually elements that begin or end the same way, cost drastically less memory.

- **Loop Elimination**: DP tables prevent the occurrence of loops (or cycles) in chains, as they can be easily detected and discarded during the precomputation phase. Conversely, Hellman tables often contain a number of loops that can significantly slow down the attack phase and reduce the coverage for a given amount of memory used.

Drawbacks of DPs tables

- **Variability**: DP tables have a high variability in the lengths of chains, which can pose challenges. Some chains might end up being extraordinarily short if a DP is found early, while others may stretch considerably longer. This variability can lead to less predictable precomputation and attack phase, potentially impacting the overall efficiency of the table generation and search process.

  The unknown length of each chain can also slow down the attack phase. For instance, an attacker might have to perform ν applications of the hash-reduction function before switching to the next table. This issue is partially mitigated in [BPV98], where the authors propose storing chain lengths in addition to the DP, but this approach incurs a considerable increase in memory usage.

- **Complexity**: Predicting the average chain length poses a significant challenge when using DP tables. In Hellman's tables, chain lengths are pre-determined, which simplifies the estimation of coverage and precomputation time. On the contrary, the variability of chain length in DP tables, complicates the prediction of the average chain length, thereby affecting the overall coverage of the search space.

2.3.2. Rainbow Tables

Rainbow tables (RTs) were introduced by Oechslin in 2003 [Oec03]. Alongside the original paper, the author also presented *Ophcrack* [TO], a tool designed to use his tables. The primary difference of RTs, in comparison to DPs and Hellman tables, is the use of multiple reduction functions. A Rainbow matrix corresponds to a Hellman matrix, but where each column of the matrix uses a distinct function. The assignment of a color to each function

ultimately creates a "rainbow" following the precomputation, hence the name "Rainbow tables" (also illustrated in [BBS06]). In [Li16], the authors propose a variant of RTs in the form of a TMDTO used to attack A5/1, and the practical use cases of RTs are analyzed in [KHP13, ACKT17].

2.3.2.1. Precomputation phase

Similarly to Hellman and DPs tables, the precomputation phase of RT involves generating a matrix where only the first and last columns are kept for the attack phase.

Matrix Generation At the beginning of the precomputation phase, m_0 elements are selected as SPs. To minimize storage, these elements are typically chosen to be as small as possible and in sequential order. Subsequently, a first hash-reduction function f_1 is applied to each SP, followed by the application of a second hash-reduction function f_2, and so forth, until the hash-reduction function f_t is applied. The elements resulting from the application of the hash-reduction function f_t are the EPs, which are stored alongside their corresponding SPs to form the RT, while all other intermediate elements are discarded.

The generation of a RT requires the application of t hash-reduction functions per table, as opposed to a single function for both Hellman and Distinguished Point (DP) tables. When multiple tables are generated to increase coverage, each table employs a unique set of t hash-reduction functions that differ from those used in other tables. Therefore, if ℓ tables are generated, ℓt hash-reduction functions are used, instead of ℓ for Hellman and DP tables. To achieve a high coverage, only 2 to 5 tables are typically required when using RTs, compared to hundreds or thousands for Hellman and DP tables.

Clean Rainbow Table As for Hellman and DPs tables, collisions between chains occur during matrix computation. When two distinct chains merge in different columns, they remain distinct in subsequent columns due to the application of different reduction functions in each column. However, when two chains become equal in the same column, they merge and become identical in all subsequent columns and thus end with the same EP. A *merge* is formally defined in Definition 2.2.

Definition 2.2. *A merge in column j between two chains in rows i and i' with $i \neq i'$ occurs when $x_{i,j-1} \neq x_{i',j-1}$, but $f(x_{i,j}) = f(x_{i',j})$.*

As discussed in Section 2.5.4, equal EPs significantly slow down the attack phase. Consequently, *clean* RTs, which keep only one instance of merged chains at the end of the precomputation phase, are always used instead of non-clean RTs. Finding merged chains is straightforward since they end with the same EP. Thus, cleaning the matrix simply consists in keeping one instance of chains ending with the same EP.

> **Reminder:** m_0 is the number of elements selected as SPs at the beginning of the precomputation phase (in column 0).

When clean rainbow tables, m_t chains remain in the final clean matrix instead of m_0 chains. With typical parameters, $m_0 = 0.05 m_t$ (See Section 2.5 and Chapter 3 for more details). Figure 2.4 represents a non-clean Rainbow matrix, Figure 2.5 represents its corresponding clean Rainbow matrix. Henceforth, we will assume that all RTs are clean.

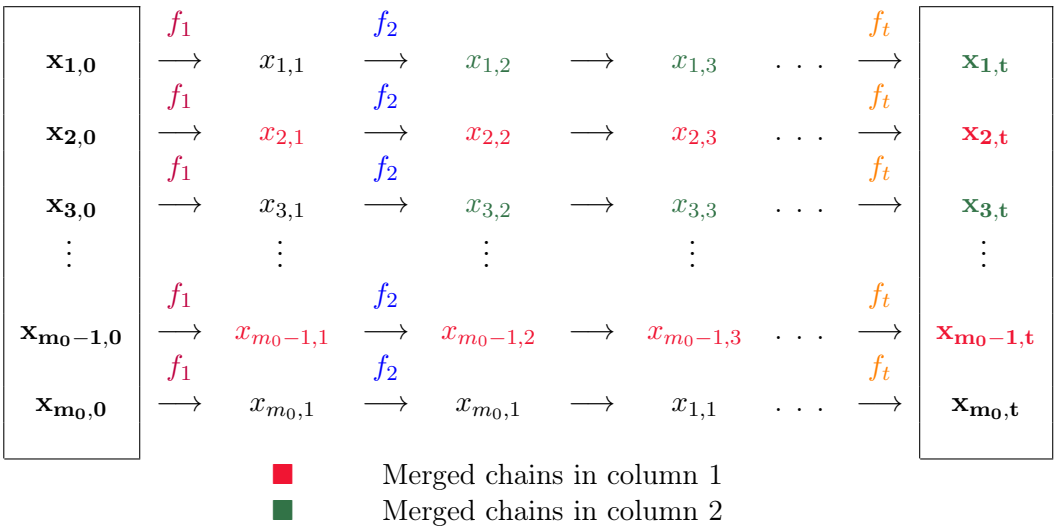


Figure 2.4. – Non-Clean Rainbow Matrix

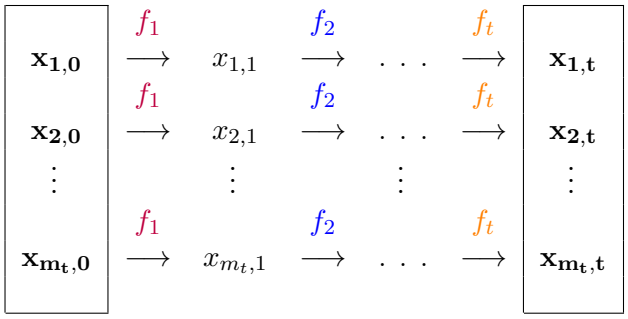


Figure 2.5. – Clean Rainbow Matrix

2.3.2.2. Attack Phase

During the attack phase, the attacker starts by hypothesizing that the searched element x is in the penultimate column of the rainbow matrix, and tests this assumption. If x is not in this column, the attacker proceeds to assume that x is in the second to last column and iteratively searches through all columns until x is found or until a search in all columns have been performed.

> **Reminder:** The term *attack chain* is used to denote the chain that hypothetically begins at the preimage of y, i.e., with the searched element x^*.

To perform a search in a column c, the attacker begins by computing the attack chain. For RTs, this chain initiates with $R_c(y)$ and ends in column t. Formally, the computation of the attack chain consists of computing Z such that $Z = f_t(f_{t-1}(...(f_{i+1}(R_c(y)))))$. After the computation of this chain, the attacker checks whether Z matches any of the EPs stored in the RT. If there is a match, the attacker then computes the chain starting from the corresponding SP and ending at the element $x_{i,c}$, where i represents the row of the matched EP. Following this, the attacker computes $h(x_{i,c})$. If $h(x_{i,c}) = y$, then the searched element is found and it is equal to $x_{i,c}$.

Even when using distinct reduction functions in each column, collisions can still occur within the same column, often leading to false alarms. Therefore, it is possible (and often the case) that a match between Z and an EP occurs while $h(x_{i,c}) \neq y$. In such case, the attacker continues the search in other columns to the left until either all columns have been searched, or the searched element x is found.

When using Hellman tables, the attacker must perform t searches to confirm that the searched element is not in the table, it costs t^2 operations for RTs. This increased cost is counterbalanced by the fact that RTs have significantly higher coverage per table and thus require far less tables than Hellman tables.

A noteworthy point when using multiple RTs is that it is faster to alternate searching through the columns of each table, rather than sequentially searching through each table. This is contrary to Hellman tables where the cost of search does not increase when searching further to the left.

2.3.2.3. Comparison with DPs and Hellman Tables

Several papers provide an extensive comparison between Hellman tables, DP tables, and RTs [HJK+08b, HM13]. We summarize the key points that we believe to be crucial below.

Advantages of RTs over DPs and Hellman Tables

- **Coverage**: RTs achieve higher coverage with fewer tables compared to DPs and Hellman tables. This results in reduced precomputation and attack time for same coverage. A non-formal yet intuitive argument to explain this is that for DPs and Hellman tables, $mt^2 \simeq N$, whereas when using RTs, $mt\ell \simeq N$, with ℓ being significantly lower than t.

- **Collision**: A significant advantage of RTs over Hellman and DPs tables is the reduced number of collisions when using RTs as opposed to DP tables. The fewer collisions allow for both the increase of coverage and the reduction of attack time.

- **Utilization**: Compared to DP tables, RTs provide more predictable outcomes. The computation of average DP chain lengths can be tricky, and the high variance can result in increased variance in attack phase time, and a more challenging precomputation phase.

Drawback of Rainbow Tables

- **Memory Access**: The number of memory accesses during the attack phase is higher with RTs than with DPs. This is because, when using RTs, a single memory access allows the checking of one column, whereas a single memory access for a DP table enables the checking of an entire table. Even when considering that more tables need to be searched when using DP tables, this usually does not offset the disadvantage of RTs number of accesses per table.

- **EPs Storage**: Given that EPs of DP tables are DPs themselves, they can be stored efficiently. In contrast, it is more challenging to store EPs of RTs efficiently. While solutions do exist [ABC15, AC13], they are typically less efficient than DPs, which by their very nature are easy to store.

2.3.3. Fuzzy Rainbow Tables

We now present the Fuzzy Rainbow tables, a variant initially proposed by Barkan, Biham, and Shamir in [BBS06]. This approach synthesizes concepts from both the RTs and the DP tables, resulting in a novel variant. The authors of [BBS06] assert that Fuzzy Rainbow tables are more efficient in comparison to the other two variants. It is worth noting that this technique is also presented in [vdBP13].

The precomputation phase begins by generating an initial DP matrix, followed by concatenating a second DP matrix to the first. The second DP matrix is created using a different reduction function. A third DP matrix is then generated from the second, and so on. The final product is a matrix that uses s distinct reduction functions, with each reduction function applied in a unique part of the matrix. Consequently, each chain contains s DP points, with each situated at the end of its respective sub-matrix. In this context, a *sub-matrix* refers to one of the s DP matrices within a Fuzzy Rainbow matrix.

For simplicity, we will henceforth refer to Fuzzy Rainbow tables as *Fuzzy tables* (FT). A clean version of FT, as with DP and Rainbow variants, has been demonstrated to be more efficient [KH13] than the standard (non-clean) version of FT. Unless otherwise specified, we will thus consider FTs to be clean in the context of this section.

2.3.3.1. Precomputation Phase

The generation of the Fuzzy matrix starts with the determination of the value s, which corresponds to the number of DP sub-matrices that will be generated, or in other words, the number of different reduction functions per matrix. The rule for selecting DPs, such as the number of bits set to 0 in each DP, is also fixed. The generation itself then proceeds as though generating s DPs matrices. Initially, SPs that are not DPs are selected, then m chains are generated from these m elements through the successive application of the hash-reduction function f_1 until each of them reaches a DP or reaches the maximum chain length ν. Once all chains have reached a DP or have exceeded a length of ν, the first sub-matrix is generated.

Prior to continuing with the precomputation, the first sub-matrix is cleaned by keeping only one instance of chains ending with the same DP (and keeping the chain with the maximum length), and by discarding chains that have reached ν iterations of the hash-reduction function.

Next, the second sub-matrix is computed, starting with the DPs elements of the first sub-matrix and applying the hash-reduction function f_2 instead of f_1, i.e., using a different reduction function than in the first sub-matrix. The generation proceeds in the same way until s sub-matrices have been generated, with each of them having been cleaned.

The EP of each chain is the last DP reached, i.e., the DP of the chains with hash-reduction function f_s. As for RT and DP tables, ℓ tables can be generated in order to cover a larger portion of A. In each table, all reduction functions are different from those of the other tables. Figure 2.6 represents a Fuzzy matrix and Figure 2.7 represents its corresponding FT.

It is worth noting that compared to DP matrix, the length of Fuzzy matrix chains are, in average more homogeneous, which simplified the computation of the theoretical coverage.

An other interesting point is that by taking $\nu = 1$, the matrix obtained is a Rainbow matrix (but with a very little height as only few chains ending by a DP in each column will remain), and by taking $s = 1$ the matrix obtained is a DP matrix.

In [KH14], the authors state that a typical good number of sub-matrix to use should be between $s = 30$ and $s = 150$ depending of the targeted coverage and trade-off between attack

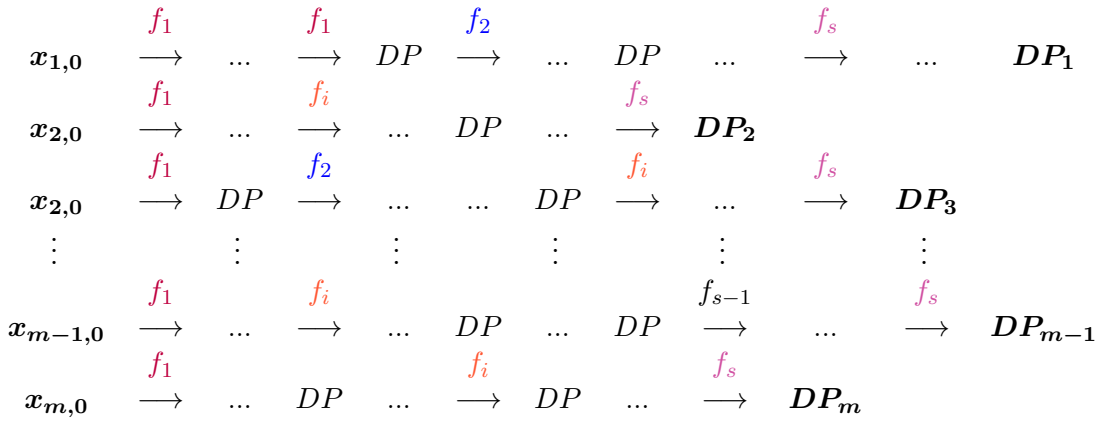


Figure 2.6. – Fuzzy Matrix

$x_{1,0}$	DP_1
$x_{2,0}$	DP_2
$x_{2,0}$	DP_3
⋮	⋮
$x_{m-1,0}$	DP_{m-1}
$x_{m,0}$	DP_m

Figure 2.7. – Fuzzy Rainbow table

time and precomputation time. The authors also present the number of tables to use in the order of $\ell = 50$ but this number of tables vary according to the targeted coverage. The authors of [KH14] also observe that when using FT instead of DPs tables, the lengths of the DP chains in the Fuzzy variant tend to be shorter than those in the DPs tables. Moreover, these lengths seem to follow a normal distribution, as opposed to a geometric one when using DPs tables.

2.3.3.2. Attack Phase

> **Reminder:** A *sub-matrix* refers to one of the s DP matrices within a Fuzzy Rainbow matrix.

The attacker first assumes that the searched element is in the s-th sub-matrix, using the hash-reduction function f_s, i.e., the sub-matrix to the right of the matrix. The attacker then computes the attack chain starting from $R_s(y)$, and subsequently applies f_s to $R_s(y)$ until a DP is found or until f_s has been applied ν times.

If a DP is reached and if a match occurs between this DP and one of the EPs, the attacker reconstructs the corresponding chain, starting from its corresponding SP. Each time a DP is reached, the attacker knows that they must switch to a different hash-reduction function.

After reaching the $(s-1)$-th DP, the attacker computes only the necessary amount of hash-reduction functions until the column where the searched element should be if it is equal to the pre-image of y. The attacker then hashes the element in the latter column and compares it to y. If they match, the attack is successful; if not, it is a false alarm.

As with the DP table variant, a false alarm in a column means that the searched element is not in the entire sub-matrix (but it can still be in the other sub-matrices of the Fuzzy matrix).

Thus, in the case of a false alarm or if ν applications of the hash-reduction function have been executed without finding a DP, the attacker moves to the next sub-matrix to the left.

The attacker starts again, but this time by applying R_{s-1} to y. Then the attacker applies f_{s-1} to $R_{s-1}(y)$ until either a DP is reached or ν applications have been performed. If a DP is reached, the attacker understands it to be the DP of the $(s-1)$-th sub-matrix, and thus they apply the hash-reduction function f_s to the attack chains until reaching a new DP or having performed ν iterations of the hash-reduction function. If a DP is reached, the attacker searches for a match with one of the EPs. If a match is found, the attacker processes as for the sub-matrix s and rebuilds the chain until reaching the $(s-2)$-th DP.

When the attacker reaches the $(s-2)$-th DP, they apply only the necessary number of hash-reduction functions until the column of the supposed searched element. They then hash the element and compare it to y; if it matches, the attack is successful, otherwise, it continues in the same way until having searched in every sub-matrix.

It is worth noting that when ℓ tables are used, as with RT, the search is not conducted table by table but sub-matrix by sub-matrix. The attacker will thus initially search in the sub-matrix s of table 1, then in sub-matrix s of table 2 and so on. Once the attacker has searched in all s sub-matrices, they begin to search in sub-matrices $s-1$.

2.3.3.3. Comparison with other variants

Although there is consensus on the advantages of FTs over Hellman tables, no such consensus exists between DPs, Fuzzy, and RTs. Moreover, while extensive studies and proposals for improvements have been made for DP tables and RTs, literature about FTs and their improvements is less abundant.

Nonetheless, we provide below some advantages and disadvantages associated with the use of FTs.

Advantages of Fuzzy Tables

- **Improved Coverage**: Compared to Hellman and DP tables, FTs generally offer higher coverage, without the need to increase the number of stored chains.

- **Reduced False Alarms**: Compared to Hellman and DP tables, FTs significantly lower the frequency of false alarms during the attack phase. This advantage comes from the ability of fuzzy chains to better handle collisions that lead to merges, thereby reducing the probability of false alarms.

- **Memory Usage**: For the same coverage, and given comparable precomputation and attack times, FTs require less memory than Hellman and DP tables. Although dealing with false alarms takes more time, as there are fewer tables to generate and use during the attack, it often results in a quicker attack phase than DP and Hellman tables. Some authors claim that in some cases, it may even outperform RTs [BBS06, KH14].

- **Memory Access**: Compared to Hellman tables and RTs, FTs require a significantly fewer number of memory accesses during the attack phase. In some environments, such as on a GPU or FPGA, the cost of memory access during the attack phase cannot be ignored.

- **Precomputation Time**: The precomputation time of FT is easier to predict than the precomputation time of DPs tables. This is due to the fact that DP sub-matrices "compensate" each other and the resulting chains length are much more homogeneous than DP tables.

Drawbacks of Fuzzy Tables

- **False Alarm**: While FTs have less false alarm than DPs and Hellman tables they have more of them than RTs. This can result in additional time and computational resources, which can be particularly problematic in larger search spaces.

  The cost to deal with false alarms is higher when using FTs than DPs tables. As every chain of every sub-matrix has a different length, the attacker can not know until reaching ν applications of a hash-reduction function or until reaching the supposed EP of the attack chain if the attack chain will end with a false alarm or not.

  Compared to DP,s this slows down the attack phase, but in the same time the coverage is much higher than with DPs tables and thus, less tables have to be generated and searched in through.

  Compared to RTs, discussions are still ongoing, as some authors claim that fuzzy attacks are faster than RTs[KH13], but do not give the parameters used for RTs to make the comparison.

- **Coverage**: The coverage of FTs is typically less than that of standard RTs, especially for larger search spaces. The extra false alarms and the increased complexity can lead to a less efficient overall attack phase.

- **Optimization Challenges**: The variable nature of the chains makes optimizing FTs more difficult. Determining the optimal balance between fuzziness and chain length can be a complex task requiring careful consideration and potentially sophisticated computations.

- **Complexity**: FTs introduce a new level of complexity into the table construction and search process. This additional complexity can lead to increased computational costs and make the overall process harder to comprehend and execute effectively. Moreover, the literature concerning FTs is less extensive compared to the materials available on DP, Hellman tables, or RTs.

2.4. Variant Analyzed in the Thesis

It is important to note that apart from the Hellman tables, which are evidently inferior to other variants at hight coverage, there is no general consensus on the superior variant among FTs, DP tables, and RTs. Therefore, the study of any of these three variants is justifiable.

This thesis focuses exclusively on the RTs variant. Part of this decision is arbitrary as we had an initial intuition for improving this specific variant at the beginning of the thesis. With numerous aspects of TMTOs needing further research, a choice had to be made at some point. However, since there are also objective reasons that led us to analyze this specific variant given our objectives. These reasons are listed below.

- Several papers [HM13, LH16b] argue that the RT variant is superior to the DP variant. As far as we know, the primary argument against this claim is that for equal coverage and memory usage, the RTs variant requires significantly more memory access than DP tables. In [Wie04], the authors contend that the number of memory accesses is one of the most crucial, if not the most crucial, criteria to consider when performing a cryptanalysis attack. However, we believe that this argument can be mitigated. Paper [ACKT17] demonstrates that in environments where the memory access cost is high (such as Hard disk or SSD), the most significant cost factor of the attack is the number of hashes to perform. We conjecture that, even if GPUs or FPGAs were employed, the attack cost would still predominantly depend on the number of hashes performed. There is currently no consensus on this point, and as will be discussed in Chapter 8, future works should aim to reach a consensus on this topic.

- The main objective of this thesis, particularly in its initial stages, was to emphasize the importance of the precomputation phase of TMTO, instead of focusing exclusively on the attack phase, as is commonly done in the literature. Given this, improving the RTs precomputation phase seems more challenging than other methods due to the use of a different reduction function in each column, especially when considering the distribution of the precomputation.

- Throughout this thesis, the aim was to work on high-coverage TMTOs. The RTs variant is particularly suitable for this, as it performs very well with high coverage and has been, in part, designed to cover a large portion of the search space.

- In [KH13], authors argue that the Fuzzy Rainbow tables variant is superior to the RTs variant in all cases. The same authors argue in [HM13] that the RTs variant is superior to the DP tables variant in all cases. From these papers alone, we might conclude that the Fuzzy Rainbow tables variant is the best and thus should be analyzed. However, in [KH13], the author dit not compare variants for all coverage and did not provide the RTs parameters used for comparison, only those of the Fuzzy Rainbow table. In particular, if maximal RTs (see Section 2.5) were used in the study, this could have significantly disadvantaged the RTs variant.

2.5. Vanilla Rainbow Tables Analysis

In this section, we delve into the details of the Vanilla RT variant. In Section 2.5.1, we recall the method for generating the rainbow matrix using more formal terms and expressions, and we introduce the crucial notion of maximality. We then provide formulas to estimate the precomputation time for ℓ RTs. In Section 2.5.2, we offer formulas to evaluate the coverage of RTs, while in Section 2.5.3, we discuss the approach to estimate the memory requirements for storing RTs. Finally, in Section 2.5.4, we present the attack time of RT in terms of the number of hash operations to be performed.

2.5.1. Precomputation phase

2.5.1.1. Matrix Generation

> **Reminder:** When using *clean* RTs, only one instance of merged chains is kept at the end of the precomputation phase.

During the precomputation phase, a clean matrix with $t+1$ columns and m_t rows is computed using elements from the search space A. The matrix elements are denoted as $x_{i,j}$ with $0 \leqslant i \leqslant t$ representing the column and $1 \leqslant j \leqslant m_t$ representing the row. Two types of functions are used to construct the matrix: reduction functions R_i and the hash function h.

Reduction functions R_i with $0 < i \leqslant t$ are fast, mapping elements from the hash space B to A with nearly uniform distribution. In contrast, the function h with $h : A \to B$ is considered slow and is the function that the attacker want to target. The matrix element $x_{i+1,j}$ is obtained from $x_{i,j}$ (element in the same row, previous column) with $x_{i+1,j} = R_{i-1}(h(x_{i,j}))$.

A *chain* depicts the list of elements of the same row with successive columns. Functions f_i with $f_i : A \to A$ are the composition functions of R_i and h such as $x_{i+1,j} = f_{i-1}(x_{i,j})$ and are called *hash-reduction* functions. Elements in the first column are chosen arbitrarily but must be different.

2.5.1.2. Maximality

> **Reminder:** The generation of a RT begins by computing m_0 chains. With typical parameters, a large portion of the m_0 chains are merged and are thus discarded to keep only m_t chains to form a clean RT.

When using clean RTs, the number of elements in the last column of the cleaned matrix is less than the number of elements with which the precomputation begins. The number of elements considered at the start of the precomputation phase is denoted by m_0, while the number of elements in the last column after cleaning is denoted by m_t.

It is possible to determine the number of chains with distinct elements in a column i. This chains are called the *surviving chains*. In a column i, there is m_i surviving chains i.e., m_i chains with distinct elements in column i.

As presented in [AJO08], given that m_i surviving chains are in column i, and that the probability of picking an element among the distinct elements of surviving chains in column i, in column $i+1$ is $\frac{1}{N}$ and that there are N possible elements to pick, m_{i+1} can be defined recursively from m_i as follows:

$$m_{i+1} = N\left(1 - \left(1 - \frac{1}{N}\right)^{m_i}\right) \tag{2.6}$$

Equation (2.7) gives the solution of Equation (2.6) and allows us to give a good approximation of the number of surviving chains remaining in column i.

$$m_i = \frac{2N}{i+\gamma}, \quad \text{with} \quad \gamma = \frac{2N}{m_0}. \tag{2.7}$$

To achieve the highest success probability, one can choose $m_0 = N$ elements at the beginning of the precomputation phase. In this case, a maximum of $m_t^{\max}$ elements will remain at the end of the phase, where $m_t^{\max}$ is given by equation (2.8) borrowed from [Oec03].

$$m_t^{\max} = \frac{2N}{t+2}. \tag{2.8}$$

A table generated using $m_0 = N$ elements is known as a *maximal* table. However, selecting $m_0 = N$ elements is impractical as it drastically increases computation time. Thus in general less than N elements are considered at the beginning of the precomputation phase.

2.5.1.3. Precomputation Time

Given m_0 SPs considered at the beginning of the matrix generation, with $m_0 \leqslant N$, given $t+1$ columns in per matrix and ℓ matrix to generate, the number of hash operations to perform in order to generate a clean RT is given by Equation (2.9).

$$P^{RT} = \ell m_0 t. \tag{2.9}$$

2.5.2. Success Probability

> **Reminder:**
>
> - Cleaning is the process of keeping one instance of each merged chain and discarding the others.
> - The number of elements in a clean RT is m_t, which is the number of remaining elements after cleaning in the last column t.

In a RT attack, the success probability of the attack phase depends on the coverage of A per RT, which depends on the number of different elements in the clean matrix. The success probability of a single RT is provided in [Oec03] and is given by Equation (2.10).

$$p = 1 - \left(1 - \frac{m_t}{N}\right)^t. \tag{2.10}$$

As mentioned in Section 2.3.2.1, the use of several tables increases the success probability beyond the limit of 86% [Oec03] for a single table. Specifically, when ℓ tables are used, the success probability of the attack is given by Equation (2.11), which is also taken from [Oec03].

$$p_\ell = 1 - (1-p)^\ell. \tag{2.11}$$

2.5.3. Memory Used

The naive way of storing RTs results in a memory that is expressed in the same way as in the case of Hellman tables, i.e., as it is presented in Equation (2.12).

$$M_{naive}^{RT} = 2\ell m_t \log_2(N) \tag{2.12}$$

In their paper [AC13], the authors introduce a method called *compress delta encoding* for storing RTs. This method achieves a memory usage very close to the theoretical lower bound, with a difference of only approximately 0.66% from the lower bound. For simplicity, we approximate the memory used to store a single RT, denoted as M^{RT}, using the lower bound proposed in [AC13]. This lower bound is provided in Equation (2.13).

Since only the SP and EP are used for the attack, only the memory required to store the SP (M_{sp}^{RT}) and the memory required to store the EP (M_{ep}^{RT}) needs to be considered. The total memory used to store a RT, M^{RT}, is the sum of M_{sp}^{RT} and M_{ep}^{RT}.

$$\begin{aligned} M^{RT} &= M_{sp}^{RT} + M_{ep}^{RT} \\ &= \ell \left[m_t \lceil \log_2(m_0) \rceil + \log_2 \binom{N}{m_t} \right]. \end{aligned} \tag{2.13}$$

2.5.4. Attack Time

> **Reminder:** The process of assuming that the searched element is in a certain column c of the matrix and performing computations to verify this hypothesis is referred to as a *search* in column c.

The average number of hash operations required to search through a single column of a RT is given by Proposition 2.2 taken from [AJO08].

Proposition 2.2. *For a given column c, the average number of hash operations C_c needed to perform a search is given by:*

$$C_c = t - c \prod_{i=c}^{t} \left(1 - \frac{m_i}{N}\right).$$

Given the cost of searching through a single column, the average total time required to perform an attack using ℓ tables is given by Theorem 2.3. This theorem is introduced and proven in [AJO08]. Intuitively, it corresponds to the sum of the cost of a search in column i weighted by the probability that the search stops there, plus the cost of the fail case.

Theorem 2.3. *Given a search space of size N, the average number of hash operations T required to perform an attack using ℓ RTs with $t+1$ columns, is:*

$$T = \ell \sum_{c=1}^{t} \left(\frac{m_t}{N} \left(1 - \frac{m_t}{N}\right)^{\ell(c-1)} \sum_{j=1}^{c} C_{t-j+1} \right) + \ell \left(1 - \frac{m_t}{N}\right)^{t} \sum_{c=1}^{t} C_c.$$

2.6. Most Relevant Variants and Improvements

In the following chapters, we will present improvements and variants introduced in our work. Some variants and improvements are compared to the most known and efficient variants and improvements of RTs. We thus briefly present the most relevant ones in this section.

2.6.1. Checkpoints

The checkpoint technique introduced in [AJO05] is an additional improvement in the use of RTs, aimed at reducing the computational cost of lookups. Since their introduction, checkpoints have been analyzed and improved, e.g., in [WL13, Hon16, KSH+15, KSH+12].

The crux of the checkpoint technique involves periodically storing intermediate information within each chain in the matrix, and not just the SPs and EPs. These stored information, referred to as "checkpoints", allow for a more efficient lookup process.

During the attack phase, when the attack chain matches an EP of the table, the checkpoints of the built attack chain are compared to the checkpoints of the matching chain. If at least one checkpoint differs, then the match is a false alarm and it is therefore useless to rebuild the chain of the table from its starting point.

The frequency of checkpoints in a chain is a trade-off between the extra memory needed for storing the table and the gain provided in the attack phase. More frequent checkpoints can result in faster lookups but at the expense of increased storage requirements. This trade-off needs to be carefully balanced to ensure optimal performance.

2.6.2. Fingerprints

Fingerprints technique has been introduced in [ABC15]. It consists in applying in an efficient way the checkpoints improvement with a memory improvement called *truncated end points*.

The principle of truncated end points method is to store EPs that have been truncated to reduce the memory needed to store the table. As the memory used to store the table decreases, for a given memory and given coverage, shorter chains can be used, thus allowing to reduce the attack time. The drawback of the truncated end points alone is that it introduces false positive match between the attack chain and the EPs, but the gain in memory counterparts this drawback.

Fingerprints thus allow to deal with the extra cost of memory necessary to store checkpoints by using the truncated method to counterbalance it. The resulting technique allows a significant speedup compared to vanilla RTs.

2.6.3. Delta Encoding

Delta Encoding method has been introduced in [AC13] and is a technique allowing the optimization of tables storage.

Traditionally, one of the primary challenge with RTs has been the storage of the tables. Without delta encoding methods, the size of the RTs would make them unfeasible for practical use due to the excessive storage needs. Delta encoding was introduced to address this issue. When using Delta encoding method, the memory taken by the final tables is less than 1% higher that the theoretical memory lower bound (Equation (2.13)).

The principle of delta encoding lies in the observation that a set of consecutive EPs, when sorted, presents a high degree of similarity. Rather than storing every EPs explicitly, the delta (difference) between successive EPs is stored instead.

In simpler terms, instead of storing each EP, delta encoding records the difference between the current EP and the previous EP. Because sorted EPs tend to be very similar, this difference requires less storage than the list of EPs. As a result, using delta encoding can dramatically reduce the size of the RT, making it more practical for real-world usage.

To decode the delta-encoded values when looking up a value in the EPs list, the attacker starts at the first stored EP, then adds the first delta to obtain the next EP, and continues this process through the table. Even though this decoding step adds a very small computational overhead during the table look-up process (when searching for a match between the attack chain and an EP), the trade-off for a much smaller storage requirement outweighs this disadvantage. In [AC13] the authors even consider this little extra cost as not significant.

We can imagine that the degree of compression achieved using delta encoding depends on the specific properties of the parameters used, in particular if the final RT comes from a very

sparce Rainbow matrix. However, in practical applications, delta encoding has proven to be a highly effective strategy for managing RT sizes.

2.6.4. Heterogeneous Tables

In [AC17], the authors introduce the Heterogeneous tables variant. When using heterogeneous tables, matrices of different lengths are generated, in contrast to generating matrices of the same shape in the vanilla RT. Figure 2.8 taken from [AC17] clearly illustrates the process. Instead of using four matrices of identical shape, four matrices of varying shapes are used, ranging from the tightest to the largest. The quantity of chains to store, the precomputation time, and the coverage all remain the same. However, the search within the tighter matrix is much less expensive than in one of the other matrices. With, in this case, 85% coverage per table, the majority of values can be found in it.

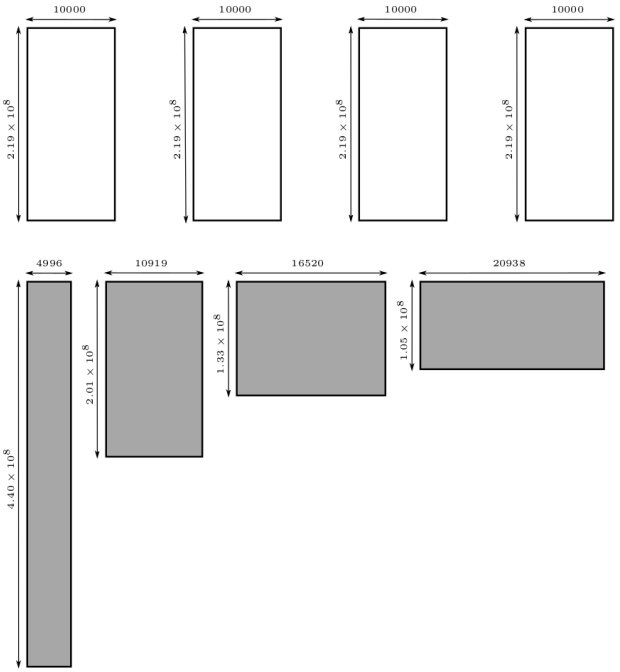


Figure 2.8. – Taken from [AC17], this figure represents four vanilla Rainbow matrices above with one possible set of four corresponding heterogeneous matrices below.

The particularity of heterogeneous tables is to not search uniformly in every matrix, as is the case with the vanilla version (where the search is initially conducted in the last column of the first matrix, then the last column of the second, and so on). Instead, a metric is used to choose in which column of which matrix to search. This metric is basically the probability of finding the element in a particular column of the matrix[4], divided by the cost of the search in this column. The search in the tightest matrix is, thus, generally favored. This variant reduces the average attack time by up to 40% (with the gain being greater when more tables are used). However, the worst-case scenario (searching in every table without success) is

[4]The probability varies depending on the columns of the matrix in which a search has already been performed without success.

worse than in the vanilla RTs.

3 Preliminary Results

A Clean RT (CRT) takes a coffee at her Hellman Table (HT) friend's place.

- HT: "Would you like a double espresso?"

- CRT: "Oh, no sorry, I only drink filtered coffee."

3.1. Motivations

Before delving into the major contributions of the thesis, it is essential to present some preliminary results.

Although certain results have been known and occasionally utilized for several years, they have not been formally documented. Similarly, some techniques have been employed for a while but have not been previously introduced. Finally, a few results obtained at the beginning of the thesis may be incremental or minor, but they nonetheless warrant introduction, as they are utilized throughout the thesis.

This chapter thus presents techniques and results that were introduced at the onset of the PhD in the paper [ACLA21], which are employed in all subsequent chapters. Specifically, this chapter introduces the notion of *maximality*, the method for selecting the parameter for the precomputation phase to achieve a specified probability of success, the *filtration* technique, and the precomputation time lower bound $P_{\min}$.

Takeaway:

In this chapter, we introduce key notions and techniques that are employed throughout the thesis's contributions. Specifically, we present:

- the maximality factor α, which characterizes the extent to which a table deviates from a maximal table.
- The filtration method, which involves cleaning the table during the precomputation process rather than only in the final column.
- The precomputation lower bound $P_{\min}$, which is attained when the matrix is cleaned in all columns during the precomputation phase.

3.2. Maximality Quantification

Reminder:

- A RT without duplicated EPs is called a *clean* RT.
- A clean RT is said to be *maximal* when it has been generated from $m_0 = N$ SPs. A maximal table contains in average $m_t^{\max}$ EPs.

As discussed in Chapter 2, using maximal tables is infeasible when addressing real-life problems. Consequently, non-maximal tables are generated using m_0 SPs, where $m_0 \ll N$. In the body of literature, the technique for generating non-maximal tables consists in arbitrarily picking m_0 values, computing the corresponding chains until obtaining the expected m_t number of EPs. This can turn out to be a lengthy and tedious process.

To address this issue, we introduce α in [ACLA21], called the *maximality factor*, and generalize the problem as follows: Given a target m_t such that $m_t \leqslant m_t^{\max}$, define α and r with $0 < \alpha \leqslant 1$ and $r > 1$ such that $m_t = \alpha m_t^{\max}$ and $m_0 = r m_t^{\max}$. In Lemma 3.1 and Proposition 3.2, we provide formulas to obtain r from α.

Lemma 3.1. *Let $m_0 = r m_t^{\max}$. The expected number of distinct EPs is given by:*

$$m_t \approx \frac{1}{(1+\frac{1}{r})} m_t^{\max}.$$

Proof. From Equation (2.7), we have $m_0 = \frac{2N}{\gamma}$. Given that $m_0 = r\, m_t^{\max}$, we can write $\gamma = \frac{2N}{r\, m_t^{\max}}$. Using Theorem 2.8, we obtain $\gamma \approx \frac{t+2}{r} \approx \frac{t}{r}$. Replacing γ in Equation (2.7) for $i = t$, we finally have:

$$m_t \approx \frac{2N}{t(1+\frac{1}{r})} = \frac{1}{(1+\frac{1}{r})} m_t^{\max}.$$

□

Proposition 3.2. *During the precomputation of a table, to obtain m_t chains such that $m_t = \alpha\, m_t^{\max}$, we must have $m_0 = r m_t^{\max}$, with:*

$$r \approx \frac{\alpha}{1-\alpha}.$$

Proof.

From Lemma 3.1:	$m_t \approx \frac{1}{(1+\frac{1}{r})} m_t^{\max}$.
Since $m_t = \alpha\, m_t^{\max}$:	$\alpha \approx \frac{1}{(1+\frac{1}{r})}$
Which conducts to:	$\frac{1}{r} \approx \frac{1}{\alpha} - 1 \Leftrightarrow r \approx \frac{\alpha}{1-\alpha}$

□

Instead of searching for the appropriate m_0 manually, this approach allows determining the targeted number of chains m_0 to compute in order to generate a clean table with m_t chains and a maximality factor α using Equation (3.1), which is obtained from Lemma 3.1 and Proposition 3.2.

$$m_0 = \frac{m_t}{1-\alpha}. \tag{3.1}$$

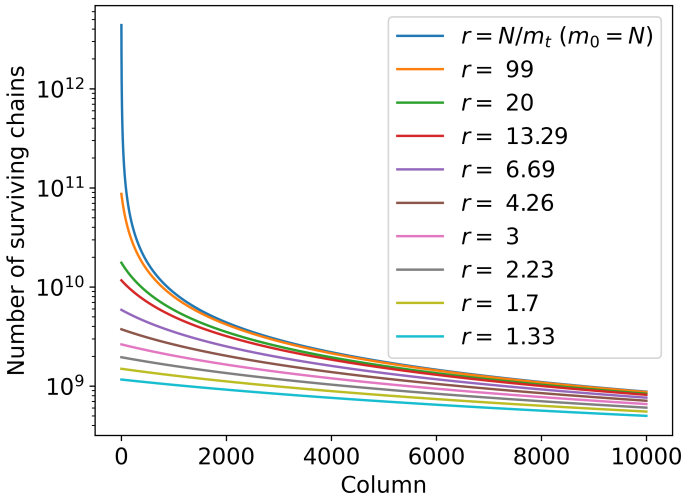


Figure 3.1. – Number of surviving chains m_i remaining in column i, according to the value r when $m_0 = rm_t^{\max}$ and for $N = 2^{42}$ and $t = 10\,000$.

Figure 3.1 illustrates the difference between the maximal table scenarios ($m_0 = N$) and the non-maximal table scenarios ($m_0 = rm_t^{\max}$) in terms of the maximum number of elements in each column. The case $m_0 = N$, equivalent to $r = \frac{t+2}{2}$, is represented in dark blue, while the other curves correspond to cases where $r < \frac{t+2}{2}$.

To illustrate Proposition 3.2, let us consider the case $r = 20$ (dark green curve), which results in $m_t \approx 0.95239 m_t^{\max}$. This is a reasonable number that provides a high-quality table[1] (m_t relatively close to $m_t^{\max}$), while significantly reducing the precomputation cost (compared to the $m_0 = N$ case). For example, with $N = 2^{42}$ and $t = 1\,000$, generating $20m_t^{\max}$ chains instead of N brings the number of chains to be generated from 440×10^{10} to 1.7×10^{10} – a 258-fold reduction – while keeping the number of EPs very close to the maximum. In real life cases, r is typically chosen between 4 and 30.

3.3. Filtration Method

Although, the filtration method was probably used in practical tools for a while, it has, up to our knowledge, never been published before [ACLA21]. The fundamentals of the filtration method are described in this section, while Chapter 4 introduces techniques to optimize and distribute the filtration.

3.3.1. Intermediate Filtration

Classically, generating a RT requires $m_0 \times (t+1)$ elements to compute, although only $m_t \times (t+1)$ elements are represented in the final (clean) matrix. Discarding merged chains at the end of the precomputation is a wasted effort because a single chain is kept among multiple chains with the same EP. So each chain is computed from its SP to its EP even if it merges with

[1] This case is typically used to generate *quasi-maximal* tables. These tables are very close to the maximal case while enabling a drastically lower m_0 than $m_0 = N$.

another chain before reaching the EP. An option to mitigate this waste is to remove merged chains progressively.

Thus, instead of computing the full chains in a row, from the SPs to the EPs, chains are divided into *sub-chains*, and merging chains are detected and discarded at the end of each sub-chain. A *sub-chain* is delimited by Intermediate Points (IPs). Computation of chains is thus performed "column by column" (or group of columns after group of columns), as opposed to the typical "chain-by-chain" method. A *filter* is placed in selected columns: when all sub-chains have been computed up to the filter, a *filtration* is performed: only one of the merged chains is saved. Figure 3.2 presents the mechanism of filtration with an example zooming on one filter placed in column 3.

In column 0, all elements are distinct. Subsequently, in column 1, five chains (illustrated in blue) merge. The precomputation proceeds to column 2, where two additional chains merge (depicted in purple). Upon reaching column 3, a filtration is performed, retaining only one representative of the merged chains and discarding the others. The generation continues without computing the discarded chains.

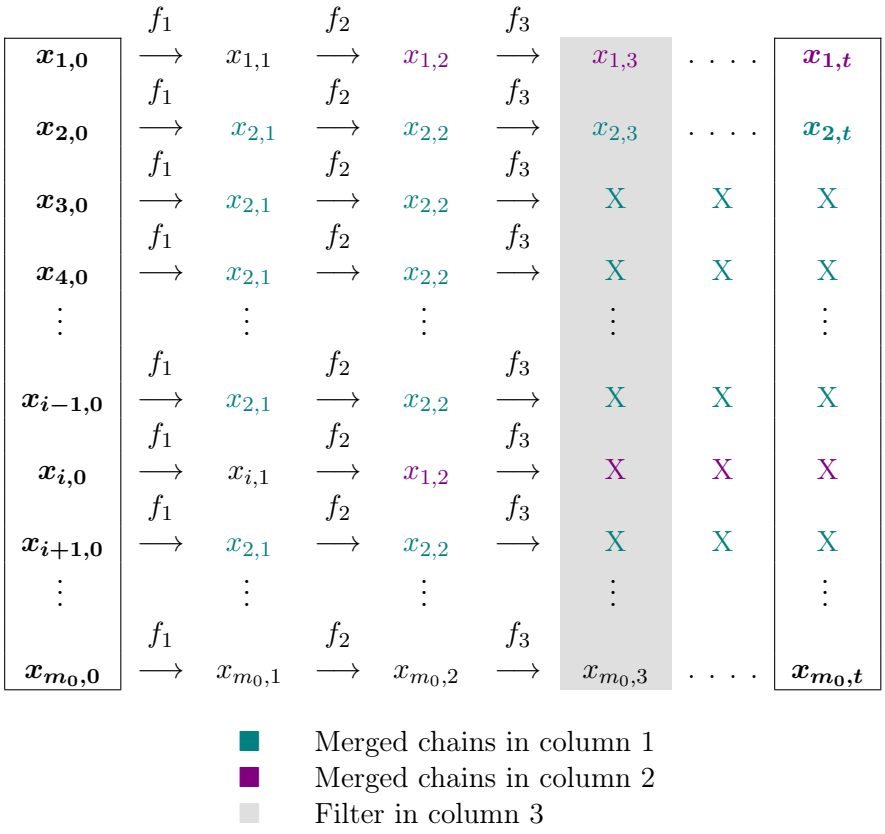


Figure 3.2. – Filtration with one filter in column 3.

3.3.2. Filtration in Each Column

Reminder: *Surviving chains* in a given column c, are chains that remain after a filtration in this column c. In a given column c, there is m_c surviving chains.

The minimum precomputation time needed to generate a RT corresponds to performing a filtration in all columns of the table. In other words, this corresponds to discarding duplicated chains as soon as the merge occurs with other chains. We provide below, the minimum precomputation time $P_{\min}$, needed to generate a clean RT. Proposition 3.3 first provide a lower bound on the precomputation time, an Theorem 3.4 quantifies this with results from Section 3.2.

Proposition 3.3. *Let m_i denote the number of surviving chains in column i of a rainbow matrix. The number P of hash operations to precompute a $m_t \times (t+1)$ clean rainbow matrix is lower bounded by:*

$$P \geq \sum_{i=0}^{t-1} m_i.$$

Proof. Given that the minimum hash operations to compute in order to obtain a table is when duplicates are removed from each column and that m_i denotes the number of surviving chains in column i, the expression of the lower bound is trivial. □

> **Reminder:** γ is a variable introduced in Equation (2.7) of Chapter 2, its only purpose is to simplify calculations with $\gamma = \frac{2N}{m_0}$.

Theorem 3.4. *Given $m_0 = rm_t^{\max}$ the number of SPs, $t+1$ the number of columns, and $r \ll t$, the naïve precomputation cost is:*

$$P_{naive} = m_0 t \approx 2rN, \tag{3.2}$$

and the minimum precomputation cost is:

$$P_{\min} = \sum_{i=0}^{t-1} m_i \approx 2N \ln(1+r). \tag{3.3}$$

Proof. The proof of Eq. (3.2) follows directly from $m_0 = \frac{2N}{\gamma} \approx \frac{2rN}{t}$. For Eq. (3.3), we have:

$$\begin{aligned}
\sum_{i=0}^{t-1} m_i &= 2N \sum_{i=0}^{t-1} \frac{1}{i+\gamma} = 2N \sum_{i=\gamma}^{t+\gamma-1} \frac{1}{i} = 2N \left[\sum_{i=1}^{t+\gamma-1} \frac{1}{i} - \sum_{i=1}^{\gamma-1} \frac{1}{i} \right] \\
&= 2N \left[H_{t+\gamma-1} - H_{\gamma-1} \right] \approx 2N \left[\ln(t+\gamma-1) - \ln(\gamma-1) \right] \\
&= 2N \ln\left(\frac{t+\gamma-1}{\gamma-1} \right)
\end{aligned}$$

with $H_n = \sum_{k=1}^{n} \frac{1}{k}$ the n-th harmonic number, and where γ is as defined in Equation (2.7), we obtain $\gamma \approx \frac{t}{r}$, and given that $r \ll t$, the expected result is obtained. □

For values of r such that $m_0 \ll N$ (i.e., "reasonable" values), we can make the approximation that γ is large (leading itself to the asymptotic approximation of H_n). This allows for expressing $P_{\min}$ that only depends on N and r and is, in particular, virtually independent of t. For $m_0 = N$ however, these approximations do not hold, and the resulting expression of $P_{\min}$ does depend on t (Corollary 3.5). We remark that the precomputation cost in all cases is linear in N.

Corollary 3.5. *For the case $m_0 = N$, the precomputation costs are respectively $P_{naive} = Nt$ and $P_{\min} \approx 2N(H_{t+1} - 1)$, with H_n the n-th harmonic number.*

Proof. The expression for $P_{\min}$ results from instantiating $2N\left[H_{t+\gamma-1} - H_{\gamma-1}\right]$ (similarly to the proof of Theorem 3.4) with $\gamma = 2$ (from Equation (2.7)). □

From Theorem 3.4, for instance, that using a filter in each column with a typical $r = 20$ reduces the number of performed hash operations by about 85% (regardless of N or t). Table 3.1 and 3.2 display the maximum speedup $P_{\text{naive}}/P_{\min}$ that can be obtained when filtering in each column with respect to no intermediary filtering. Results in Table 3.1 are valid for non-maximal tables with $r \ll t$, while results in Table 3.2 are valid for maximal tables.

Table 3.1. – Speedup for quasi-maximum tables for various values of r.

r	10	15	**20**	30	50
$\frac{r}{\ln(1+r)}$	4.17	5.41	**6.57**	8.74	12.72

Table 3.2. – Speedup for maximum tables of various lengths.

t	1 000	10 000	100 000
$\frac{t}{2(H_{t+1}-1)}$	77.03	568.98	4508.50

In the case of non-maximal tables, the speedup only depends on r. In line with the intuition, the larger r, the greater the gain. Indeed, if r is large, m_0 is big and thus filtering is crucial. For maximal tables, as the number of hash operations performed in the minimal case depends on the $(t+1)$-th harmonic number, the speedup naturally increases with the increase of t.

3.3.3. Filtration in Chosen Columns

3.3.3.1. Optimal Placement

Results provided in Section 3.3.2 consider the number of hash operations, but they do not consider the additional time due to filtration and communication. Considering this additional time, it appears that filtering in every column is counter-productive in practice. Indeed, using few filters is more efficient in practice than using t filters. We denote by χ the number of filters used to generate a table of $t+1$ columns, with $\chi \ll t$.

When considering χ filters instead of t, the difficulty is to find the optimal position of each filter to minimize the total number of operations to perform. We introduce the precomputation cost using χ filters denoted as $(g_1, g_2, ..., g_\chi)$, and placed in columns $(c_{g_1}, c_{g_2}, ..., c_{g_\chi})$ in Equation (3.4). By convention and for equation simplification, we denote $g_0 = 0$ and $g_\chi = t$.

$$P = \sum_{i=0}^{\chi-1} m_{c_{g_i}}(c_{g_{i+1}} - c_{g_i}). \tag{3.4}$$

If χ is too close to t, several filters could be assigned to the same column. Choosing $\chi \ll t$ is not a problem, as it will be presented in Figure 3.3. Equation (3.4) is obtained straightforwardly from Equation (3.3), since instead of computing one column per column, at each filtration, $(c_{gi+1} - c_{gi})$ have been computed instead.

Using Equation (3.4), Theorem 3.6 gives the optimal placement of χ filters and the number of hash operations to perform to generate a table using these χ optimal filters.

Theorem 3.6. *The optimal average number of hash operations for precomputation with χ filters and $\chi \ll t$ is:*

$$P = 2N\chi \left[\left(\frac{t+\gamma-1}{\gamma} \right)^{\frac{1}{\chi}} - 1 \right].$$

The optimal placement of the filters is given by:

$$c_{g_i} = \gamma \left(\frac{t+\gamma-1}{\gamma} \right)^{\frac{i}{\chi}} - \gamma + 1.$$

Proof. We have $P = \sum_{i=0}^{\chi} m_{c_{g_i}} (c_{g_{i+1}} - c_{g_i})$. Deriving for each filter column position, we obtain:

$$\frac{\partial P}{\partial c_{g_i}} = \frac{\partial}{\partial c_{g_i}} \left[m_{c_{g_i}} (c_{g_{i+1}} - c_{g_i}) + m_{c_{g_{i-1}}} c_{g_i}) \right].$$

Inserting $m_i = \frac{2N}{i+\gamma-1}$ we have:

$$\begin{aligned} \frac{\partial P}{\partial c_{g_i}} &= 2N \frac{\partial}{\partial c_{g_i}} \left[\frac{c_{g_{i+1}} - c_{g_i}}{c_{g_i} + \gamma - 1} + \frac{c_{g_i}}{c_{g_{i-1}} + \gamma - 1} \right] \\ &= 2N \left[\frac{1}{c_{g_{i-1}} + \gamma - 1} - \frac{c_{g_{i+1}} + \gamma - 1}{(c_{g_i} + \gamma - 1)^2} \right]. \end{aligned}$$

To minimize P, we must have $\frac{\partial P}{\partial c_{g_i}} = 0$, and thus:

$$c_{g_i} = \sqrt{(c_{g_{i-1}} + \gamma - 1)(c_{g_{i+1}} + \gamma - 1)} - \gamma + 1.$$

It is easy to verify that a solution to this recursive relation with terminal conditions $c_0 = 1$ and $c_{g_\chi} = t$ is:

$$c_{g_i} = \gamma \left(\frac{t+\gamma-1}{\gamma} \right)^{\frac{i}{\chi}} - \gamma + 1.$$

Replacing in the expression for P gives the expected result.

□

3.3.3.2. Number of Filters

Theorem 3.6 is applicable only when $\chi \ll t$. In this section, we show that only a few filters are necessary to closely approximate the minimal precomputation lower bound $P_{\min}$.

Firstly, Figure 3.3 illustrates Theorem 3.6 for $N = 2^{42}$, $t = 1\,000$, and $r = 20$. The blue curve depicts the number of hash operations needed for precomputation with varying number of filters χ, placed according to Theorem 3.6. It also displays the lower bound $P_{\min}$ (black dashed line) in terms of hash operations, which is reached when a filter is applied in each column. The case $\chi = 1$ corresponds to P_{naive} (filtration only in the last column).

It indicates diminishing returns in increasing χ. For instance, in that scenario, using a single filter decreases P by a factor of 2.88. However, using a filter in each column is only about 1% faster than using 50 filters.

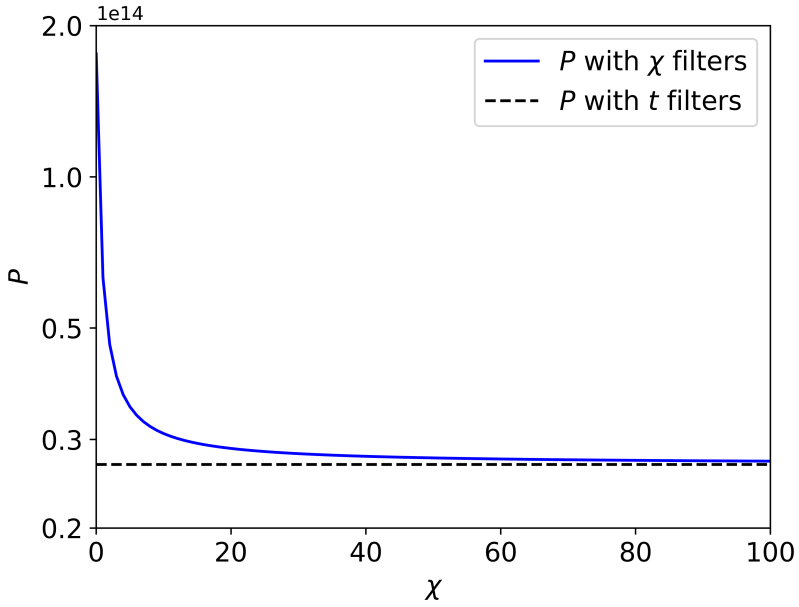


Figure 3.3. – Number of hash operations P according to the number of filters used χ, with $N = 2^{42}$, $t = 10\ 000$, and $r = 20$. The value P is evaluated with χ filters according to Theorem 3.6 and P with t filters according to Theorem 3.4.

It is worth noting that the optimal distribution of filters is actually not uniform. Detecting merges as soon as they occur avoids wasting time computing the useless remaining parts of the chains. As a consequence, filters are mostly located on the left hand side of the chains. Figure 3.4 illustrates this non-uniformity. It depicts the position and number of hashes needed to generate a matrix according to the number of filters, placed according to Theorem 3.6.

The area under the blue line represents the cost of precomputation without any filter. The area under the dashed black curve is the minimal number of hash operations to perform in order to generate the matrix. Green, purple, and red curves represent the number of surviving chains remaining in each column when using one, height, or thirty filters, respectively. The area under these three curves thus represents the total number of hash operations to perform to generate the matrix using one, height, or thirty filters.

Figure 3.4 shows that even using one filter considerably decreases the number of hash operations to perform but still computes 57% more operations than the theoretical lower bound. However, using only height filters allows for achieving only 15.8% more operations than the theoretical lower bound, and using thirty filters allows for computing less than 5% more operations than the theoretical lower bound. For clarity, we have not represented the curve for more filters, but using fifty filters allows obtaining a number of hash operations to perform approximately 2% more than the theoretical lower bound and using one hundred filters allows for computing 1% more hashes than the theoretical lower bound.

Figure 3.4 also illustrates that the majority of filters are placed in the columns in the left part of the matrix. When using 8 filters, only one is in the right part of the matrix (after column 5000), and when using 30 filters, only 6 filters out of the 30 are in the right part of the matrix.

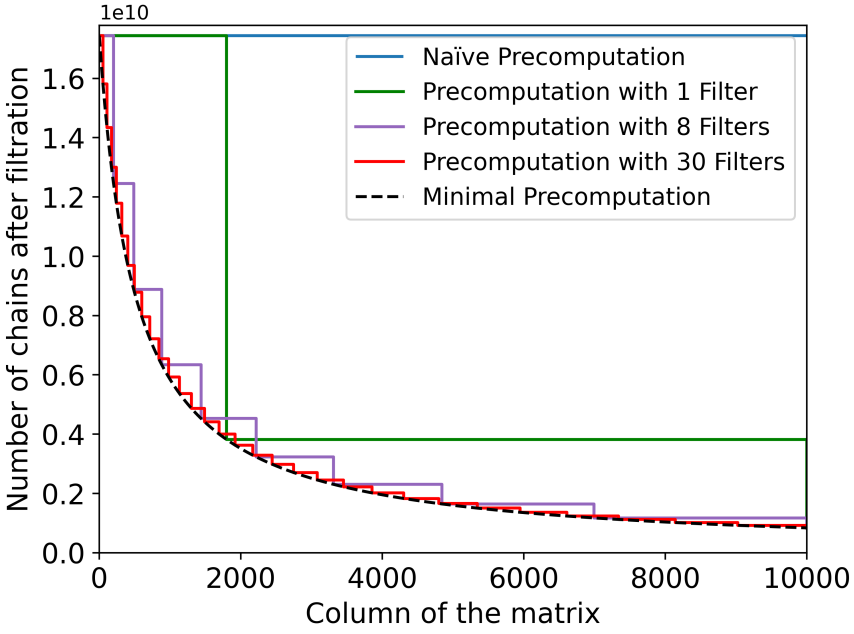


Figure 3.4. – Precomputation according to the number of filters used, with $N = 2^{42}$, $t = 10\ 000$, and $r = 20$. Filters positioned according to Theorem 3.6.

3.4. Conclusion

In this chapter, we have addressed the issues associated with the characterization of the generation of non-maximal tables using m_0 SPs, particularly when $m_0 \ll N$. We introduced the maximality factor, α, to simplify the process of obtaining a targeted m_t number of EPs. We provided formulas to predict the required m_0 value for achieving, on average, $\alpha m_t^{\max}$ distinct EPs, thereby allowing the generation of tables for a targeted success probability and a targeted memory. We have also shown that by choosing a value of $r \ll t$, high-quality tables can be generated while significantly reducing the precomputation cost.

Additionally, we introduced the filtration method, which consists of cleaning (or filtering) the matrix several times during the precomputation process instead of only in the final column. We explored the optimal placement of filters within a matrix to minimize the number of hash operations required for precomputation. We provided a method for determining the optimal placement of filters and demonstrated its effectiveness.

In conclusion, our findings offer valuable insights for the generation of non-maximal tables and the strategic placement of filters to optimize the precomputation process. These insights will be used throughout the chapters of this thesis. In the subsequent chapters, only non-maximal tables using α as a parameter of their maximality will be analyzed. Moreover, the filtration method will be applied in every chapter to generate tables.

4 Distributed Filtration

A RT is coming back by car from a party and gets stopped by a police officer:

- Are you clean?

- Of course!

- Perfect!

4.1. Motivations

In Chapter 3, we introduced methods for characterizing non-maximal tables and presented the filtration method. Although we assume that the filtration method was informally used prior to our work, we believe that optimizing filter placement has not been explored. Additionally, no methods for distributing the precomputation of RTs or conducting such precomputation experiments were proposed.

Generating RTs on a single CPU for a realistically sized space N is infeasible. The problem addressed in this chapter is the following: given a total number of available nodes (or cores) n, how to distribute the precomputation phase across the n nodes. After determining the number of *computing nodes* or *hashing nodes*, n_h that will perform the computation of chains and the number of *filtration nodes*, n_f that will clean the matrix during precomputation, such that $n_h + n_f = n$, we propose a method to distribute the precomputation of RTs using the filtration method and the n available nodes. We also provide a way to optimize the positions of filters, taking the number of computing nodes, filtration speed, communication time, and potential overhead due to implementation into account. We then test our method on two different environments and evaluate the gains compared to a distributed generation without using filtration.

Takeaway:

In this chapter, we introduce a method for distributing the precomputation phase of RTs. Our conclusions are as follows:

- Distributing the precomputation phase using the filtration method is both possible and worthwhile.
- We reduced the precomputation time of RTs by more than six times compared to a distributed precomputation without the filtration method.
- Even when communication times are significant, the filtration method remains beneficial.

4.2. Distributing Precomputation

In this chapter, the precomputation time is evaluated in seconds rather than in number of hash operations, to provide realistic results using various environments. When relying solely on the number of hash operations, it is not possible to account for the unique aspects of each experimental environment, such as differences in hash speed, communication time, etc.

Reminder:

- A *sub-chain* is a chain that starts with either a SP or an IP and ends with either an IP or an EP. A sub-chain is a part of a chain of length t.
- The *length* of a chain or a sub-chain is the number of applications of the hash-reduction function performed to construct it.

4.2.1. Benefits

For the sake of clarity, the problem is illustrated with concrete examples and parameters that are used several times throughout this work, i.e., $N = 2^{42}$, and $r = 20$. The choice of these parameters is justified in Section 4.3. For simplicity, in this section, a single table ($\ell = 1$) is considered. The scenario for multiple tables can be inferred by multiplying the total time by the number of tables.

To provide real-world context, the computation time is illustrated using one or more nodes with computational capabilities similar to those presented in Section 4.3. These nodes can be characterized as having upper mid-range performances, in particular, they are considered as components with a single CPU core.

To justify the interest of using both distribution and filtration, the expected results of filtration without distributed precomputation, and distribution without filtration, are presented below.

4.2.1.1. Filtration Without Distribution

When the precomputation is not distributed, only one node is used for both filtration and sub-chains computation. Hence, the filtration cannot be executed in parallel with the computation of sub-chains. The node must either filter or compute, not both simultaneously. On a single node environment, precomputing a table for the specified space would take 40.5 days, even with optimal use of filtration. While this is significantly better than the 253 days required without filtration, it is still too long compared to what would be possible by using both filtration and distribution.

4.2.1.2. Distribution Without Filters

Distributing the precomputation phase without using filtration is a common practice. It is easy to implement and offers only benefits compared to non-distributed precomputation. Given that there are $m_0 t$ hashes to be computed and thus m_0 chains to be computed, it is sufficient to distribute the m_0 chains to be computed across the available cores. Thus, if n cores are available, the precomputation time is the time that a single core takes to perform $m_0 t/n$ operations.

If the available cores do not have equal capacities, a weighting constant multiplies the $m_0 t/n$ value to ensure that not all nodes are assigned the same number of chains to compute. With an environment of 128 cores, similar to the one used in Section 4.3, computing a table for the above-mentioned space and parameters would take 47.5 hours.

Although 47.5 hours is better than filtering without distributing, it is still far from optimal. Given these parameters, 95% of the computed chains are discarded at the end of precomputation instead of progressively during the precomputation. The number of hashes computed exceeds the minimum number of hashes that could be computed by filtering in each column by a factor of just under 9.

4.2.2. Possible Distributions

Given the waste of time represented by the use of a distributed environment that does not use filtration or by the use of an environment that is not distributed, the interest in using a distributed environment while still using filtration is significant.

When considering the distributed method with filtration, several distributed architectures can be considered. The architectures that appear to be the most relevant to us are briefly described below, and the chosen architecture is more formally described in Section 4.2.5.

The challenge of distributed precomputation with filtration arises from the fact that to filter chains, they need to be gathered. It is not straightforward or costless to filter chains when they are all scattered across different nodes. Thus, the challenge lies in performing computations on different nodes while still managing to filter all chains.

4.2.2.1. Centralize

The first architecture involves performing filtration on a single node. If n nodes are available, then the chains to be computed are divided into batches and sent to these $n-1$ nodes, which compute the sub-chains up to the first filter and then return them to the filtration node for filtering.

One challenge lies in determining the batch size. Naively, each of the $n-1$ computation nodes can compute $m_0/(n-1)$ sub-chains up to the filtration column and return them all at once to the filtration node, which, after performing the filtration, sends them back to the computation node. However, this implies that filtration would not be performed in parallel with the computation of chains, resulting in a loss of time.

Instead of using batches of $m_0/(n-1)$ sub-chains to be computed, smaller batches are used (see Section 4.2.5.1), and the precomputed pairs (SP, IP) are returned progressively to the filtration node, which can thus begin filtering while the computation nodes are working.

The downside compared to the naive version is that this requires more communication between nodes and the filtration node. As seen in Section 4.4, this problem can be mitigated by reducing the number of filters or increasing the batch size. The challenge, therefore, lies in finding the optimal batch size and number of filters for a given environment (see Section 4.3).

4.2.2.2. Decentralize

Another approach to the problem is to use multiple filtration nodes. This architecture is particularly interesting if the filtration time is large.

The principle is to use n_h computation nodes and n_f filtration nodes, with $n_f > 1$ and $n = n_h + n_f$. In this architecture, the computation nodes receive batches of chains from different filtration nodes and return the computed pairs (SP, IP) to specific filtration nodes.

The difficulty here is that if n_f filtration nodes are used, one must ensure that filtered chains do not end up duplicated on two different filtration nodes, which would lead to unnecessary computation.

To prevent this, once a computation node has computed a batch, it subdivides it into n_f sets of pairs (SP, IP) based on a specific criterion on the IP of each chain, and sends each set to the appropriate filtration node. The criterion can be on the value of the bits by which each IP begins. For instance, if $n_f = 2$, when a computation node has finished computing a batch, it subdivides the computation chains into two groups: those with an IP starting with 1 and those with an IP starting with 0, and then sends each set to the filtration nodes responsible for sorting batches with IPs starting with 0 and 1, respectively.

The division of computed chains into multiple groups or sets can be done as the computation nodes precompute the chains, and it does not incur an additional time.

The drawback of this method is that it requires a larger number of exchanges between the filtration and computation nodes. For each batch, instead of returning the result of the sub-chains precomputation to a single node, the computation node must return its result to n_f nodes. Similarly, although it seems sufficient for each computation node to receive batches from only one filtration node, in some cases, they may need to receive them from several ones.

The advantage of this architecture is that the filtration time is divided by n_f, so if this time is prohibitive, then using this architecture is indicated.

4.2.2.3. Hybrid

This architecture involves having all n nodes computing and filtering simultaneously. In this case, each of the n nodes is thus, both a computation and filtration node.

In the same way as the decentralized architecture, each of the n nodes will precompute a batch of chains of a given size, split the result into n sets and return each of them to a specific node based on a given criterion on the IP of each sub-chain.

Thus, if n nodes are available, a criterion on the first $n/2$ bits of the IP is chosen, each time a node has finished computing chains and subdividing them into n sets, it sends $n - 1$ sets to the other nodes and keeps the last one for itself. It can then filter the chains it received while computing the sub-chains with the one it kept for itself and then starts the precomputation again.

This architecture requires to solve numerous challenges. The first is the synchronization of all the nodes. It must be ensured that sub-chains computed up to a column c_j are not filtered with sub-chains computed up to a column c_i with $c_i \neq c_j$. Therefore, whether a signal must be set up so that each node waits at each filter for all others to reach the same level, or each node must be able to differentiate between batches that have been computed up to each filtration column. The latter solution implies increasing the number of subdivisions into groups and hence the complexity.

Moreover, if all nodes do not have the same capacity, then the process becomes even more complex. Finally, this architecture is the one that requires the most communication costs since each node must communicate with all the others continuously.

The advantage, however, is that if shared memory is limited, then this allows for spreading memory costs over n nodes rather than just one for the centralized architecture and n_f for the

decentralized architecture. Similarly, if filtration is extremely slow, it allows for the maximum spread of this process.

4.2.3. The Chosen Architecture

Given the environments at our disposal, the implementation is based on the centralized architecture. The environments at our disposal (see Section 4.3 for their specifications) have an extremely fast filtration time. Furthermore, one of the two environments has a relatively low communication speed (at least it cannot be described as fast), using this architecture allows us to limit the communication between each node while maximizing the time spent on computation, knowing that it is this cost that is prohibitive, not the filtration.

4.2.4. Compatibility with RTs Improvements

Before presenting the experiments performed to demonstrate the effectiveness of the filtration method, we briefly revisit the various existing improvements to the attack phase (see Chapter 2 for a comprehensive presentation of each improvement) and explain why the precomputation using the filtration method will not impact the attack phase when employing these improvements.

- Chain storage optimizations (prefix/suffix decomposition or compressed delta encoding [AC13]): lossless compression can be applied at the end of the table generation, with no impact on the precomputation process.
- Truncated EPs [ABC15]: EPs can be truncated at the end of the table generation, again with no impact on the precomputation time.
- Checkpoints [AJO08, ABC15]: Saving checkpoints can be done during the filtered and distributed precomputation with ease, although specific care must be taken. Hashing nodes must be made aware of which columns are checkpoint columns, and the filtration node needs to keep track of this. This adds no significant burden on either of them.
- Heterogeneous tables [AC17]: Precomputation of tables of different shapes is done independently, regardless of whether they operate on the same input set. Consequently the use of heterogeneous tables has no impact on precomputation improvements.
- Interleaving [ACL15]: Just like with heterogeneous tables, the different tables are independently computed, again having no impact on precomputation improvements.

4.2.5. Estimation of the Precomputation Time

4.2.5.1. Precomputation Process

> **Reminder:**
>
> - n_h is the number of computation nodes.
> - n_f is the number of filtration node and in our case $n_f = 1$.

In an environment with a single filtration node, this node is also responsible for managing the sequencing of various tasks. A *job* is defined as a collection of sub-chains to be computed

between two filters. A job of size j_s contains j_s (SP, IP) pairs. Precomputations are therefore divided into two main parts: jobs to be performed between filters and the filtration of these jobs. Precomputations involve managing, for each filter, the dispatch of jobs to the hashing nodes and the filtration of the completed jobs.

The rationale behind bundling sub-chains into jobs is to mitigate the communication overhead. For instance, using $j_s = 1$ is a poor choice, as the overhead due to communication would significantly hinder performance. Contrarly, using a very large j_s may result in additional idle time for computing nodes, for example, if there are no available chains left to compute for an idling computing node while other computing nodes are still busy[1]. In the following discussion, we assume that the value of j_s is reasonable.

Upon receiving a job, a hashing node starts from each IP to compute the new IPs corresponding to the column of the next filter. Once these computations are complete, it returns the SPs and the new IPs to the filtration node, which sends a new job back to the latter. The filtration node's purpose is to receive jobs from hashing nodes and send new ones as soon as the jobs are received. Meanwhile, the *filtration node filters already received jobs*. This procedure is repeated between each filter until the end of the computation phase. As described in Section 3.3.3, filtering in every column is not feasible in a distributed architecture due to the added communication and filtration overhead. The problem of choosing the positions of the filters is discussed in Section 4.2.6.

4.2.5.2. Total Precomputation Time

> **Reminder:** In this chapter, we evaluate the precomputation time in seconds and not in number of hash operations.

To perform the table generation, hashing nodes compute sub-chains while the filtering node filters and sends jobs to be computed simultaneously. In this section, we evaluate the time H, required for hashing only, the time F, for filtering, the overhead O, due to filtration, and the time C, spent on communication only. Subsequently, we provide Formula 4.5 to evaluate the total precomputation time. This formula is only applicable if the filtration is performed in parallel with the hashing.

Hashing time (H). Jobs computations are carried out by hashing nodes. Given χ filters with $\chi < t + 1$, the total number of hash operations to be performed is $\sum_{i=0}^{\chi-1} m_{c_{g_i}}(c_{g_{i+1}} - c_{g_i})$ (see Equation 3.4) with c_{g_i} the column of the i-th filter, $c_{g_0} = 0$ and $c_{g_\chi} = t + 1$. A hashing node can perform v_h applications of the reduction function $f_i = R_i \circ h$ per second. Parameter v_h represents the hashing speed and is determined before the beginning of precomputation and depends of the hashing nodes performance. Computations of chains are considered to be done in parallel, by n_h hashing nodes with equal performance. The total hashing time can therefore be estimated as:

$$H = \frac{1}{n_h v_h} \sum_{i=0}^{\chi-1} m_{c_{g_i}}(c_{g_{i+1}} - c_{g_i}). \tag{4.1}$$

[1] Experiments show that, for the typical problem size and architectures considered, choosing j_s to be anywhere from 1 000 to 100 000 mitigates both of these issues. The particular choice of j_s therefore has negligible impact on the performance, provided it lies within that range.

Filtration time (F). For a filter in column c_{g_i}, the number of elements that have to be filtered is $m_{c_{g_i}}$. The total number of elements that have to be filtered in the entire precomputation is hence $\sum_{i=0}^{\chi} m_{c_{g_i}}$. We consider that a filtration node can perform v_f filtration per second. The total time due to filtration is thus:

$$F = \frac{1}{v_f n_f} \sum_{i=0}^{\chi} m_{c_{g_i}}. \tag{4.2}$$

We also consider a potential overhead due to the filtration. This can for instance result from processing the output of the filtration into jobs to be sent to hashing nodes. This overhead depends on the number of elements generated and depends on the implementation (filtration algorithm, memory allocation, etc.). Given v_o the average overhead time per element, the total overhead O can be expressed as:

$$O = v_o \sum_{i=0}^{\chi} m_{c_{g_i}}. \tag{4.3}$$

Communication time (C). Communication time is the time needed for sending jobs between filtration nodes and computing nodes. Let v_c be the average time for a job to be sent from a filtration node to a hashing node and back. We assume that when a communication is in progress with one hashing node, all the other hashing nodes are computing. The impacting communication time can then be expressed by:

$$C = \frac{v_c}{n_h} \sum_{i=0}^{\chi} m_{c_{g_i}} \tag{4.4}$$

Total time. Given that computation of sub-chains and filtration are performed in parallel[2], the most impacting component in the total time spent to generate a RT is the maximum time between the hashing time H and the filtration time F, i.e., $\mathrm{Max}(H, F)$. To obtain the total time, the communication time has to be added as well as the overhead time due to filtration. The total time P needed to generate a clean RT is hence:

$$P = \mathrm{Max}(H, F) + C + O. \tag{4.5}$$

4.2.6. Optimal Configuration

The number of filters and their positions have a considerable impact on the precomputation time. When considering real environments, the formulas provided by Theorem 3.6 in Chapter 3 give an approximation of the real optimal filter positions only. In this section, we describe the method that will be used to determine the optimal filter positions in practical cases. A comparison between placements provided by Theorem 3.6 and our method is given in Section 4.3.3.2.

[2]In general, for an architecture with a single filtration node, hashing time is much longer than the filtering time. If the parameters of the problem and the architecture are such that it is not the case, then another architecture with several filtration nodes should be considered.

Let a *configuration* be a set $S = \{c_{g_1}, \ldots, c_{g_\chi}\}$, where χ is the number of filters, and c_{g_i} the position (column number) of the g_i-th filter. Let S^*_χ be the configuration of χ filters that minimizes Equation 4.5, and $S^* = \min_\chi S^*_\chi$.

Due to the various operations outside of hashing, particularly the filtering process (which, to some extent, can be done in parallel to hashing) and other communication/data processing overheads, the configuration given by Theorem 3.6 typically yields sub-optimal results. For this reason, we rely on numerical minimization of Equation 4.5, which models the precomputation time given by our implementation.

We settled on a truncated-Newton method [Nas00], an optimization algorithm suitable for solving bounded optimization problems with many variables (see, e.g., [Sur00] for a thorough description). The minimization is used to find S^*_χ, coupled with an exhaustive search[3]. on χ. The optimality of the configuration found by the numerical minimization is based on the following two conjectures: (1) S^*_χ is a convex function of χ, and (2) Equation 4.5 is smooth enough (w.r.t. S) to guarantee or approach the conditions of optimality of the truncated-Newton search[4] [Nas00]. We offer no proof of these conjectures but note that (a) they seem to be true both intuitively and after extensive testing and (b) if these conjectures do not hold, it could only lead to better results.

Regardless of the validity of these conjectures, the configuration obtained through numerical minimization presents a significant improvement over the analytical minimization that assumes no overhead or filtration cost (Theorem 3.6). Moreover, the estimated precomputation time comes very close to the theoretical minimum, as detailed in Section 4.3.

4.3. Experimental Set Up

Reminder:

- Three parameters are used to characterize a RT, namely, the number of column t, the quasi-maximality factor α and the number of tables ℓ.
- Variable r is used to simplify computation of m_0 and is equal to $r = \frac{\alpha}{1-\alpha}$.

In this section, we present experiments that consist of implementing the distributed filtration method in two distinct environments with varying components, hash speeds, communication speeds, etc. We first describe the environments used, followed by the implementation of filtration, filter positioning, and parameter selection, as well as the rationale behind these choices. Finally, our results are presented in Section 4.4.

We have chosen to perform our experiments for a size $N = 2^{42}$ and for a value $r = 20$ (corresponding to $\alpha \simeq 0.95$). This choice is justified for N due to its applicability in practical scenarios (such as retrieving passwords of 8 characters), as well as for ease of comparison with other contributions [AC13, AC17, ACKT17]. We selected $r = 20$ because it is a typical value that enables achieving high success probability while allowing for a significant reduction in m_0 compared to the maximal case $m_0 = N$. Furthermore, for simplicity, the analysis is made for $\ell = 1$.

[3] To maintain efficiency, the search is from 0 up to a reasonable upper bound $\chi_{\max}$. A more sophisticated approach could be used here (e.g., Newton descent on χ), but we found it to be unnecessary.

[4] In our case, the conditions are strong convexity and Lipschitz-continuous Hessian.

4.3.1. Computing Environments

Our experiments were conducted in two distinct environments, described below. Prior to beginning the precomputation phase, we benchmarked these environments to measure the hashing speed v_h, the filtration speed v_f, the overhead constant v_o associated with the implementation of filtration, and the communication cost v_c. The benchmarking process involved generating tables for a small-sized problem ($N = 2^{32}$), with filters positioned according to Theorem 3.6.

Environment 1 is an environment with high communication speed while Environment 2 has lower communication speed. We show in Section 4.4 that the communication speed does not change significatively, the results obtained.

Environment 1 consists of a computer hosting two AMD EPYC 7742 3.2 GHz processors, each comprising 64 cores, resulting in a total of 128 cores. We used 127 cores to ensure full exploitation during the precomputation, reserving the remaining core for basic system operations. The benchmark measured $v_h = 7\ 747\ 002$ hashes per second, $v_f = 15\ 949\ 709$ filtrations per second, and $v_0 = 1.37 \times 10^{-10}$. The communication overhead is negligible compared to v_h and v_f, therefore $v_c = 0$ can be assumed, which implies that $v_o + \frac{v_c}{n_h} = v_o = 1.37 \times 10^{-10}$ seconds are required to process one element.

Environment 2 comprises a cluster of eight computers, each equipped with two CPUs and 14 cores per CPU, totaling 224 cores. Each CPU is an Intel Xeon E5-2680 v4 (Broadwell, 2.40GHz, 14 cores). The computers are directly connected via switches, facilitating communication through the Ethernet protocol. The benchmark yielded $v_h = 6\ 403\ 611$ hashes per second, $v_f = 7\ 918\ 745$ filtrations per second, and $v_o + \frac{v_c}{sn_h} = 7.5 \times 10^{-10}$ seconds to process one element.

4.3.2. Filtration Implementation

For the filtration, we employed an open addressing hash table [ML75, MC86] with the following parameters:

- A load factor $\lambda = 2/3$, which strikes a balance between size overhead and a low probability of collision [SW11].
- The number of slots in the table is $k = m_{c_i}/\lambda = 1.5m_{c_i}$, with m_{c_i} representing the theoretical number of surviving chains after filtration (provided by Equation (2.7)).
- The hash function utilized is $IP \bmod k$.

We implemented linear probing [Knu63, Pet57, Ers58] for collision resolution (with an interval of 1). This method is suitable because inputs to the hash table are uniformly distributed in A (by construction). Each filtration proceeds as follows:

- A hash table of size $k = 1.5m_{c_i}$ is created.
- As soon as a job is received by the filtration node, it is filtered as follows:
  1. For each pair (SP, IP) in the job, $IP \bmod k$ is computed.

2. The index IP mod k of the table is checked.
3. If no value is present at the computed index, then IP and its corresponding SP are inserted at this index.
4. If the value equal to IP is present at the computed index, a merge between two chains has occurred, and IP and its corresponding SP are removed.
5. If a different value of IP is present at the computed index, a new index value is computed as $IP + 1$ mod k, the index $IP + 1$ mod k is checked, and steps 3 to 5 are repeated.

- Once all jobs have been filtered, the hash table is scanned, and all IPs and their corresponding SPs are transferred by copying them to an array in a format that facilitates job transmission.
- The hash table is deleted.

4.3.3. Parameters Choice

> **Reminder:** The last parameter to set among the three that characterize a RT, is the number of columns t. When the filtration method is used, optimized number of filters and their position also need to be set.

This section presents experiments conducted on the environments to determine the optimal parameters for both environments, aiming to minimize the precomputation time.

We conducted experiments on Environments 1 and 2, as they have different components. The results were similar in both environments. For brevity and simplicity, however, we only present experiments conducted on Environment 1 in this section. The precomputation times for both environments are presented in Section 4.4.

4.3.3.1. Number of Columns

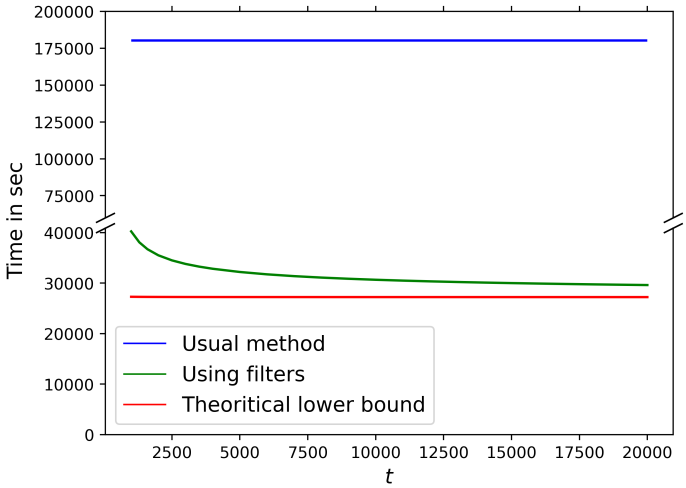


Figure 4.1. – Time (sec) to generate a clean RT according to t ($N = 2^{42}$, $r = 20$)

The first parameter to select is the number of columns t. The primary objective of this section is to ascertain if a given number of columns t is less optimal or less appealing than others. To achieve this, we compute the minimal precomputation time for various values of t using filters. We vary t and the optimal number of filters to find the minimal precomputation time for all t using Equation 4.5 (we refer to Section 4.3.3.2 for a description of how the optimal number and position of filters are determined for each t). We then compare this precomputation time with the theoretical lower bound and the usual method (without filter).

The chain length impacts the online phase: when t decreases, the time required for the online phase decreases, but the required memory increases. When filters are employed, the chain length also affects the precomputation phase, but this effect is substantially smaller, as illustrated in Figure 4.1. In this figure, the blue curve represents the precomputation time in seconds using the conventional method (no filters), according to the value of t. The green curve represents the precomputation time when employing the filtration method with an optimal number and placement of filters. The red curve is the theoretical lower bound. It can be observed that the smaller t is, the greater is the precomputation time, which is due to the fact that when t is too small, fewer filters are used, thus increasing the precomputation time. Conversely, when t is too large, the impact of additional filters is not significant and, therefore, less impactful. It is noteworthy that our experimental results are close to the theoretical lower bound, and our method is substantially more efficient than the conventional way of generating tables, as detailed in Section 4.4.

We selected $t = 10\ 000$ because this value results in a fast online phase on the order of a few seconds with reasonably-sized memory (for $N = 2^{42}$). For instance, choosing $t = 20\ 000$ would yield a 20% faster precomputation but would quadruple the time of the online phase. As depicted in Figure 4.1, this value allows us to achieve a theoretical precomputation time sufficiently close to the theoretical lower bound.

Corollary 3.2 provides the number of SPs m_0, with $m_0 = 2rN/t \approx 1.76 \times 10^{10}$. According to Proposition 3.1, the expected number of chains in a single column using our chosen value of t ($t = 10\ 000$) is $0.9524, m_t^{\max} \approx 8.38 \times 10^8$.

4.3.3.2. Filters

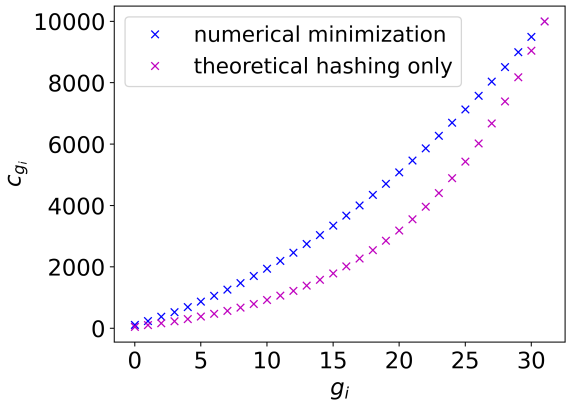


Figure 4.2. – Positions of the 31 filters

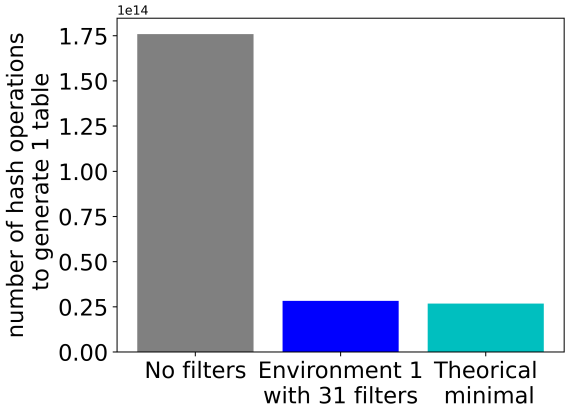


Figure 4.3. – Number of hash operations

To determine the optimal positions and number of filters, we employed the Truncated

Newton Constrained (TNC) algorithm (Section 4.2.6) applied to Equation (4.5). We then compared the positions obtained with the optimal number of filters acquired with TNC to those obtained with Theorem 3.6. Finally, we compared the number of hashes to perform according to the technique used.

Figures 4.2 and 4.3 present these results when considering the hash speed, filtration speed, communication, and overhead of Environment 1. For brevity, we only present results for Environment 1 since we obtained similar results for Environment 2.

We initially computed the optimal configuration for Environment 1 using TNC. For our parameters $N = 2^{42}$, $r = 20$, and $t = 10\ 000$, the optimal configuration consists of 31 filters. Figure 4.2 illustrates the position of the 31 filters placed according to TNC (blue cross) and according to Theorem 3.6 (violet cross), where communications are free. The theoretical minimization that considers only hashing (and not filtration nor communication) places filters in columns more to the left of the table (lower column) than the numerical minimization of Equation (4.5). The difference between the positions of filters is the vertical distance between a blue cross and a violet cross. The maximal difference is obtained for the 21st filter with a difference of 1,910 columns.

Figure 4.3 displays the number of hash operations required to generate a clean RT in Environment 1 when there are no filters (left case), which was the state of the art prior to our contribution; when there are optimally placed filters, which is our approach (middle case); and when filtration and communication are free, with a filter in each column, which is the theoretical lower bound (right case) obtained from Equation (3.3). It is worth noting that our approach is tightly close to the theoretical lower bound (approximately 10% of the theoretical lower bound).

We thus conclude that our parameters $N = 2^{42}$, $r = 20$, $t = 10\ 000$, and 31 filters placed according to TNC are valid and suitable for generating tables.

4.4. Results

We generated a RT on Environment 1 and Environment 2, using the optimal parameters determined in Section 4.3. This section presents the results obtained for each environment. Table 5.5 summarizes these results.

Takeaway:

- In an environment with high communication speed, distributed filtration reduces the precomputation time by a factor of approximately 6.
- In an environment with low communication speed, distributed filtration reduces the precomputation time by a factor of approximately 5.

Environment 1. Precomputing a single RT without any filter requires $m_0 t$ hash operations in our scenario, corresponding to 100×2^{42} operations. Given that $v_h = 7\ 747\ 002$ and the environment consists of 127 cores (each core corresponding to one node) with one filtration node and 126 hashing nodes, the precomputation time is estimated to be 180 225 seconds (50 hours and 3 minutes), which is very close to our experimental result of 179 850 seconds (49 hours and 57 minutes).

Using thirty one filters optimally placed significantly reduces the precomputation time. Using Equation (4.5), we obtain that the predicted precomputation time is as low as 30 657 seconds (about 8 hours). Filters thus divide the precomputation time by 5.9 in this scenario. The experimental result is 31 029 seconds, which is very close (1%) to the theoretical result.

Table 4.1. – Summary of the results ($N = 2^{42}$, $t = 10\ 000$, $r = 20$)

Scenario	#Filters	#Hashes ($\times 10^{12}$)	#Cores	Time	
				Experimental	Predicted
Environment 1 (state of the art)	0	176	127	179 850	180 225
Environment 1 (our approach)	31	28	127	31 029	30 657
Environment 1 (theoretical bound)	10 000	26.8	127	-	27 452
Environment 2 (state of the art)	0	176	224	123 717	123 194
Environment 2 (our approach)	11	31	224	26 499	26 139
Environment 2 (theoretical bound)	10 000	26.8	224	-	18 765

Environment 2. According to the TNC algorithm, without any filter, the precomputation time of a single RT should be approximately 123 194 seconds (about 34 hours and 13 minutes).

The computed optimal configuration for this environment is the utilization of eleven filters. For an easy comparison, we choose $t = 10\ 000$ as for Environment 1.

When eleven filters optimally placed are used, our experimental results show that a table can be generated in 26 499 seconds (about 7 hours and 20 minutes). According to the TNC algorithm with the parameters for this second environment, given in Section 4.3.1, this precomputation time is estimated to be 26 139 seconds (about 7 hours and 12 minutes). Our experimental results are therefore very close to the predicted ones.

Utilization of filters in this environment allows generating a table five time faster than the naive method. Table 5.5 provides a summary of the results obtained for Environments 1 and 2.

4.5. Conclusion

This chapter presents the distributed filtration method for precomputing RTs. Given that the precomputation phase is highly resource-consuming, the distributed filtration method offers

significant practical benefits. We also provide formulas to compute optimal filter positions and evaluate precomputation time.

Our technique is illustrated using a typical scenario with a problem size of $N = 2^{42}$ ($t = 10\ 000$ and $r = 20$). In such a scenario, the precomputation phase requires 1.76×10^{14} hash operations, taking approximately 50 hours on a 128-core computer (Environment 1). In contrast, our technique requires 2.8×10^{13} hash operations, which were completed (including filtering) in about 8 hours and 36 minutes on the same 128-core computer. The distributed filtration method thus reduces the expected precomputation time by a factor of approximately 6. It is also close to the theoretical lower bound of 27,452 seconds, i.e., 7 hours and 33 minutes (for $r = 20$), with the difference attributed to filtering and communication overheads.

We considered a typical scenario in that we used quasi-maximum tables ($r = 20$) instead of maximum tables, which were often considered in the literature before this thesis's contributions. Maximum tables (corresponding to $r = 5,001$ in our case) are not used in practice due to the prohibitive precomputation requirements. Therefore, we focus on quasi-maximal tables throughout this thesis. However, considering maximum tables would make the distributed filtration method even more valuable (e.g., increasing the speedup from 6 to 4,500 with the same parameters).

It is worth noting that our technique for speeding up the precomputation phase has been applied to vanilla RTs. We assert in this chapter that it is fully compatible with improvements published in recent years to enhance the attack phase's efficiency. Moreover, this technique will be applied throughout this thesis, as it significantly accelerates the precomputation phase without impacting the attack phase, and is applicable to all variants that will be introduced in this work.

5 Rainbow Tables on CPU

Why do Rainbow Tables researchers never stop computing? ***They hope that by tirelessly searching at the end of the rainbow, they will find a leprechaun!***

5.1. Motivations

This chapter presents an analysis of the primary bottleneck on the RTs performance. Although traditional research often emphasizes the attack phase as the limiting factor, the focus of this chapter is shifted towards the precomputation phase, assumed as the potential bottleneck to RT efficiency.

To probe this hypothesis, an analysis of the maximum feasible RT across various environments and scenarios is conducted. The intent is to characterize and identify the predominant performance limitation for RTs. The study is performed on CPUs based precomputation and aims to provide pragmatic, actionable insights.

The analysis takes four constraints into account: precomputation time, precomputation memory, attack time, and attack memory. The targeted coverage is 95%, which implies the use of four quasi-maximal tables. The application of formulas from Chapters 2 and 4 to a range of scenarios in different environments aims to highlight the main bottleneck preventing the extension of TMTO on bigger spaces.

To ensure the applicability of the findings, three environments are analyzed and various memory configurations are considered. Given the lack of TMTOs multi-core literature, the attack phase is assumed to be performed on a signle core.

Takeaway:

- The precomputation phase is the bottleneck preventing the generation of Rainbow Tables for larger spaces.
- In certain circumstances, increasing the memory available for the precomputation phase allow to decrease the attack time.
- It is not the attack time that prevents the generation of Rainbow Tables for larger spaces using CPUs.

5.2. Environments and Scenarios Considered

5.2.1. Context

In practice, the entities that perform TMTOs have different needs, purposes, and resources (e.g., available memory, amount of money to spend, time available, etc.). Resources are, in addition, not the same for the attack and precomputation phases. It is therefore necessary to define the context in which each phase is performed. A company, for instance, does not have the same resources than a nation state. The memory available for the precomputation as well as the one for the attack must be defined. Other variables, such as the available time, should also be specified in advance.

To define the context in which a TMTO is performed, we use the concept of *environments* and *scenarios*. An *environment* corresponds to the resources available on a given equipment (RAM, number of cores, CPU performances, etc.) while a *scenario* defines the resources available, namely time and memory.

In this chapter, various environments are considered and are described in Sections 5.2.3 (precomputation) and 5.2.4 (attack phase). These environments aim to represent different entities. For each environment, several scenarios are considered depending on the available time and money. For each scenario, the largest N that can be reached is evaluated to identify the technological bottleneck that prevents going further. Experiments are performed to illustrate the theoretical results.

Three distinct environments are considered for the precomputation phase and various memory are considered for the attack phase. However, due to the lack of literature on multi core based attack phases, a single core is considered for the attack. Two environments, referred to as `supercomputer` and `computer`, represent large computing systems with 128 cores. `supercomputer` is a computer typically listed in the top-100 worldwide[1], while `computer` represents a machine accessible to medium-sized companies or academic research teams. The third environment, the `cloud` environment corresponds to what could be expected on rented machines available in the cloud. The latter could be economically interesting if the precomputation is performed only once or occasionally. We indicate which biggest TMTOs each of those environments can achieve today.

5.2.2. Considered RTs Parameters

As the objective is to estimate feasible TMTOs across different environments, it is critical to set a constant target coverage for each scenario. This ensures fairness in the comparison. Given that this thesis mainly focuses on high coverages TMTOs, the coverage is arbitrarily set to 99.95%.

This choice facilitates the maximization of N, while ensuring that the attack phase remains manageable. For all scenarios, 4 tables are used with the quasi-maximality factor α set to 0.95, as a consequence, we have $r = 20$, where r is such that $m_0 = rm_t^{max}$.

[1] `https://top500.org/`

5.2.3. Precomputation phase

> **Reminder:** Three different types of precomputation environments are considered in this chapter: (1) `supercomputer`, which is representative of a supercomputer among the top 100 worldwide; (2) `computer`, which corresponds to a 128-core computer; and (3) `cloud`, which consists of rented computing units from on the main cloud platforms, e.g., AWS, Azure, or GCP.

For each precomputation environment, the number of cores, the number of hashes/second/core, and the available memory (RAM) are provided for `supercomputer` and `computer` in Table 5.1 and in Table 5.2 for `cloud`.

For each environment, three *scenarios* are considered. As shown in Table 5.1, for `supercomputer` and `computer`, the scenarios depend on the available time for the precomputation phase: one year, one month, or one week. For `cloud` however, the precomputation phase is bounded to one month, and as shown in Table 5.2, the environments are defined by the budget assigned to the precomputation phase according to the different scenarios: 1 000 000 USD, 100 000 USD, or 10 000 USD[2].

Table 5.1. – Precomputing `supercomputer` and `computer` environments

Environment	#Cores	Hashes/Sec/Core	RAM (TB)
`supercomputer`	86 344	25 000 000	16 182
`computer`	128	11 000 000	1

Table 5.2. – Precomputing cloud environments according to the scenarios

Scenarios	`cloud` Environments		
	#Cores	Hashes/Sec/Core	RAM (TB)
1M USD	13 055	11 000 000	256
100K USD	1 279	11 000 000	20
10K USD	128	11 000 000	2

5.2.4. Attack phase

- There are nine different attack environments, one for each scenario of the three precomputation environments.
- All attack environments are composed of a single core.
- The difference between attack environments is only the available memory.

For the attack phase, we consider environments with a single core[3]. This makes the analysis

[2]The computing characteristics shown in Table 5.2 have been obtained by simulating on these costs on AWS (cluster EC2).

[3]Note that a single core is considered to provide a reference value, but the attack phase can be easily parallelized to operate on several cores.

and description of environments a little simpler, and as discussed in Section 6.6, does not change the overal situation.

The number of hashes per second n_t, on the attack core is fixed for each environment at $n_t = 11\ 000\ 000$[4]. The idea is that the attacker usually has limited resources for the attack phase and cannot benefit from a supercomputer for this phase.

The only attribute that has a bearing on the efficiency of the attack is the size of the memory available. The different values for the attack memory M_T are chosen to correspond to realistic, practical cases and influence the parameter t. Memory available (but not necessarily used) are presented in Table 5.3.

To match practical cases, the memory available for this phase should be smaller than the memory available for the precomputation phase. The memory available for the attack therefore depends on considered scenarios and the environments used for the precomputation phase.

For each environment, scenarios depend on the time or money invested in the precomputation phase. The aim is to perform the attack as quickly as possible given the tables generated during the corresponding precomputation scenarios: one year, one month, one week of precomputation or 1M, 100K, 10K, USD invested.

Table 5.3. – Memory available for the attack according to the scenarios and environments

	Memory available for the attack (TB)			
Scenarios	`supercomputer`	`computer`	Scenarios	`cloud`
1 year	32	0.8	1M USD	16
1 month	16	0.4	100K USD	8
1 week	8	0.2	10K USD	4

5.3. Evaluation of the Maximum TMTO Size

5.3.1. Methodology

The maximum problem size N that can be addressed by a CPU-based TMTO in a given time frame or budget can be accurately evaluated from the analytical formulas provided in Chapters 3 and 4 (precomputation time) and in Chapter 2 (attack time). We provide bellow the way of evaluating each parameter.

5.3.1.1. Precomputation Phase

Reminder: When considering filtration, the following parameters are used:

- χ: Number of filters.
- c_{g_i}: Column of the i-th filter.
- n_h and n_f: Number of hashing nodes and of filtering nodes respectively.
- v_h, v_f, v_o: Respectively the hash speed, filtration speed and overhead constant.

[4]This corresponds to the number of SHA256 hashes per second measured on an AMD EPYC 7742 3.2 GHz processor.

The minimum precomputation time $P_{\min}$ in seconds is given by Equation (5.1) directly obtained from Theorem 3.4 of Chapter 3. The total precomputation time P in seconds is given by Equation (5.2) directly obtained from Chapter 4, with the same speed values ($v_o = 1.37.10^{-10}$ and $v_c = 0$) used in the first environment of Chapter 4. As in Chapter 4, for simplicity, $c_{g_0} = 0$ and $c_\chi = t$

$$P_{\min} = \frac{2N}{n_h v_h} \ln(1+r). \tag{5.1}$$

$$P = \text{Max}\left(\frac{1}{n_h v_h}\sum_{i=0}^{\chi-1} m_{c_{g_i}}(c_{g_{i+1}} - c_{g_i}); \frac{1}{v_f n_f}\sum_{i=0}^{\chi} m_{c_{g_i}}\right) + v_o \sum_{i=0}^{\chi} m_{c_{g_i}}. \tag{5.2}$$

As in Chapter 4, a single filtration core is used, thus $n_f = 1$. Values of n_h and v_h according to the environment and scenario are given in Tables 5.1 and 5.2. Values for v_f are given in Table 5.4.

For `computer` and `cloud` environments the number of hashes per second corresponds to typical hashes value of an environment similar to `computer`. The filtration speed corresponds to the one measured on an environment similar to `computer`. For `supercomputer`, environment hashes and filtrations per second have been computed from typical FLOPS numbers on this kind of environment.

Table 5.4. – Hashes and filtrations per second per core

Environment	Hashes/Sec/Core	Filtration/Sec/Core
`supercomputer`	25 000 000	51 270 585
`computer`	11 000 000	15 949 709
`cloud`	11 000 000	15 949 709

5.3.1.2. Memory Used

The memory required for the attack, M_T, is defined by Equation (5.3). This represents the memory used for the attack, excluding any form of compression or intelligent table storage. More precisely, it accounts for the storage of m_t SPs and m_t EPs on N bits for each of the ℓ tables. We have opted to estimate memory usage without considering potential improvements, as this does not significantly affect the overall conclusion and simplifies our analysis. This method of memory estimation results in a value approximately twice the size of the minimum lower bound presented in Chapter 2 for the scenarios considered in this chapter.

$$M_T = 2\ell m_t \log_2(N) \tag{5.3}$$

The maximum RAM needed for the precomputation phase M_P is given by Equation (5.4) with $m_{c_{g_1}}$ the number of surviving chains remaining after performing the first filtration. Factor 3 is explained by the fact that we consider the use of a hash table for filtration with the same load factor as in Chapter 4, i.e., $\lambda = 1.5$ and both SPs and EPs are stored.

$$M_P = 3m_{c_{g_1}} \log_2(N) \tag{5.4}$$

Table 5.5. – Evaluation of the maximum problem size

			Precomputation phase								
			supercomputer			**computer**			**cloud**		
			1 year	1 month	1 week	1 year	1 month	1 week	1M USD	100K USD	10K USD
N	Problem size		$\mathbf{2^{61}}$	$\mathbf{2^{56.83}}$	$\mathbf{2^{54.2}}$	$\mathbf{2^{50.68}}$	$\mathbf{2^{47.05}}$	$\mathbf{2^{44.9}}$	$\mathbf{2^{53}}$	$\mathbf{2^{50.13}}$	$\mathbf{2^{47.05}}$
t	Number of columns	($\times 10^3$)	7 611	788	242.9	1 200	89	17.1	75.5	37	41.5
M_P	Mem.(TB) prcmp.	Eq.(5.4)	182.23	59.82	14.1	0.98	0.96	0.98	25.05	15.29	1.99
M_T	Mem.(TB) attack	Eq.(5.3)	32.0	16.0	8.0	0.13	0.14	0.15	11.02	2.89	0.29
P	Prcmp. time (days)	Eq.(5.2)	366.05	30.06	7.73	364.35	30.0	7.11	31.23	30.16	30.04
P_{min}	Prcmp. low. bnd.(days)	Eq.(5.1)	301.13	16.73	2.7	354.34	28.62	6.45	17.21	24.01	28.4

			Attack phase								
			supercomputer			**computer**			**cloud**		
T_{RAM}	Attack time	Eq.(5.5)	9.62 d	2.47 h	14.12 m	5.74 h	1.9 m	4.15 s	1.35 m	19.6 s	24.73 s

The abbreviations “d”, “h”, “m”, and “s” respectively denote “days”, “hours”, “minutes”, and “seconds”.

5.3.1.3. Attack Phase

As only one core is used for the attack phase regardless of the environment used for the precomputation, the attack time can be well estimated by dividing result of Theorem 2.3 by the number of hashes per second n_t performed by the CPU used for the attack. Therefore, the attack time T_{RAM}, is given by Equation (5.5)

$$T_{RAM} = \frac{T}{n_t} \tag{5.5}$$

with T the number of hashes needed to perform the attack given in Theorem 2.3. As the attack should be performed by an average computer, we have chosen to use $n_t = 11\ 000\ 000$ as the number of hash per second used for the attack phase.

5.3.1.4. Choice of Parameter t

The parameter t is chosen for each time frame (year, month, or week) or according to the budget (1M, 100K, or 10K USD) in order to maximize M_T while keeping M_P and M_T under the available memory (see Tables 5.1, 5.2 and 5.3). As t impacts M_P, it influences the memory of the two phases. If t is too high, the attack phase will be too slow. On the contrary, if t is too small, the memory needed for precomputation and especially for storing tables may be too large.

5.3.2. Results

The results of the evaluation are provided in Table 5.5, where nine configurations are presented according to the environments and scenarios considered.

5.4. Discussion

5.4.1. Noteworthy observations

This section discusses the impact of memory allocation, precomputation, and attack phase bottlenecks across the environments `supercomputer`, `computer`, and `cloud`. Each environment presents unique challenges and characteristics influencing the capacity to handle larger TMTOs. For each environment, various scenarios are explored, taking the amount of memory available for precomputation and attack, the impact of N and t on memory requirements, and the limitations arising from the precomputation phase into account.

Furthermore, it is worth noting that the attack phase is considered to be executed on a single core as, to our knowledge, no literature explores the parallelization of the attack phase. However, considering the use of four tables, a reasonable assumption is that the attack time could be divided by four by executing the attack with four core and attacking with one table per core.

- Concerning `supercomputer` environment:
  - With the considerable memory capacity available for precomputation, one could potentially allocate a greater amount of memory to this phase, which might result in a reduced attack time. Despite this possibility, no further memory is allocated due to the attack phase memory already operating at its limit. This results in a scenario where there is a disproportionate allocation of available memory between the precomputation phase and the attack phase. However, given the satisfactory performance of the attack phase under the current conditions, it cannot be conclusively inferred that the memory allocated for the attack phase is a bottleneck.
  - In all scenarios of `supercomputer`, P and P_{min} are relatively distant (considerably in the one-year scenario). This is due to considering only a single filter node[5]. In this case, employing two or three filter nodes instead of only one could considerably reduce the precomputation time and hence bring P closer to P_{min}.
  - Even if P_{min} were to be considered instead of P, there would be an increase between 1 and 3 bits in the size of N in the one-month and one-week scenarios, but not even by 1 bit for the one-year scenario. Ultimately, the issue remains: the maximum allocated time has been consumed for precomputation. To increase N, it would require more time for precomputation, additional computational resources, or the introduction of a new algorithm or variant that would achieve these problem sizes with a lower P_{min}.
- Concerning `computer` environment:
  - In this environment, the memory for precomputation is constrained to 1TB. For all scenarios considered, the maximum available memory for precomputation is used, leading to an attack memory that is nearly identical across all scenarios.
  - One might question why there is a slight increase in the attack memory between the one-year and one-week scenarios, given that N is smaller in the one-week scenario and nearly the same memory is used for precomputation in both scenarios.

[5]Formulas for multiple filter nodes have not yet been introduced.

This can be explained by the number of columns t, which is lower in the one-week scenario. Since the same precomputation memory is used for a smaller N to achieve the same success probability, a smaller t can be adopted, thus increasing the memory needs for the attack.

  - It is observed that in all scenarios within this environment, P is very close to $P_{\min}$.
  - The bottleneck for this environment is also the precomputation phase, as the only way to increase the size of N would be to either extend precomputation time, use more computational resources, or introduce a variant that would have a lower $P_{\min}$ for the same N. As the attack times and the necessary memory for the attack are quite reasonable, they do not constitute a bottleneck for this environment.

- Concerning `cloud` environment:
  - In the case of the `cloud` environment, unlike the other environments, there is only one scenario in which the maximum memory for either precomputation or attack is reached, and that is the 10K USD scenario. For the two other scenarios, the maximum available memory is never reached.
  - In the 10K USD scenario, despite the use of the maximum memory for precomputation, the attack time remains low and P is close to $P_{\min}$. Therefore, the use of the maximum available memory for precomputation is not what prevents one from performing a larger TMTO.
  - In the 100K USD and 1M USD scenarios, P is a bit farther from $P_{\min}$. As in the case of the `supercomputer` environment, this is due to the fact that only one node instead of several is used for filtration.
  - For this environment, even if $P_{\min}$ would be considered instead of P, the conclusion would be the same, namely, that both the memory for precomputation and the memory for attack are sufficient. The attack times are good. What prevents the generation of larger TMTOs is still the precomputation phase. More funding would need to be invested to acquire more computational power, or a variant that covers the same space for a lower $P_{\min}$ could be employed.

5.4.2. Take Away Observations

(1) The *cost P and the theoretical lower bound $P_{\min}$ are generally close enough.* In the cases where the gap between $P_{\min}$ and P is more significant, converting some computation nodes to filtering nodes would reduce the gap significantly, as it is mainly explained by a slow filtration speed. Therefore, unless a significant algorithmic change is introduced, improving the efficiency of the precomputation phase would have negligible impact on the domain size that can be realistically attacked.

(2) For all environments considered, the largest domain sizes on which tables can be precomputed in each scenario lead to tables that are *practical* for the attack phase (i.e., reasonably small in memory, and reasonably fast to execute a search). In other words, within the context established in this Chapter on the environments, scenarios, technology, and algorithms, the *precomputation phase is the bottleneck* to using RTs on large domains. The slight overhead due to performing the attack phase on secondary memory (HDD or SSD) would thus not change this conclusion.

(3) The *memory* M_P needed to perform the precomputation is *not a bottleneck* for generating bigger TMTOs in the environments and scenarios analyzed in this chapter. Yet, even if in all scenarios the attack time is reasonable, in some instances, expanding the attack time entails increasing the precomputation memory rather than the attack one.

5.4.3. Conclusion

Our main observation from Table 5.5 is that the precomputation cost P is the current bottleneck of TMTO. Indeed, the attack time remains reasonably low, even for the biggest space sizes considered. The memory needed for the precomputation and the attack is affordable (for the corresponding entity that performs it). Therefore, it is the precomputation time P, and more precisely, the computing power of the adversary (number of computation nodes and number of hashes per second) dedicated to the precomputation that limits an increase in N.

Using the state-of-the-art precomputation algorithm, there is little room for improvements on the precomputation cost P. Indeed, P is very close to the theoretical lower bound $P_{\min}$. New algorithms that have a lower bound, should thus be introduced.

Going further may require forgoing the CPU technology for a more efficient one, which could currently be GPU or FPGA. While these technologies have constraints in how they are put to use, they do operate several orders of magnitude faster than typical CPUs. Several articles, e.g., [KSH+15, LLH15], treat this problem and websites propose implementations or programs[6] to purchase or generate RTs on GPU. Other contributions, e.g., [SRQL02, MBPV06], focus on FPGA-based TMTOs. However, these papers address the problem with relatively small domains N (in the 2^{40} to 2^{50} range), deprecated one-way functions, or older GPU/FPGA models. Furthermore, they do not use recent improvements in TMTOs, e.g. filtration. Therefore, it is not possible to compare these results obtained on GPU/FGPA with the one from this chapter. A deeper look into efficient GPU- or FPGA-based precomputation represents interesting future work that could evaluate these technologies capabilities to deal with larger TMTOs.

In summary, this work helps quantify the vulnerable spaces to TMTOs performed on CPUs. In addition, although the precomputation phase has not been as widely studied as the attack phase, it seems to be the main bottleneck preventing larger TMTO-based attacks nowadays.

The precomputation phase being a bottleneck in our observations may likewise be challenged by such changes. Indeed, the precomputation cost is by nature linear in N, whereas the attack phase is by nature quadratic in N. A situation where the attack phase is the bottleneck is therefore not far-fetched, depending on the development of technologies and research in this field.

[6] For instance, https://www.cryptohaze.com/ offers a GPU-based rainbow cracker, and http://project-rainbowcrack.com/ has implementations of RTs on GPU.

Descending Stepped Rainbow Tables

6

What is the height of irony for a DSRT researcher?
To miss a step

6.1. Motivation

The preceding chapters have highlighted the importance of considering all aspects of Time-Memory Trade-Offs (TMTOs) rather than solely focusing on the time or memory required for the attack. In this chapter, we introduce a new variant of Rainbow Table (RT) called the *Descending Stepped Rainbow Table* (DSRT) that allows users to choose the best trade-off between the memory required for storing the table, the coverage, the precomputation time, and the attack time.

To compare the DSRT variant with the vanilla RT, we compared them for the same coverage and same memory. Thus, the attack time and the precomputation time are the two remaining characteristics to consider. Our conclusion is that the DSRT variant performs better than the RT in both attack time and precomputation time.

The key observation leading to the DSRT construction is that during the cleaning process of RT, most computed chains are discarded due to merges. To reduce the precomputation time, our idea is to keep some of these merged chains instead of discarding them, and thus keep chains that are shorter than the regular ones. The merged parts are removed, leading to a shape of 'stair' and thus to the so-called DSRT.

Takeaway:

In this chapter, we introduce a new variant of Rainbow Table (RT) called the Descending Stepped Rainbow Table (DSRT).

- The DSRT principle is to keep some well-chosen merged chains after cutting them, rather than discarding them.
- DSRT performs better than vanilla RT in both attack and precomputation time.
- To achieve a targeted success probability and memory, a large number of DSRT configurations are possible, allowing various trade-offs between precomputation time and attack time.

6.2. DSRT Overview

The DSRT algorithm involves keeping some specific additional chains in the matrix during the precomputation phase. In given columns of the matrix called *steps*, a cleaning is performed, and well chosen chains are kept with a smaller length. The number of steps in a DSRT is denoted as τ. For the sake of clarity, we first introduce a DSRT with one step before generalizing to DSRTs with τ steps. We first introduce how the precomputation phase is performed, then we present how performing the attack phase using the DSRT. The analysis of the DSRT performance with τ steps is then proposed in Section 6.3.

6.2.1. DSRT with One Step

6.2.1.1. Precomputation phase

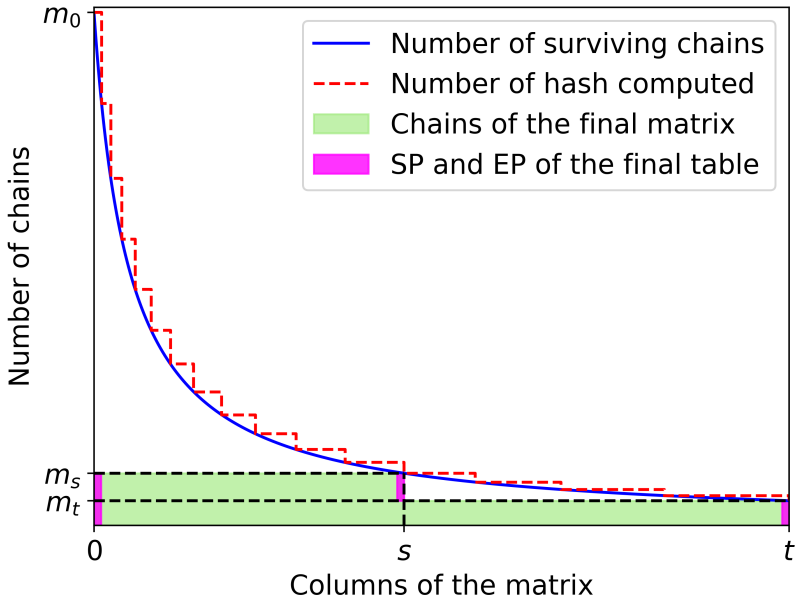


Figure 6.1. – Construction of DSRT matrix with single step.

> **Reminder:** *Surviving chains* in a given column c are chains remaining after a cleaning in column c. We denote by m_c the number of surviving chains in a given column c.

The precomputation phase of DSRT begins by building m_0 chains until reaching a first filter. A filtration is then performed and the computation of chains continues with the remaining surviving chains. Once the column s (called the *step*) is reached, a filtration is performed and remaining elements are temporarily stored. At each subsequent filtration, when chains merge, one is kept for further extension while the remaining ones are cut short in column s. The chains that are cut short have length s and their elements present in column s (that have been temporarily stored) are kept as final EP. Chains remaining after the final filtration in column t have length t and their final EP is their value in column t.

Figure 6.1 illustrates the construction of a DSRT. The solid blue curve is m_c. The area under the dashed red curve represents the amount of hash operations needed to generate the table using the filtration method (each landing of the curve corresponds to the application of a filter). The DSRT matrix obtained at the end of the precomputation phase is colored in green. The SPs and EPs kept as the final table are colored in pink.

6.2.1.2. Attack Phase

Reminder: The *attack chain* refers to the chain built from y. The attack chain starting in column c and ending in column t is thus the chain finishing by the value $f_t(f_{t-1}(\dots f_{c+1}(R_c(y))))$.

Reminder: The three events occurring during the attack phase are:

True alarm: The attack chain matches with an EP and the starting element of the attack chain is the searched element. The attack is successful.

False alarm: The attack chain matches with an EP but the starting element of the attack chain is not the searched element. The attack continues by searching in other columns.

No alarm: The attack chain does not match with any EP. The attack continues by searching in other columns.

Reminder: The process of assuming that the searched element is in a given column c of the matrix and performing computations to verify this hypothesis is referred to as a *search* in column c.

In a vanilla RT, all columns have the same probability of raising a true alarm, but the further to the right the column is, the lower is the cost of searching in that column (in other words, the cost of the search in that column is decreasing). Therefore, the most effective searching order is from the last column t in decreasing order.

For a 1-step DSRT, the probability of finding the searched element depends on the column index, because not all columns contain the same number of elements: columns 0 to s contain m_s elements and columns $s+1$ to t contain m_t elements.

Moreover, when an alarm (true or false) occurs in a column c with $c < s$, it can either be detected in column s or in column t: it is detected in column s if a match occurs with one of the chains of length s, otherwise the alarm is detected in column t. The number of cryptographic operations needed to raise an alarm is consequently not the same in all columns. Additionally, the number of cryptographic operations to check whether an alarm is true or false depends on whether the alarm was raised in column s or in column t.

Consequently, with a 1-step DSRT, instead of performing a search from column $t-1$ to column 0, the search is performed in an order that minimizes the total expected time. This corresponds to searching in the non-explored column with the highest coverage over average cost ratio[1].

[1] Another example of non-monotonic search order is discussed in the analysis of heterogeneous RTs [AC17]. Optimality of the "coverage over average cost ratio" as a decision metric is also discussed there.

The procedure to perform the attack phase with 1-step DSRT is provided in Algo. 1 in Appendix A.1. The definition of the function μ that appears in Algo. 1 is provided in Definition 6.3. In a nutshell, this function returns the index of the most promising column for the forthcoming search.

6.2.2. DSRT with τ Steps

The concept of DSRTs can be naturally generalized to τ steps with $\tau > 1$.To construct the matrix, the precomputation begins with m_0 elements and is performed until reaching the first step s_1. The m_{s_1} last elements of the surviving chains in column s_1 are temporarily stored, and the precomputation continues. At each subsequent filtration, when chains merge, one is kept while the other ones are cut and stored with a length of s_1. When step s_2 is reached, a filtration is performed. For each merge, one instance of merged chains is kept, the other are cut and stored with a length of s_1. The m_{s_2} last elements of the m_{s_2} surviving chains are temporarily stored, and the precomputation continues. The same process as with step s_1 is repeated, but now the merged chains are kept with a length of s_2.

The precomputation continues until reaching column t, where a final filtration is performed to obtain a clean DSRT.

Figure 6.2 illustrates the resulting final structure.

> ⚠ In what follows, the steps are denoted by s_i with $0 < i \leqslant \tau$, where $s_i < s_j$ for $i < j$. By convention, s_0 corresponds to column 0 ($s_0 = 0$), and $s_{\tau+1}$ corresponds to column t ($s_{\tau+1} = t$).

For each step s_i, with $0 < i < \tau$, $m_{s_i} - m_{s_{i+1}}$ chains of length s_i are stored, and m_t chains of length t are also stored. Chains of length t form the *main table*, while chains of length s_i are chains of step s_i.

In total, m_{s_1} chains form the final matrix, and thus m_{s_1} EPs and their corresponding SPs are stored.

The attack phase using τ steps is similar to Algo.1. Algo.2, provided in Appendix A.2, presents the attack phase of the generalized DSRTs.

6.3. Analysis of DSRTs

6.3.1. Preliminaries

To facilitate the characterization and visualization of DSRTs, this section provides notations and notions used when considering DSRTs. The introduced notions are then used throughout the chapter.

Definition 6.1. *The value $k(c)$ is the index of the leftmost step that is to the right of column c, i.e., $s_{k(c)-1} \leqslant c < s_{k(c)}$.*

Definition 6.1 is useful to express some quantities concisely, in particular: there are $m_{s_{k(c)}}$ surviving chains at the step immediately to the right of column c. Similarly, there are $m_{s_{k(c)}} - m_{s_{k(c)+1}}$ chains of length $s_{k(c)}$ there.

Where appropriate, $s_{\tau+1}$ refers to column t and s_0 refers to column 0. This convention allows for some results to be presented more clearly and concisely.

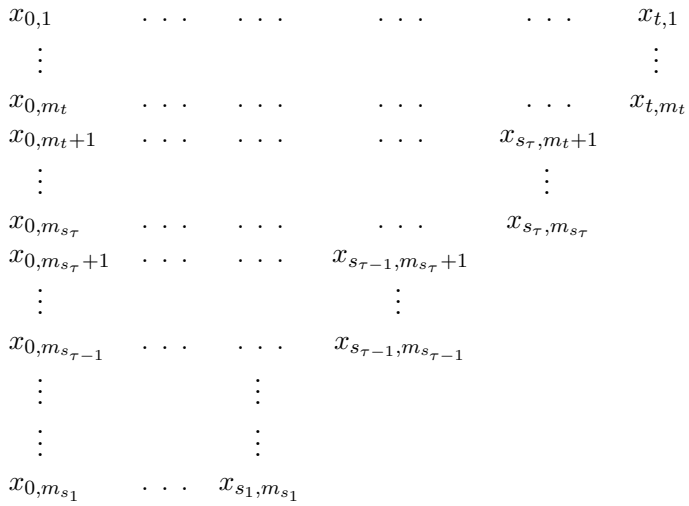


Figure 6.2. – DSRT matrix with τ steps.

The ratio ρ is also introduced in Definition 6.2 for the purpose of making the presentation of the subsequent results more concise. The value $\rho_{i,j}$ represents the proportion of chains with a length s_i in column j. For instance, let us consider a DSRT with two steps s_1 and s_2 with $s_1 < s_2 < t$. In column c, with $c < s_1 < s_2$, chains can be of length s_1, s_2, or t. Values $\rho_{1,k(c)}$, $\rho_{2,k(c)}$, $\rho_{3,k(c)}$ give the ratio of chains in column c that have a length respectively of s_1, s_2 and s_3 (with $s_3 = s_{\tau+1} = t$). Definition 6.2 generalizes this concept to all steps and columns.

Definition 6.2. *Given a column c and a step s_i of a DSRT, the proportion of chains with length s_i in column c is defined by $\rho_{i,k(c)}$ where $\rho_{i,j}$ is defined as:*

$$\rho_{i,j} = \begin{cases} \frac{m_{s_i} - m_{s_{i+1}}}{m_{s_j}} & i \leqslant \tau \\ \frac{m_t}{m_{s_j}} & i = \tau + 1 \end{cases}$$

6.3.2. Success Probability

The success probability of a DSRT is computed similarly to that of RTs, with the difference that some chains are shorter, which must be taken into account when computing the success probability.

The success probability for a single DSRT is given in Theorem 6.1.

Theorem 6.1. *Given τ steps denoted by s_i with $0 < i \leqslant \tau$, $s_0 = 0$ and $s_{\tau+1} = t$, and considering m_{s_i} the number of surviving chains in column s_i, the success probability p of a single clean DSRT is:*

$$p = 1 - \prod_{i=1}^{\tau+1} \left(1 - \frac{m_{s_i}}{N}\right)^{s_i - s_{i-1}}.$$

Proof. Each column c covers $m_{s_{k(c)}}$ different elements. Following a similar argument as in Chapter 2, we have:

$$p = 1 - \prod_{c=1}^{t} \left(1 - \frac{m_{s_{k(c)}}}{N}\right).$$

Given that $s_{k(c)}$ (and therefore $m_{s_{k(c)}}$) are equal for all c between two steps, we can group these factors together, resulting in the formula provided in Theorem 6.1. □

The success probability when using ℓ DSRTs, denoted by p_ℓ, is then directly obtained from Theorem 6.1, and is given by Corollary 6.2

Corollary 6.2. *Given τ steps denoted by s_i, with $0 < i \leqslant \tau$, $s_0 = 0$ and $s_{\tau+1} = t$, and considering m_{s_i} the number of surviving chains in column s_i, the success probability p_ℓ of ℓ clean DSRTs is:*

$$p_\ell = 1 - (1-p)^\ell .$$

6.3.3. Precomputation Time

Generating DSRTs costs the same number of hash operations as in the case of vanilla RTs (see Chapter 2). During the precomputation phase, whether at each step intermediary elements are saved or not has no bearing on the number of hash computations (only the m_0 SPs considered at the beginning of generation and the number and positions of filters matter). As shown in Section 6.5.4, the overhead due to filtration or the storage of intermediary elements is insignificant.

However, a distinguishing feature of DSRTs is that for a given targeted memory and success probability, m_0 is typically much smaller than with vanilla tables, thus allowing to significantly reduce the precomputation time.

The minimum precomputation time of a RT has been established and demonstrated in Chapter 2 and is presented in equation (6.1).

$$P_{\min} = \sum_{i=0}^{t-1} m_i \approx 2N \ln(1+r). \tag{6.1}$$

In this chapter, we use $P = 1.13 \times P_{\min}$. Considering $1.13 \times P_{\min}$ instead of $P_{\min}$ is arbitrary but comes with significant justification from our earlier findings. As discussed in Chapter 4, it has been shown that one can closely approximate the minimum precomputation time using the filtration method. We also observed that by using between eleven and thirty one filters, it is feasible to get close enough to the theoretical bound. In the first environment discussed in Chapter 4, the precomputation time for a vanilla RT was found to be approximately $1.13 \times P_{\min}$ when using twenty filters. Given that the purpose of this chapter is to provide insights into what could be achievable in real-world applications, we opted for $1.13 \times P_{\min}$, assuming that it would offer a good approximation[2]. In addition, this factor closely aligns with our experimental results as discussed in Section 6.4.

6.3.4. Attack Time

In contrast to RTs, the DSRT attack is not carried out by searching linearly from the right to the left in the matrix. The search in each column has an associated average cost, and a given probability that the search succeeds. This means that, while the cost does increase linearly with the size of the attack chain (just as it does in RTs), the probability of successful search does not remain constant (contrarily to RTs). The average attack time thus corresponds to

[2]Nevertheless results of this chapter can also be replicated using $P = P_{\min}$ to obtain values that can be easily transposed to any environment.

the sum of the cost of the search in all columns, visited in optimal order, and weighted by the probability that the search stops there.

As in the vanilla case, a search in a given column either leads to a *true alarm* (successful search), a *false alarm* or to *no alarm*. An analysis of these events probabilities of occurrence and the associated costs is the focus of Sections 6.3.4.1 to 6.3.4.6. A useful intermediate result is the probability that no merge occurs between two given columns. This is one of the fundamental results in [Oec03] and is generalized to any two columns c and c' in Lemma 6.3.

Lemma 6.3. *Given two columns c and c' with $c < c' \leqslant t$, the probability that the attack chain does not merge with any chain of the rainbow matrix by c', given it had not merged in or before c, is:*

$$p_{nomrg}(c, c') = \prod_{i=c+1}^{c'} \left(1 - \frac{m_i}{N}\right). \tag{6.2}$$

6.3.4.1. Probability of True Alarm

A true alarm occurs when the first element of an attack chain appears in the corresponding column of the DSRT matrix.

> **Reminder:** $s_{k(c)}$ is the leftmost step that is to the right of c. In column c, there is $m_{s_{k(c)}}$ elements in the DSRT matrix.

Proposition 6.4. *The probability that a true alarm occurs when starting the attack chain in the column c is:*

$$p_{find}(c) = \frac{m_{s_{k(c)}}}{N}.$$

Proof. There are $m_{s_{k(c)}}$ different elements in column c of the DSRT matrix. Given that elements are uniformly distributed and that the number of elements in the searched space is N, the probability that x^* is an element of a column c is the stated quantity. □

6.3.4.2. Probability of False Alarm

In DSRTs, a false alarm occurs when the attack chain at column c merges with a chain of the matrix. This is similar to vanilla RTs, with the difference that this merge can be observed at any of the steps $s_i > c$, including $s_{\tau+1} = t$, but not exclusively. There is a distinction to be made whether the merge occurs before or after $s_{k(c)}$. The two cases are analyzed in Propositions 6.5 and 6.6.

Proposition 6.5. *The probability to raise a false alarm due to a merge between columns c and $s_{k(c)}$ is*

$$p_{fa\text{-}pre}(c) = 1 - p_{find}(c) - p_{nomrg}(c, s_{k(c)}).$$

Proof. Straightforward, as these events (no alarm, true alarm, false alarm) are mutually exclusive. □

Proposition 6.6. *The probability to raise a false alarm due to a merge between columns s_i an s_j, with $c \leqslant s_{k(c)} < s_i < s_j$, is:*

$$p_{fa\text{-}post}(c, s_i, s_j) = p_{nomrg}(c, s_i) - p_{nomrg}(c, s_j).$$

Proof. A false alarm due to a merge between columns s_i and s_j means that no merge occurs between c and s_i but one occurs between s_i and s_j. Denoting by E_1 and E_2, respectively, the events "no merge occurs between c and s_i" and "a merge occurs between s_i and s_j", we have:

$$\begin{aligned} p_{\text{fa-post}}(c, s_i, s_j) &= \Pr(E_1 \wedge E_2) \\ &= \Pr(E_1) \times \Pr(E_2 \mid E_1) \\ &= p_{\text{nomrg}}(c, s_i)\,(1 - p_{\text{nomrg}}(s_i, s_j)) \\ &= p_{\text{nomrg}}(c, s_i) - p_{\text{nomrg}}(c, s_i) p_{\text{nomrg}}(s_i, s_j) \\ &= p_{\text{nomrg}}(c, s_i) - p_{\text{nomrg}}(c, s_j) \end{aligned}$$

The last equality stems from the nature of Lemma 6.2. □

6.3.4.3. Probability of No Alarm

When no match occurs between the attack chain and the EPs of a table, no alarm is raised.

Proposition 6.7. *The probability that no alarm occurs between a column c and column t is*

$$p_{noalarm}(c) = p_{nomrg}(c, t).$$

Proof. For no alarm to happen, the attack chain must not merge with the DSRT matrix at any point between its start in column c and its end in column t. □

6.3.4.4. Cost of Alarms

The cost of an alarm, i.e., the number of cryptographic operations performed when an alarm occurs, is the same for true and false alarms. In both cases, an entire chain has to be rebuilt: firstly from column c, where the search is performed, to a step s_i or to column t (depending on where the merge is detected) and then from column 0 to column $c-1$. Therefore, the cost of an alarm does not depend on its nature (true or false alarm) but only on the column in which it is detected. The cost of an alarm is given in Theorem 6.8 using Definition 6.2.

> **Reminder:** $\rho_{i,k(c)}$ is the ratio of chains in column $s_{k(c)}$ that have a length i. In column $s_{k(c)}$, the possible chains length are $\{s_{k(c)}, s_{k(c)+1}, ..., s_{\tau+1}\}$ i.e., the length of chains ending at step $s_{k(c)}$ and all steps that are to the right of it.

Proposition 6.8. *Given a search performed in a column c and $k(c)$ the index of the leftmost step that is to the right of column c, the number of hash operations necessary to rule out a false alarm is:*

$$\sum_{i=k(c)}^{\tau+1} \rho_{i,k(c)} s_i.$$

Proof. $\rho_{i,k(c)}$ is the proportion of chains with length s_i in column c. It corresponds to the number of chains with the length s_i in column s_i divided by the total number of chains in column s_i. Given c and $k(c)$ such as $s_{k(c)-1} \leqslant c < s_{k(c)}$, alarms are not necessarily detected in step $s_{k(c)}$ but can be detected in each step $s_j > s_{k(c)}$. If a merge occurs in column j with $c < j \leqslant s_i$, the probability that this merge is detected in step s_i is $\rho_{i,k(c)}$.

If an alarm is detected in s_i, a chain of length s_i has to be rebuilt, which costs s_i hash operations. Therefore, the average cost of an alarm in a column c is the sum of the probability to detect the alarm in each step s_i with $c \leqslant s_i$, multiplied by the number of hash operations needed to rule it out, depending on which step the alarm is detected. □

6.3.4.5. Cost of No Alarm

The cost of no alarm in a given column c is $(t - c)$. If no alarm occurs in any step, no chain will be rebuilt, therefore only $(t - c)$ hash operations will be computed.

6.3.4.6. Total Time of a Search in a Given Column

The number of operations needed to perform a search in a column c is given by Theorem 6.9.

Theorem 6.9. *For a given column c and the index $k(c)$, the average number of cryptographic operations C_c needed to perform a search is:*
for $s_\tau < c \leqslant t$:

$$C_c = t - cp_{noalarm}(c),$$

for $c \leqslant s_\tau$:

$$\begin{aligned} C_c = \left(1 - p_{nomrg}(c, s_{k(c)})\right) & \sum_{i=k(c)}^{\tau+1} \rho_{i,k(c)} s_i \\ + & \sum_{j=k(c)}^{\tau} p_{fa\text{-}post}(c, s_j, s_{j+1}) \sum_{i=j+1}^{\tau+1} \rho_{i,j+1} s_i \\ + & \, (t - c)\, p_{noalarm}(c). \end{aligned}$$

Proof. To obtain the total cost of the search, the probability of the three events (true alarm, false alarm, no alarm) has to be multiplied by their cost and summed up together.
Case 1: $s_\tau < c \leqslant t$.

(a) The probability of a true alarm is obtained from Proposition 6.4 and is $\frac{m_t}{N}$, its cost is obtained from Proposition 6.8 and is $\rho_{\tau+1,k(c)} s_{\tau+1} = \rho_{\tau+1,\tau+1} s_{\tau+1} = t$.

(b) The probability of false alarm is given by Proposition 6.5 and is: $1 - \frac{m_t}{N} - p_{\text{noalarm}}(c)$. The cost is the same as for true alarms.

(c) The probability of no alarm is given by Proposition 6.7 and its cost is $t - c$ (see Section 6.3.4.5).

We thus have:

$$\begin{aligned} C_c &= t\frac{m_t}{N} + t\left(1 - \frac{m_t}{N} - p_{\text{noalarm}}(c)\right) + (t - c)p_{\text{noalarm}}(c) \\ &= t - cp_{\text{noalarm}}(c). \end{aligned}$$

Case 2: $c \leqslant s_\tau$.

(a) The probability of a true alarm is $\frac{m_{s_{k(c)}}}{N}$, and its cost is $\sum_{i=k(c)}^{\tau+1} \rho_{i,k(c)} s_i$.

(b) The probability of a false alarm due to a merge between c and $s_{k(c)}$ is given by Proposition 6.5 and is $1 - \frac{m_{s_{k(c)}}}{N} - p_{\text{nomrg}}(c, k(c))$. Its cost is the same as for a true alarm.

(b') The probability of a false alarm due to a merge between s_j and s_{j+1} is $p_{\text{fa-post}}(c, s_j, s_{j+1})$ with $c \leqslant s_j < s_{j+1}$ and is given by Proposition 6.6. Therefore the probability of a false alarm due to a merge between $s_{k(c)}$ and s_τ is $\sum_{j=k(c)}^{\tau} p_{\text{fa-post}}(c, s_j, s_{j+1})$. Its cost, given by Proposition 6.8, depends on step s_j in which the false alarm is detected and is $\sum_{i=j+1}^{\tau+1} \rho_{i,j+1} s_i$.

(c) The probability of no alarm is given by Proposition 6.7 and its cost is $t - c$ (see Section 6.3.4.5).

We thus have:

$$
\begin{aligned}
C_c = & \frac{m_{s_{k(c)}}}{N} \sum_{i=k(c)}^{\tau+1} \rho_{i,k(c)} s_i \\
& + \left(1 - \frac{m_{s_{k(c)}}}{N} - p_{\text{nomrg}}(c, s_{k(c)})\right) \sum_{i=k(c)}^{\tau+1} \rho_{i,k(c)} s_i \\
& + \sum_{j=k(c)}^{\tau} p_{\text{fa-post}}(c, s_j, s_{j+1}) \sum_{i=j+1}^{\tau+1} \rho_{i,j+1} s_i \\
& + (t-c)\, p_{\text{noalarm}}(c).
\end{aligned}
$$

The conclusion follows from the $\frac{m_{s_{k(c)}}}{N} \sum_{i=k(c)}^{\tau+1} \rho_{i,k(c)} s_i$ terms canceling out. □

6.3.4.7. Average Attack Time

> ⚠ Metric μ (corresponding to coverage over average cost ratio) is used to determine the most efficient column in which to perform each iteration of the search.

Using μ as defined in Definition 6.3, vector $\boldsymbol{v}$ is defined in Definition 6.4. Vector $\boldsymbol{v}$ is composed of all columns of a DSRT from the most to least efficient.

Definition 6.3. *Given N and a column c with $0 \leqslant c \leqslant t$:*

$$\mu(c) = \frac{m_{s_{k(c)}}}{N C_c}.$$

Definition 6.4. *Given a DSRT with t columns, the vector $\boldsymbol{v}$ with $\boldsymbol{v} = [v_1, v_2, \ldots, v_t]$ is obtained by sorting the columns $[1, \ldots, t]$ in decreasing order of $\mu(c)$.*

The average attack time i.e., the average number of hash operations needed to perform an attack with a set of ℓ DSRTs, is given by Theorem 6.10.

Theorem 6.10. *Given N, ℓ DSRTs with τ steps, and considering its vector $\boldsymbol{v} = [v_1, v_2, \ldots v_t]$ ordering the columns of tables, the average number of hash operations T required to perform an attack is:*

$$T = \ell \sum_{c=1}^{t} \left(\frac{m_{v_c}}{N} \prod_{i=1}^{c-1} \left(1 - \frac{m_{v_i}}{N}\right) \sum_{j=1}^{c} C_{v_{t-j+1}} \right) + \ell \prod_{i=1}^{t} \left(1 - \frac{m_{v_i}}{N}\right) \sum_{c=1}^{t} C_{v_c}.$$

with m_{v_c} the number of surviving chains in column c, and with $s_{v_{c-1}} \leqslant c < s_{v_c}$.

Proof. This expression is a generalization of Chapter 2. The proof is constructed using an approach similar to the one used in [Oec03]. T is obtained by adding on the one hand, the success probability of the attack using ℓ tables, multiplied by its average cost, and on the other hand, the failure probability of the attack using ℓ tables, multiplied by the cost of a failed search.

The first term is obtained by multiplying, for each column c, the probability of a true alarm in the column with the probability of no true alarm in all earlier iterations:

$$\frac{m_{v_c}}{N}\prod_{i=1}^{c-1}\left(1-\frac{m_{v_i}}{N}\right).$$

This is multiplied by the cost of all searches performed until reaching this column: $\sum_{j=1}^{c} C_{v_{t-j+1}}$.

The second term is obtained by multiplying the failure probability using ℓ tables, namely $\ell\prod_{i=1}^{t}\left(1-\frac{m_{v_i}}{N}\right)$, with the cost of performing a search in all columns of a table, namely $\sum_{c=1}^{t} C_{v_{t-j+1}}$. □

Takeaway:

- As for RT, the overall attack time (or cost) using a DSRT is the sum of the search cost in each column of the table, weighted by the probability of performing a search in that column.
- False alarms can be detected in steps, which saves the cost of building the attack chain until column t.
- In contrast to RT, the search in a DSRT is not linear since it may be more beneficial to search in chains ending at a step to the left rather than continuing to search chains ending at the same step.

6.3.5. Memory Used

Formulas to evaluate the memory needed to store RTs are presented in [AC13]. In this section, we adapt them to DSRTs.

6.3.5.1. Rationale

When using DSRTs, instead of considering one set of elements per table, EPs and SPs are grouped according to their length. For the same number of chains and the same number of tables, storage of DSRTs EPs thus takes more memory than RTs. Indeed, the EPs to be stored are divided into independent collections as opposed to a single collection, resulting in less efficient compression.

On the other hand, in order to generate the same number of chains and the same number of tables, much fewer SPs need to be considered at the beginning of precomputation. Thus for a given number of chains (regardless of their size), m_0 is smaller for DSRTs than for vanilla tables. As the memory needed for storing SPs depends on m_0, the storage of SPs ends up using less memory for DSRTs than for RTs.

Overall, the memory for DSRTs compared to vanilla tables depends on specific parameters (see Section 6.5.5.3 for a discussion). Nevertheless, in this manuscript we always choose parameters to compare with RTs for the same coverage and memory.

6.3.5.2. Analysis

For each step s_i, there are $m_{s_i} - m_{s_{i+1}}$ elements to be stored. These are in addition to the m_t chains of length t.

The memory needed to store the table EPs is given by Equation (6.3), adapted for DSRTs from [AC13].

$$M_{ep}^{DSRT} = \ell \left(\log_2 \binom{N}{m_t} + \sum_{i=1}^{\tau} \log_2 \binom{N}{m_{s_i} - m_{s_{i+1}}} \right). \tag{6.3}$$

As for SPs, m_{s_1} EPs are stored for each table instead of m_t. The storage of individual SPs is the same as in the vanilla case. The memory needed to store DSRTs SPs, M_{sp}^{DSRT}, is given in Equation (6.4):

$$M_{sp}^{DSRT} = \ell m_{s_1} \lceil \log_2(m_0) \rceil. \tag{6.4}$$

The total memory used to store ℓ DSRTs M^{DSRT} is simply $M^{DSRT} = M_{sp}^{DSRT} + M_{ep}^{DSRT}$.

6.4. Experiments

This section illustrates the theoretical results obtained in Section 6.3 with practical experiments. DSRTs with between one to five steps have been generated, and their precomputation time, success probability, and attack time have been compared to the respective theoretical results. In all cases, when a sufficient number of experiments has been performed, the relative difference between theoretical and practical results is below 1%.

It is worth noting that the precomputation time (number of hash operations) is the same for both RTs and DSRTs when considering equal parameters, namely number of SPs, number and positions of filters, and the value of t. The comparison that follows consequently focuses on attack time and success probability only.

In what follows, for the sake of clarity, we call *configuration* a list of parameters describing a set of RTs: maximality factor of the tables, number of columns, and number of tables, denoted by α, t, and ℓ respectively. When considering DSRTs, the configuration also contains the number of steps, denoted by τ, and their positions $s_1, \ldots, s_\tau$.

6.4.1. Availability

The code used to generate DSRTs and perform attacks is publicly available on GitHub[3]. It provides a python script that launches DSRTs generation for parameters (i.e., N, ℓ, α, t and step positions) given in the script header, and performs attacks using the generated tables. The number of attacks to perform has to be specified. Log files are created for tracing success probability, precomputation time and attack time.

6.4.2. Environment

Experiments are conducted on a computer hosting two AMD EPYC 7H12 processors composed of 64 physical cores and 64 virtual cores each, for a total of 128 physical cores and 128 virtual cores. We used up to 128 of the physical cores in our experiments.

[3] https://github.com/DianeLeblancAlbarel/Stairway-To-Rainbow

The precomputation phase has been distributed as proposed in Chapter 4, using the filtration method adapted to DSRTs. Both the precomputation and attack phases were written in the C language, using the SHA256 function of OpenSSL for hashing. The Open MPI library is used to distribute the precomputation phase.

6.4.3. Success Probability

Theorem 6.1 is used to choose DSRT configurations that reach a target success probability.

Experiments with different configurations were performed, containing up to five steps positioned between columns $0.3t$ and t. The positions of the steps are not critical in these experiments, given that there is no "optimal" configuration. Indeed, the position of the steps defines a trade-off between precomputation time and the attack time. Nevertheless, configurations with widely different step positions were tested to cover various cases: concentrated on the right, concentrated on the left, uniformly distributed, etc. Success probabilities were tested from 85% to 99.5%. All experiments lead to the same conclusion. For clarity, only the case using two tables and a 95% success probability is presented here. The probability of 95% has been chosen arbitrarily but same results are obtained with other target success probabilities.

Figure 6.3 depicts the success probabilities when considering experiments based on two DSRTs. The blue horizontal line in the figure is the theoretical success probability, chosen to be 95%.

A test consists in randomly choosing one of the N elements of the space A, hashing it, and trying to find its preimage using the generated DSRTs. In Figure 6.3, twenty five configurations are tested. A batch of 10 000 attacks was performed for each configuration. Each point of the figure represents a different tested configuration. The success probability obtained corresponds to the proportion of the attacks in the batch that succeeded.

For all configurations, the obtained success probabilities are all between 94.7% and 95.4%.

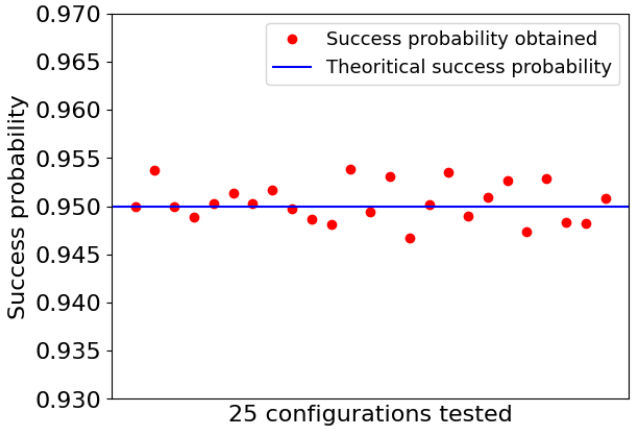


Figure 6.3. – Experimental success probability according to the configuration used, with theoretical success probability equal to 95%.

6.4.4. Attack Time

To test the compliance of the attack time with the model given in Section 6.3, various configurations have been tested. For each configuration tested, 25 batches of 20 000 attacks were performed, i.e., 500 000 attacks per configuration. For brevity and clarity of presentation, we display a random selection of 25 among the large sample of tested configurations. They were made for the same coverage and memory, though experiments performed for different memory and coverage lead to the same conclusion. The twenty five configurations presented here use five steps, two tables, and a 95% success probability. The differences between each configuration tested are various values of the parameters α and t, and steps positions. The target probability and memory are the same. The memory M and N have been arbitrarily chosen to correspond to reasonable cases, namely $M = 63.9$ GB and $N = 2^{42}$. The position of the steps varies between $0.3t$ and t (see Section 6.5.3 for explanations about this bound).

Figure 6.4 presents the results obtained. For each tested configuration, the average attack time of the configuration corresponds to the average number of hashes performed over the 500 000 attacks. Each point represents the ratio between the average attack time of each configuration and the theoretical counterpart given by Theorem 6.10.

The black line represents for each point the first and third quartile obtained on each of the 25 batches of 20 000 attacks. The black dashes represent quartiles 0 and 4.

The results are well distributed around 1.0. The large majority of experiments (between first and third quartile) are within ±1.5%. Figure 6.4 leads to conclude that experimental results fit the theory very well. It is worth noting that significant variations in the attack time are intrinsic to RTs and are noticeable for both DSRT and classical RTs.

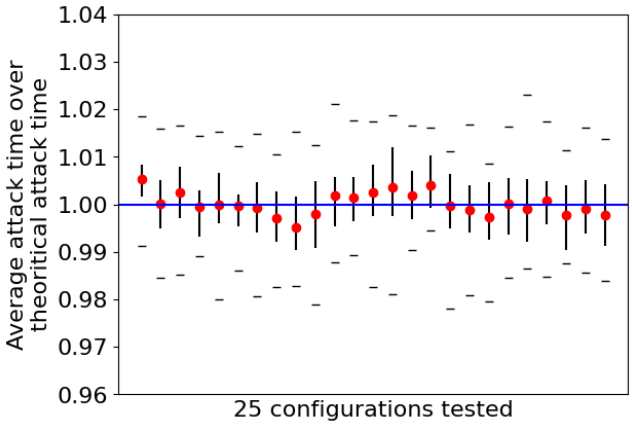


Figure 6.4. – Average number of hash for an attack over theoretical number of hash for different configurations.

6.5. Evaluation

This section compares the performance of RTs and DSRTs. The comparison methodology is first introduced, then the configurations used for the comparison are established, the results are presented, and the compatibility of the DSRTs with existing improvements is finally discussed.

6.5.1. Comparison Methodology

> **Reminder:** A *configuration* is a list of parameters describing a set of RTs or a set of DSRTs. For RTs the parameters are: maximality factor of the tables (α), number of columns (t), and number of tables (ℓ). When considering DSRTs, the configuration also contains the number of steps, denoted τ, and their positions $s_1, \ldots, s_\tau$.

Defining configurations is needed prior to generating a set of tables. As concluded in Chapter 5, the precomputation time is usually the bottleneck that prevents the use of large-scale TMTOs. Nonetheless, the attack time is of primary importance as well. The comparison thus focuses on the trade-off between these two constraints.

Various trade-off curves are compared. For each comparison, a target success probability and an available memory for the attack phase are set. These parameters being fixed, the *best configuration* refers to the configuration for which there is no other configuration having *both* better precomputation time and better attack time, whether we consider RTs or DSRTs. These configurations define the *Pareto frontier* for each comparison.

We describe the costs (both precomputation and attack phases) in terms of number of hash operations. This allows for better accuracy, and can be transposed to any computing environment. A good approximation of the time in seconds for the precomputation or attack phase is indeed simply the number of hash operations to be performed divided by the number of hash operations computable per second on the chosen environment. As shown in Chapter 4, this approximation may vary slightly for the precomputation phase depending on the distributed environment used. For the attack phase, this approximation is very accurate.

6.5.2. Configurations

This section presents how the configurations for the comparisons were determined.

6.5.2.1. Rainbow Tables

As previously explained in Section 6.4, three parameters, namely α, t, and ℓ are sufficient to fully define the characteristics of a RT. The success probability of a single table and the memory available for the attack phase, given respectively by Equations (2.10) and 2.13, are fixed by α and t. Only the number of tables, ℓ, remains free. As a consequence, for any given value of ℓ, there exist a unique configuration once α and t are chosen.

Typically, the number ℓ of tables chosen, is the smallest or second smallest value that allows reaching the expected success probability[4] given by Chapter 2. In practice, only a small range of values for ℓ is used, since, above a certain number of tables (typically six tables and up), adding a table does not reduce the precomputation time significantly but does increase the attack significantly. Therefore, in practice, for a given success probability and a given memory for the attack phase, only a few configurations of RTs are meaningful.

6.5.2.2. DSRT

By using DSRTs instead of RTs, new parameters are available for determining possible configurations: the number of steps τ, and their positions. Considering various numbers of steps and their possible positions, a large number of DSRT configurations can reach

[4] Certain success probabilities are only achievable by using a sufficient number of tables.

the expected success probability and memory. When considering τ steps, the number of possible configurations is technically bounded by $(t-1)^\tau$. Among these, many provide better performance than RTs. However, some of them perform worse. This often arises for example when the steps are mostly located towards the left part of the chains. Indeed, in such a case, a lot of very short chains are stored, causing a significant increase in memory without significantly increasing the success probability.

6.5.3. Parameters

We considered configurations with one to five steps positioned between $0.3t$ and t. Placing steps further to the left of the table (in columns earlier than about $0.3t$) is possible and the formulas apply perfectly to these cases, but in practice these tables are very inefficient and are therefore never used. Using more than five steps is possible but we have observed that it does not improve the trade-off between precomputation time and attack time sufficiently to justify the effort in finding the optimal step positions. To determine the configurations that reach the success probability and the available memory, we performed a search according to Algo. 3 available in Appendix A.3.

We chose a set of size $N = 2^{42}$ and a memory of size 63.9 GB distributed over ℓ tables. $N = 2^{42}$ has been used to provide an easy comparison with articles [ABC15, ACLA21], also dealing with this space. The memory was chosen to enable use of our results in practical cases.

For each evaluation we performed, we identified the configurations that reach the expected success probability and available memory according to Algo. 3. For all configurations, the precomputation time and the attack time are plotted, both expressed in number of hash operations.

6.5.4. Results

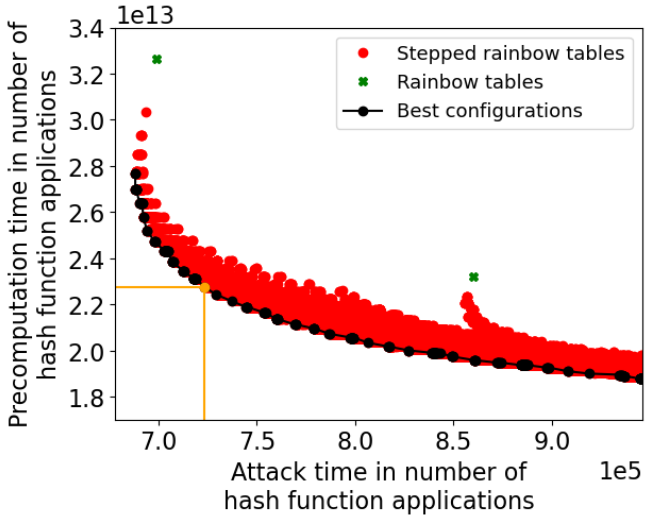
> **Takeaway:** No matter the success probability, there is always a configuration where DSRTs behave better than RTs.

Results are presented in Figures 6.5a, 6.5b, and 6.5c, which correspond to success probabilities of 96%, 98%, and 99.5%, respectively. The (green) crosses in the figures represent the results for RTs and the (red) dots, the results for DSRTs. Black dots identify the best results, which indeed all correspond to DSRTs.

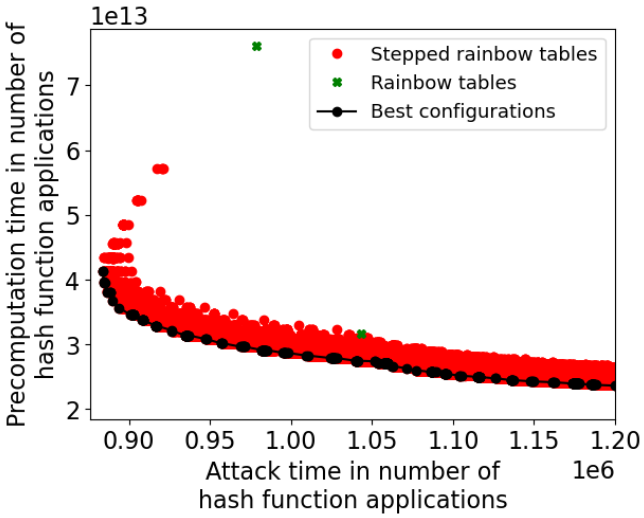
It is worth noting that the precomputation times differ between Figures 6.5a, 6.5b, and 6.5c depending on both the number of chains and the number of tables to generate to reach the target coverage given the target memory.

The main highlight is that, no matter the success probability, there is always a configuration where DSRTs behave better than RTs. In other words, when comparing RTs and DSRTs for a same memory and success probability, RTs are not on the Pareto frontier of configurations.

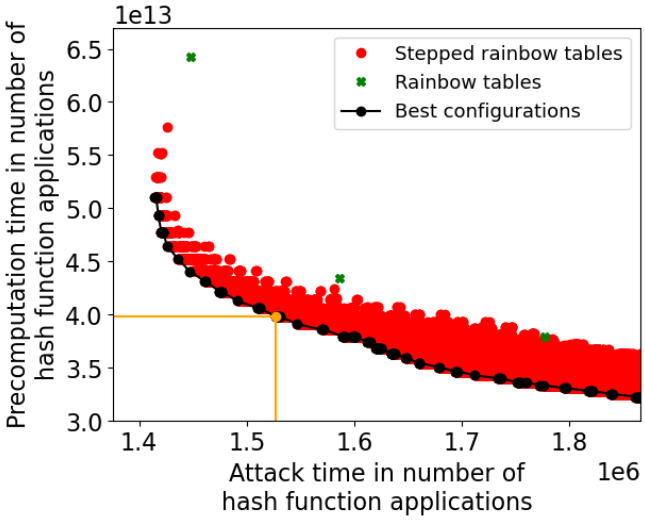
The relative gain depends on the success probability, which is mostly due to the maximality factor α. Indeed, whatever the success probability, using DSRTs reduces α and thus the number of chains to be computed during the precomputation phase. Therefore the gain in precomputation time is even more important when the success probability requires a larger maximality factor.



(a) Trade off between precomputation time and attack time **96%** of success, $N = 2^{42}$ and a 63.9 GB memory. 2 and 3 tables used.



(b) Trade off between precomputation time and attack time **98%** of success, $N = 2^{42}$ and a 63.9 GB memory. 2, 3 and 4 tables used.



(c) Trade off between precomputation time and attack time **99.5%** of success, $N = 2^{42}$ and a 63.9 GB memory. 3 to 5 tables used.

Figure 6.5. – Trade-off between precomputation and attack.

Table 6.1. – Expected gain illustrated on several examples with DSRT and RTs. Precomputation and attack phase numbers are quantity of cryptographic operations.

Success probability: 96%		
	Precomputation	Attack
2 DSRT	2.46×10^{13}	6.99×10^5
2 RT	3.26×10^{13}	
Gain	25%	
2 DSRT	2.32×10^{13}	7.17×10^5
3 RT		8.6×10^5
Gain		17%
Success probability: 98%		
	Precomputation	Attack
2 DSRT	2.96×10^{13}	9.79×10^5
2 RT	7.59×10^{13}	
Gain	61%	
2 DSRT	3.17×10^{13}	1.04×10^6
3 RT		9.26×10^5
Gain		11%
Success probability: 99.5%		
	Precomputation	Attack
3 DSRT	4.4×10^{13}	1.45×10^6
3 RT	6.42×10^{13}	
Gain	31%	
3 DSRT	4.35×10^{13}	1.59×10^6
4 RT		1.46×10^5
Gain		8%

For example, Figure 6.5b for success probability of 98%, using two DSRTs instead of two RTs divides the precomputation time by 2.56 without increasing the attack time (as all of these comparisons, for the same coverage and the same memory used). On the environment described in Section 6.4.2, this corresponds to generating a set of tables in approximately 5.6 hours, instead of 14.4 hours. If three vanilla RTs are used instead of two, DSRTs perform 11% faster. On a single core, these attack times correspond to approximately 0.08 second per attack.

Table 6.1 provides the expected gains of several configuration examples with various success probabilities.

We compared so far DSRTs and RTs either for the same attack time or for the same precomputation time. However, Figures 6.5a, 6.5b, and 6.5c shows that many other configurations ("best configurations", represented by black dots) may also be interesting in practice.

For instance, for a success probability of 99.5%, in the configuration identified by an orange dot in Figure 6.5c, the use of DSRTs results in a precomputation time of 3.98×10^{13} hash operations and an attack time of 1.53×10^6 hash operations. Compared to the 3-tables RT configuration, this DSRT configuration allows a reduction of 38% in the precomputation time for an attack time phase only 5% slower.

In the same vein, in Figure 6.5a, the DSRT configuration identified by an orange dot has a much faster attack time than the 3-table RT configuration (right green cross) and at the same time has a slightly faster precomputation time. Indeed, using this DSRT configuration instead of three RTs decreases by 2% the precomputation time, and decreases by 16% the attack time. Compared to the 2-tables RT configuration (left green cross), this DSRT configuration allows for a reduction of more than 30% in the precomputation time for an attack time phase only 4% slower.

6.5.4.1. Example Case

> **Reminder:** m_0 is the number of chains considered in column 0, when the precomputation begins. As the precomputation progresses, the number of surviving chains in each subsequent column decreases due to merges between chains.

Figures 6.6a and 6.6b illustrate the shape of the DSRT for a 98% success probability, with two RTs and two DSRTs as presented in Table 5.5.

Figure 6.6a shows the number of surviving chains per column for both DSRT and RT. The precomputation time for RT is the area under the blue curve, and the precomputation time for DSRT is the area under the pink curve. In this particular case, the m_0 used for precomputing the RT table is over 12.5 times higher than the m_0 used for precomputing the corresponding DSRT. This difference in m_0 explains the gain in precomputation time for DSRT.

Figure 6.6b shows the final shape of both DSRT and RT. It might suggest that the DSRT requires more memory than the RT, as it contains more chains. However, this is not the case. The DSRT and the RT require the same amount of memory, primarily due to the difference in m_0. The memory required to store the SP for the RT is given by:

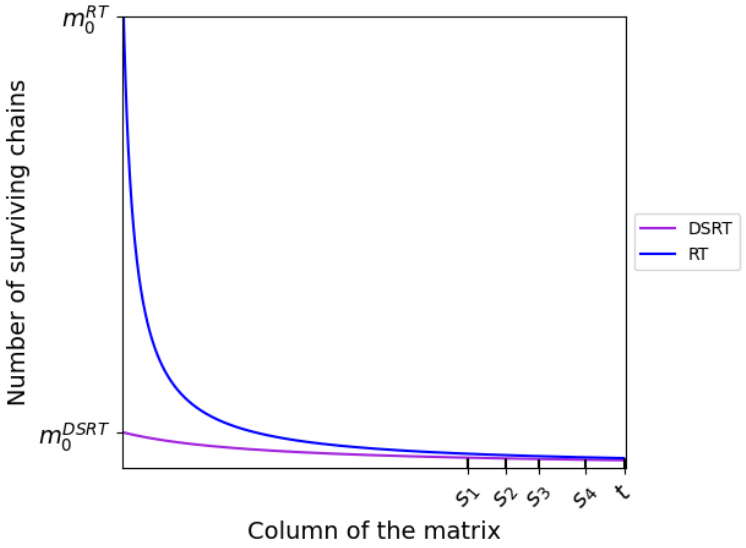
$$M_{sp}^{RT} = \ell m_t \lceil \log_2(m_0) \rceil.$$

while the memory required for the $DSRT$ is:

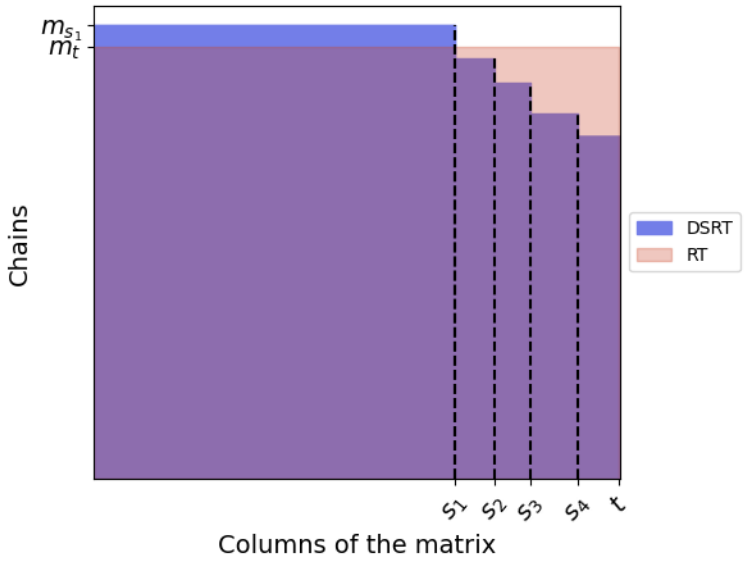
$$M_{sp}^{DSRT} = \ell m_{s_1} \lceil \log_2(m_0) \rceil.$$

As shown in Figure 6.6b, the number of chains in the RT (m_t) is slightly lower than in the DSRT (m_{s_1}). However, the difference is offset by the fact that the RT precomputation begins with 12.5 more SPs than the DSRT, so the $\log_2$ term in the formulas compensates for the higher number of chains in the table. As a result, the memory required for the DSRT SP is slightly lower than for the RT SP. However, as the EP must also be stored and compression is more efficient for the RT than for the DSRT (because the compression of one set of chains is more efficient than the compression of five set of chains), the DSRT EP thus requires slightly more memory than the RT EP. When the memory taken by the EP and SP are summed, DSRT and RT require finally the same amount of memory, even though the DSRT has more chains.

Both the DSRT and RT configurations have the same average attack time. At first glance, the DSRT configuration may appear to take slightly more time in attack since it has fewer chains on the right hand side of the table. However, a side effect of the DSRT is that it helps to reduce the cost of false alarms. In Figure 6.6b, chains ending before t account for slightly more than 25% of the total number of chains in the table. Whenever a search is performed on the left of column s_1, there is a 25% chance that a merge with a chain ending at a step



(a) Precomputation of the matrix



(b) Corresponding final matrix

Figure 6.6. – DSRT VS RT for 98% success probability

before t will occur if a chain of the matrix is encountered. If the search is performed between s_1 and s_2, the chance is approximately of 20%, and so on. If it is a false alarm, the cost of ruling out this false alarm is lower than if the merge occurs with the main table. This time gain in ruling out false alarms compensates for the lack of columns in the right part of the table and allows for the same average attack time to be achieved.

6.5.5. Conclusion on Efficiency Comparison

6.5.5.1. Coverage

As seen in Chapter 2, the maximum coverage of a RT is approximately 86%. DSRTs, however, have a maximum coverage of 100%, obtained when a step is placed in every column. Of course, placing a step in every column is not worth the cost in memory. Instead, a few, well-positioned steps are used. Nevertheless, the coverage obtained with a DSRT is better than those obtained with a RT. This allows using one or two fewer tables to obtain the same coverage for the same memory. This decreases simultaneously the attack time, and the precomputation time (all other characteristics being equal).

6.5.5.2. Attack Time

Even for the same number of tables used, there are always better configurations of DSRTs with the same memory and coverage as RTs, that are faster in both attack and precomputation. The gain in precomputation is obtained from the smaller number of chains that have to be computed to obtain the same coverage.

When using the same number of tables, the attack time is smaller for a less obvious reason. As mentioned in Section 6.3 and illustrated in Section 6.5.4.1, when a false alarm is detected at a step, the attack chain is not rebuilt until the last column of the table. This allows us to decrease the average cost of false alarms and thus, the attack time.

6.5.5.3. Memory

Intuitively, DSRTs are more expensive in memory than RTs. However, in this chapter, RTs and DSRTs are compared for the same memory and the same coverage.

The memory used to store EPs is larger for DSRTs than for RTs, but as fewer elements are considered at the beginning of the generation, less memory is required to store SPs. The storage of SPs allows us to keep the total storage cost of DSRTs close to those of vanilla tables and thus obtain more efficient tables in precomputation and attack for the same memory and coverage.

6.5.5.4. Memory Accesses

Depending on the memory used for the trade-off, the number of memory accesses may also be quite relevant to determine the real cost of an attack.

The number of memory accesses needed for an attack with DSRTs and RTs is very close. In our experiments, DSRTs in their optimal configurations typically require fewer memory accesses. This is particularly the case when DSRTs have fewer tables than vanilla RTs, which is often the case, as discussed in Section 6.5.5.1.

6.6. Conclusion

This chapter introduces DSRTs, which outperform vanilla RTs. The key idea of DSRTs consists in recycling chains that merged during the precomputation phase instead of discarding them. These recycled chains are shorter than vanilla ones. This leads to the concept of *descending stepped* RTs, a name inspired by their staircase-like appearance in a graphical representation.

The attack phase is modified accordingly to take advantage of the new construction. DSRTs allow configurations that are not reachable by vanilla RTs. The impact is twofold.

Firstly, DSRTs always perform better than vanilla RTs. Either precomputation is reduced without increasing the attack time, or the attack time is reduced without increasing precomputation. Both of those characteristics are reduced in certain scenarios. The gain depends on the problem parameters and other characteristics. In the practical examples we explored, DSRTs divide by 2.56 the precomputation time for the same attack time, same coverage and same memory used, or reduce the attack time by up to 17% for the same precomputation time, same coverage and memory.

Secondly, DSRTs are able to slightly increase a parameter to significantly reduce another one. For instance, for the same coverage and the same memory used, DSRTs can reduce the precomputation time by 38% while increasing the attack time by 5%.

The consequence of the reduction in precomputation time in particular is not to be understated. The precomputation phase is the bottleneck of RTs for large spaces today. Finally, a reduction in the precomputation cost directly translates into a (admittedly modest, but significant) increase in the reachable attack space.

7 Ascending Stepped Rainbow Tables

Why Gloria Gaynor is the favorite singer of the ASRT chains?
Because they want to survive!

7.1. Motivations

Chapter 6 introduced Descending Stepped Rainbow Tables (DSRTs) and discussed their advantages over vanilla RTs, especially in terms of precomputation time reduction. In this chapter, we introduce a novel variant, the *Ascending Stepped Rainbow Tables* (ASRTs), which can be seen as an inverse approach to DSRTs.

Unlike DSRTs, which recycle chains instead of discarding them, ASRTs add new chains during the precomputation phase to the already computed ones. This results in a larger set of chains, particularly aligned on the matrix's right hand side. Depending on the conditions, ASRTs can outperform DSRTs in terms of both precomputation and attack times, or perform better than DSRTs in attack at the cost of a slower precomputation phase.

After introducing ASRTs, we analyze their performances and detail scenarios where ASRTs outperform DSRTs and traditional RTs and explain why such performances is reached. We continue to approach ASRTs just as we did with DSRTs, considering the overall trade-off between precomputation time, attack time, memory, and coverage, rather than focusing solely on attack time.

Takeaway:

In this Chapter, we introduce a novel variant, the Ascending Stepped Rainbow Tables (ASRT).

- ASRTs add new chains during the precomputation phase, leading to a larger set of chains.
- ASRTs perform better than DSRTs under certain conditions.
- Under other conditions, ASRTs are faster in attack than DSRTs at the cost of an extended precomputation phase.
- Compare to DSRTs, a wider range of ASRT configurations are possible when targeted a given coverage and memory.

7.2. Overview

The ASRT algorithm involves a gradual addition of chains to the matrix during the precomputation phase. In well-chosen columns of the matrix called *steps*, a cleaning is performed, and new chains are added to the matrix. These chains do not begin at column 0, but rather at some columns s_i, where $0 < s_i < t$. As a result, the matrix consists of chains that start in different columns but all end in the last column. The Ascending Stepped Rainbow Matrix (ASRM) can be cleaned in the usual way since the added chains are computed using the same hash-reduction functions as those already present in the matrix.

The addition of chains to the matrix in given columns results in an increase in both the precomputation and memory cost. However, this leads to a matrix with more chains present in the right part of the table, including in particular, more short chains. The purpose is thus to reduce the attack time by using these shorter chains. Moreover, a higher number of chains also leads to a higher success probability.

When using DSRTs and RTs, m_i (defined in Equation (2.7)) represents the number of distinct elements in column i. However, this notation is unsuitable for ASRTs since not all chains start in the same column. Therefore, we introduce $m_{i,j}$, which represents the number of distinct elements in column j that are part of chains starting in column i or earlier.

7.2.1. Precomputation

7.2.1.1. Generation

Figure 7.1 represents an ASRM with two steps and notable points represented.

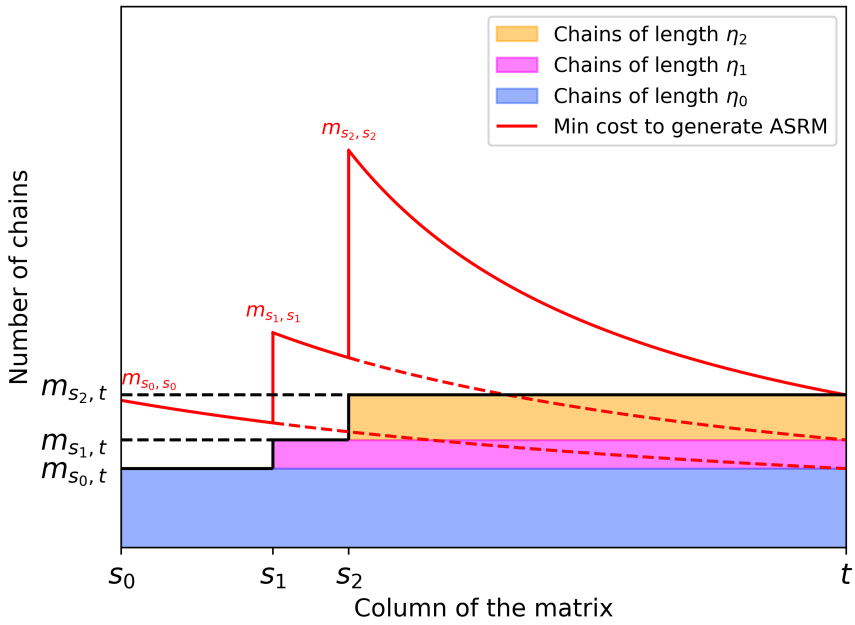


Figure 7.1. – ASRM with two steps

Reminder:

- During the precomputation phase, *cleaning* or *filtering* in a column c consists in keeping only chains that have different elements in column c and discarding all the other chains.
- The *length* of a chain or of a sub-chain is the number of hash-reduction function that need to be performed to generate it.

Differing from RT and DSRT, the initial number of SP used in ASRT precomputation is referred to as $m_{0,0}$. Accordingly, $m_{0,0}$ chains are computed from column 0 up to the first step, denoted by s_1, and then filtered to keep only m_{0,s_1} chains.

Next, a value m_{s_1,s_1}, greater than m_{0,s_1}, is chosen (refer to Equation (7.1) for the formal expression of m_{s_1,s_1}). Then, $m_{s_1,s_1} - m_{0,s_1}$ elements are chosen in a way to be as small as possible[1] and not equal to any of the m_{0,s_1} elements that remained in column s_1.

At this point, m_{0,s_1} chains with length s_1 remain from the first part of the precomputation, while $m_{s_1,s_1} - m_{0,s_1}$ additional elements have been selected. In total, m_{s_1,s_1} elements are present in column s_1. The $m_{s_1,s_1} - m_{0,s_1}$ elements are the SP of chains starting in s_1.

The process then continues by computing the next columns of the m_{s_1,s_1} chains, regardless of where they started. This computation continues until reaching column s_2, where another cleaning is performed.

Whenever chains starting in different columns merge, the longer chain is always retained. Intuitively, keeping a shorter chain decreases the success probability of the table too much compared to the gain in attack time and is, therefore, not worth it.

After cleaning in column s_2, m_{s_1,s_2} chains remain. As previously, $m_{s_2,s_2} - m_{s_1,s_2}$ elements are added to the m_{s_1,s_2} that remain from the previous part. The computation of chains starts again with m_{s_2,s_2} chains.

This continues for the τ steps chosen and until reaching the last column, i.e., the column t. As for DSRT, the total number of steps is denoted by τ, and the set of steps is denoted by $\{s_1, s_2, ..., s_\tau\}$, with by convention $s_0 = 0$ and $s_{\tau+1} = t$. The choice of the placement of the steps is explained in Section 7.4.

7.2.1.2. Maximality

Reminder: A table generated using $m_0 = N$ elements is known as a *maximal* table.

Similarly to RT and DSRT, one could set $m_{s_0,s_0} = N$, $m_{s_1,s_1} = N$,.., $m_{s_\tau,s_\tau} = N$ to achieve a maximal ASRT. However, this choice would result in a significant increase in the precomputation phase cost for comparatively little benefit. Therefore, only $m_{s_i,s_i} < N$ with $i \in \{0, 1, ..., \tau\}$ are selected.

To determine m_{s_i,s_i}, a maximality factor α_i is chosen for each step, with $0 < \alpha_i < 1$ and $i \in \{0, 1, ..., \tau\}$. Unlike RT and DSRT, ASRT has multiple maximality factors (one per step). Given α_i and $m^{\max}_{t-s_i}$, the value of m_{s_i,s_i} is obtained from Equation (7.1).

$$m_{s_i,s_i} = \frac{\alpha_i}{1-\alpha_i} m^{\max}_{t-s_i} \tag{7.1}$$

[1] This choice is justified to minimize the total memory.

where $m^{\max}_{t-s_i}$, obtained from Equation (2.8), is the maximum number of chains that started in column s_i and remain in column t.

7.3. Characterization

7.3.1. Preliminaries

To facilitate the characterization and visualization of ASRT, this section introduces notations and notions used when considering ASRT. The introduced notions are then used throughout the chapter.

Firstly, η_i denotes the length of a chain starting at s_i. As $t - s_i + 1$ columns are present between s_i and t, $\eta_i = t - s_i$. We denote ψ_{c,s_i}, the number of columns between s_i and column c if $c > s_i$, and 0 if $c \leqslant s_i$. This notation is employed to simplify future equations, resulting in $\psi_{c,s_i} = \max(c - s_i + 1, 0)$.

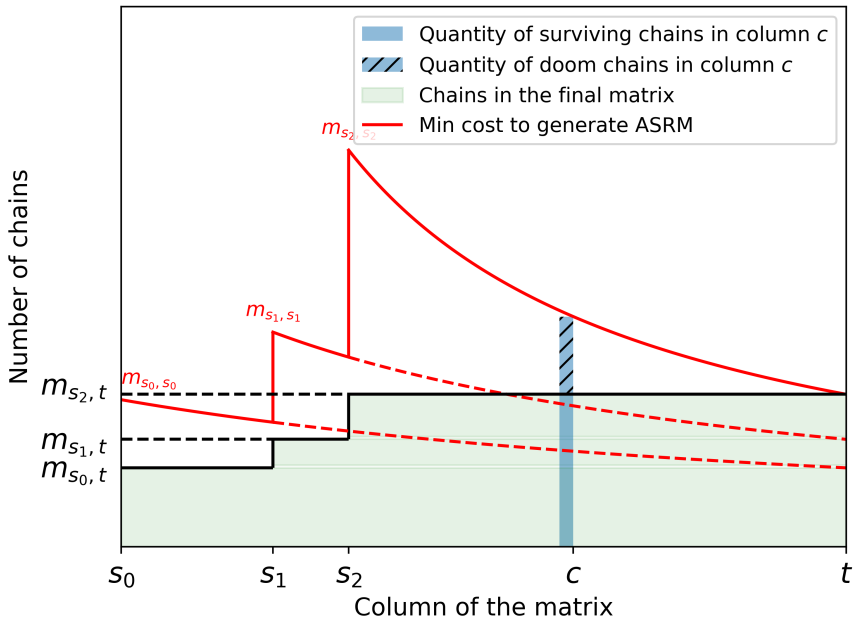


Figure 7.2. – Quantity of doom and surviving chains in a given column in an ASRM with two steps

When considering ASRT, it is essential to differentiate chains that are part of the final matrix (and thus present in the final table) from those that were present at a given column during precomputation but not in the final matrix (due to merges).

Given a column c and a starting column s_i, the *surviving chains* are chains that begin in column s_i or earlier and, during the precomputation phase, remain after cleaning in column c. These chains *may or may not* be present in the final matrix. The number of surviving chains in column c starting at s_i or earlier is denoted by $m_{s_i,c}$ and is given by Equation (7.2), which is derived from Equation (2.7).

$$m_{s_i,c} = \begin{cases} \frac{2N}{\psi_{c,s_i} + \frac{2N}{m_{s_i,s_i}}} & c \geqslant s_i \\ 0 & c < s_i \end{cases} \tag{7.2}$$

Among the surviving chains in column c, some will merge and, consequently, will not be present in the final cleaned matrix. Given a column c and a step s_i, the *doom chains* are chains that begin in column s_i or earlier and, during the precomputation phase, remain after cleaning in column c but do not belong to the final matrix.

Figure 7.2 illustrates the notion of *surviving chains* and *doom chains* by representing the number of surviving chains and doom chains in a column c.

The number of doom chains in column c starting at s_i or earlier is denoted by $m^d_{s_i,c}$ and is given by Equation (7.3), which is derived from Equation (7.2).

$$m^d_{s_i,c} = \begin{cases} m_{s_i,c} - m_{s_i,t} & c \geqslant s_i \\ 0 & c < s_i \end{cases} \tag{7.3}$$

Moreover, to effectively use ASRT, it is often necessary to determine the closest step to the left of a given column c. The index of this closest step is denoted by $k^A(c)$ and is defined in Definition 7.1.

Definition 7.1. *We denoted by $k^A(c)$ the index of the rightmost step that is to the left of column c, i.e., $s_{k^A(c)} \leqslant c < s_{k^A(c)+1}$ and with $k^A(c) \in \{0, \ldots, \tau\}$.*

7.3.2. Precomputation Time

> **Reminder:** The value $m_{c,c'}$ depicts the number of surviving chains in column c' that starts in column c or before.
> It follows that the value m_{s_i,s_i}, is the number of chains in column s_i, starting in column s_i itself, or before.

If the cleaning method is not used, the precomputation time of an ASRM is given by multiplying, for each step, the m_{s_i,s_i} chains considered by the number of columns until the next one. The maximum precomputation time $P_{\max}$ can then be obtained by summing up the product of m_{s_i,s_i} and $(s_{i+1} - s_i)$, as shown in Equation (7.4).

$$P_{\max} = \sum_{i=0}^{i=\tau} m_{s_i,s_i}(s_{i+1} - s_i) \tag{7.4}$$

However, as with RT and DSRT, filtering can significantly reduce the precomputation time by reducing the number of hash computations required. The minimum theoretical precomputation time for a given ASRM can be achieved when a filter is used in every column.

> **Reminder:** $s_{k^A(c)}$ is the rightmost step to the left of c. It follows that $m_{s_{k^A(c)},c}$, is the number of surviving chains in column c starting in column $s_{k^A(c)}$ or before.

The minimum number of operations needed to generate the matrix is obtained by summing up all $m_{s_{k^A(c)},c}$ values, for $c \in \{0, .., t-1\}$, as shown in Equation (7.5).

$$P_{\min} = \sum_{c=0}^{t-1} m_{s_{k^{A(c)}},c} \tag{7.5}$$

7.3.3. Success Probability

The success probability for a single ASRT is given by Theorem 7.1. Similarly to DSRT and RT, the success probability for ℓ ASRTs is obtained by applying Equation (2.11). As for RT and DSRT, the intuitive success probability is the number of distinct elements in an ASRM. Therefore, the success probability is obtained by counting the number of surviving chains in each column of the final matrix, which is divided by the total number of elements in the search space i.e., N.

Theorem 7.1. *Given a single ASRT with τ steps and $t+1$ columns, the success probability for this single ASRT is given by:*

$$P = 1 - \prod_{i=0}^{\tau} \left(1 - \frac{m_{s_i,t}}{N}\right)^{s_{i+1}-si}.$$

Proof. In each column c, there are $m_{s_i,t}$ distinct elements. The probability of finding the searched element in a given column is $\frac{m_{s_i,t}}{N}$.

The probability of not finding the searched element in a column c is therefore

$$\left(1 - \frac{m_{s_i,t}}{N}\right).$$

Between each step, there are $s_{i+1} - s_i$ columns. Thus, the probability of not finding the searched element in any column between s_i and s_{i+1} is

$$\left(1 - \frac{m_{s_i,t}}{N}\right)^{s_{i+1}-si}.$$

As there are τ steps plus the step s_0, the probability of not finding the searched element in the entire ASRT is

$$\prod_{i=0}^{\tau} \left(1 - \frac{m_{s_i,t}}{N}\right)^{s_{i+1}-si}.$$

□

7.3.4. Memory

The storage approach for ASRT is similar to RT and DSRT with a slight difference in the computation of the SP storage lower bound. Like RT and DSRT, we use the memory lower bound to compare each variant since the available storage methods [AC13] are very close to this lower bound and considering the lower bound simplifies the analysis.

The total memory used by ASRT, denoted by M^{ASRT}, is the sum of the memory used for storing the SPs and the memory required to store the EPs, denoted respectively by M_{sp}^{ASRT} and M_{ep}^{DSRT}.

The method for storing EPs is the same as for DSRT, where each collection of chains starting at a given step is compressed and stored separately from chains starting at other steps.

The memory used to store EPs is given by Equation (7.6), which is adapted from [AC13] and is the same as for DSRT introduced in Chapter 6, Equation (6.3).

$$M_{ep}^{ASRT} = \ell \left(\log_2 \binom{N}{m_{0,t}} + \sum_{i=1}^{\tau} \log_2 \binom{N}{m_{s_i,t} - m_{s_{i-1},t}} \right). \tag{7.6}$$

Unlike in RT and DSRT, the number of SPs varies depending on which step the corresponding chains start at. Hence, the formula used for SPs needs to be adapted to take this into account.

At each step s_i, there are m_{s_i,s_i} distinct elements and thus possible SPs to consider. Therefore, the minimal naive way to store SPs is to store $(m_{s_i,t} - m_{s_{i-1},t}) \log_2(m_{s_i,s_i})$ for each step. This is because there are $m_{s_i,t} - m_{s_{i-1},t}$ elements to store for each step, and for each step, there are m_{s_i,s_i} possible SPs.

However, when considering ASRT, among the m_{s_i,s_i} possible elements, m_{s_{i-1},s_i} are already part of the chains computed previously and cannot be chosen. These m_{s_{i-1},s_i} have only a few chances to be among the m_{s_i,s_i} elements considered in step s_i. Thus, we can compute the value $m^u(i, z_p)$, which depicts the maximum number of possible elements among the m_{s_{i-1},s_i} elements that are among the m_{s_i,s_i} elements and with a probability of $1 - p$ that more elements than $m^u(i, z_p)$ are among the m_{s_i,s_i} elements. Here, z_p is the quantile function of the standard normal distribution. In our experiments, we have chosen $p = 0.99994$, i.e., $z_p = 4$ to ensure that the probability that more elements than $m^u(i, z_p)$ are among the m_{s_i,s_i} elements is less than 0.0007.

The variable $m^u(i, z_p)$ is defined in Proposition 7.2.

Proposition 7.2. *Let s_i a step using m_{s_i,s_i} SPs, an let m_{s_{i-1},s_i} elements of chains starting at s_{i-1} and remaining at s_i after cleaning. The maximum number of elements from m_{s_{i-1},s_i} that are in the m_{s_i,s_i} elements with a probability $1 - p$ is denoted by $m^u(i, z_p)$ with:*

$$m^u(i, z_p) = m_{s_{i-1},s_i} - \left(m_{s_{i-1},s_i} \frac{m_{s_i,s_i}}{N} + z_p \times \sqrt{m_{s_{i-1},s_i} \frac{m_{s_i,s_i}(1 - m_{s_i,s_i})}{N^2}} \right).$$

Proof. The probability that an element from the m_{s_{i-1},s_i} elements of the previous chains is part of one of the m_{s_i,s_i} elements to choose from is $\frac{m_{s_i,s_i}}{N}$. This is because the hash-reduction function is considered random, and thus the m_{s_{i-1},s_i} elements are considered as m_{s_{i-1},s_i} random elements in N. The number of m_{s_{i-1},s_i} elements that are part of the m_{s_i,s_i} elements to choose from thus follows a binomial distribution with parameters $p = \frac{m_{s_i,s_i}}{N}$ and $n = m_{s_{i-1},s_i}$. The expected value E of this distribution is thus given by $E = np = m_{s_{i-1},s_i} \frac{m_{s_i,s_i}}{N}$, and the standard deviation σ is given by

$$\sigma = \sqrt{p(1-p)n} = \sqrt{m_{s_{i-1},s_i} \frac{m_{s_i,s_i}(1 - m_{s_i,s_i})}{N^2}}.$$

We obtain $m^u(i, z_p)$ by adding E with $z_p\sigma$ and subtracting this sum from $n = m_{s_{i-1},s_i}$. □

According to the Proposition 7.2, the maximum number of possible SPs to store for step s_i with $1 \leqslant i \leqslant \tau$ is given by $m_{s_i,s_i} - m^u(i, z_p)$. The total memory required to store all the SP can be computed using Equation (7.7) which is derived directly from [AC13].

$$M_{sp}^{ASRT} = \ell m_{s_0,t} \lceil \log_2(m_{s_0,s_0}) \rceil + \ell \left(\sum_{i=1}^{\tau} m_{s_i,t} \lceil \log_2(m_{s_i,s_i} - m^u(i, z_p)) \rceil \right). \tag{7.7}$$

By taking $z_p = 4$, the probability that the memory taken by the SPs is larger than the one given by Equation (7.7) is less than 0.0007%.

The total memory used by ASRT, M^{ASRT}, is then obtained by adding M_{sp}^{ASRT} and M_{ep}^{ASRT}, and is given in Equation (7.8).

$$M^{ASRT} = M_{sp}^{ASRT} + M_{ep}^{ASRT} \tag{7.8}$$

7.3.5. Attack Phase

7.3.5.1. Attack Process

The attack using ASRTs is similar to the attack using vanilla RT. Contrarily to DSRT, it is not required to define any metric for choosing the column in which the search will be performed. By construction, the most advantageous columns to perform a search are the right ones, as these are the columns for which the price to build a chain until column t is the lowest. In addition, these are the columns with the highest success probability since they have the highest number of chains. These columns also contain the shortest chains.

Thus, the attack begins by assuming that the searched element is in the second to last column. The attacker computes $R_t(y)$ and checks if the result is one of the EPs stored. If this is the case, the attacker builds the attack chain from the corresponding SP (regardless of the column in which the chain starts) to column $t-1$ and thus obtains $x_{t-1,j}$ with j the rows of the matched EP. The attacker computes $h(x_{t-1,j})$ and if $h(x_{t-1,j}) = y$ the attack ends and the searched element is x_{t-1}. If $h(x_{t-1,j}) \neq y$ it is a false alarm and the attack continues in the columns further on the left until the element they are looking for is found or until the end of the table is reached.

7.3.5.2. Roadmap

Next sections introduce propositions used to characterize the average cost of the attack. The average attack time is the sum of the average cost of the search in each column multiplied by each column's probability that a search occurs in this column. Thus, we first need to define the cost and probability of a search in a given column c.

The search in a given column c is equal to the sum of the cost of each possible event multiplied by their respective probability of occurrence.

We, thus, define each event, namely *no alarm*, *false alarm*, and *true alarm*. We define the cost of each event and its probability of occurrence. We then multiply cost of each event by its probability of occurrence and sum the whole to obtain the cost of search in any column of the table.

7.3.5.3. No Alarm

Cost In the process of searching for a match in column c, if the event of *no alarm* occurs, it indicates that there is no match with any EP in the table. In such a scenario, the cost associated with this event is equal to the cost of the construction of a chain from column c to t, which is equal to $t - c$.

Probability

> **Reminder:**
>
> - $s_{k^A(i)}$ is the rightmost step to the left of column i.
> - $m_{s_{k^A(i)},i}$ is the number of surviving chains in column i starting in column $s_{k^A(i)}$ or before.

If no match occurs with any EP in the table, it implies that the attack chain does not merge with any chain of the matrix. More formally, for all columns i between c and t, the elements of the attack chain are not present in any of the $m_{s_{k^A(i)},i}$ elements of the matrix in column i. This corresponds to the fundamental results of Equation (3) in [Oec03], which is generalized in Lemma 7.3. Instead of m_i elements, $m_{s_{k^A(i)},i}$ surviving chains are present in column i, in the ASRT variant.

Lemma 7.3. *Given a column c with $c < t$, the probability that the attack chain does not merge with any chain of the rainbow matrix by t, given it had not merged in or before c, is:*

$$p^A_{noalarm}(c) = \prod_{i=c+1}^{t} \left(1 - \frac{m_{s_{k^A(i)},i}}{N}\right).$$

7.3.5.4. True Alarm

Cost When searching for an element in column c, a *true alarm* occurs when the searched element is found in that column.

> **Reminder:**
>
> - The *length* of a chain is the number of operations to perform to compute it.
> - The value η_i is the length of chains starting in s_i. Thus $\eta_i = t - s_i$.

The cost associated with true alarm event, depends on the column in which the searched element is found. Since the attack chain needs to be built from column c to t and then from the corresponding SP to column c, the cost of the search is the length of the chains in which the corresponding SP is found.

To evaluate the cost of the search, Definition 7.2 introduces $\rho^A_{i,j}$, which represents the ratio of chains with a length of s_i in column j. For instance, consider an ASRT with two steps in columns s_1 and s_2 with $s_1 < s_2 < t$. In column c with $s_2 < c < t$, chains can be of length s_1, s_2, or t. The value of $\rho^A_{1,k^A(c)}$ gives the ratio of chains in column c that have a length of s_1. Definition 7.2 generalizes this concept to all steps and columns.

Definition 7.2. *Given a step s_i and a step s_j, the ratio of chains with length s_i in a column c with $s_j < c < s_{j+1}$ is given by $\rho^A_{i,j}$ defined as follows:*

$$\rho^A_{i,j} = \begin{cases} \frac{m_{s_i,t} - m_{s_{i-1},t}}{m_{s_j,t}} & i > 0 \\ \frac{m_{s_0,t}}{m_{s_j,t}} & i = 0 \end{cases}$$

Given that a true alarm occurs, its cost is the sum, for each s_i such that $s_i \leqslant s_{k^A(c)}$, of the probability that the true alarm is caused by a chain of length s_i with $0 \leqslant i \leqslant k^A(c)$, denoted by $\rho^A_{i,k^A(c)}$, multiplied by its length s_i. This cost is given by Proposition 7.4.

Proposition 7.4. *Given a search performed in a column c and $k(c)$ the index of the leftmost step that is to the right of column c, the number of hash operations needed to rule out a true alarm is:*

$$\sum_{i=0}^{k^A(c)} \rho^A_{i,k^A(c)} \eta_i.$$

Proof. The value of $\rho^A_{i,k^A(c)}$ is the ratio of chains with length η_i in column c. In column c, chains that remain in the final matrix have a length between $\eta_{k^A(c)}$ and t.

If a true alarm is raised when searching in column c, this means that the attack chain merged with one of the $m_{s_{k^A(c),t}}$ chains present in the final matrix in column c. Each of these $m_{s_{k^A(c),t}}$ chains has a different length, the cost for ruling out the alarm, is thus the sum of the probability of merging with a chain of a given length, multiplied by its length.

Given a merge with one of the chains present in the final matrix in column c, the probability of merge with a chain of length η_i, with $0 \leqslant i \leqslant k^A(c)$ is $\rho^A_{i,k^A(c)}$.

Thus, $\forall i \in \{0, \ldots, k^A(c)\}$, the probability of matching a chain of length η_i multiplied by the cost of going through all the chain is $\rho^A_{i,k^A(c)}\eta_i$. □

Probability Equation (7.2) asserts that a given column c contains $m_{s^A_{k(c)},t}$ elements. Consequently, for each column c, the probability of finding the searched element in any of the column's c elements can be computed straightforwardly. This probability is provided by Proposition 7.5.

Proposition 7.5. *The probability that a true alarm occurs when starting the attack chain in c is:*

$$p_{find}(c) = \frac{m_{s_{k^A(c)},t}}{N}.$$

Proof. As the search space consists of N elements, and since the column c contains $m_{s_{k^A(c)},t}$ elements, the probability of finding the searched element among the $m_{s_{k^A(c)},t}$ elements of column c is simply $\frac{m_{s_{k^A(c)},t}}{N}$. □

7.3.5.5. False Alarm

The false alarm's cost and probability depend on various factors. To characterize it efficiently, we first need to introduce several propositions.

Reminder: Given a column c and a starting column s_i:

- There is $m_{s_i,c}$ *surviving chains* in column c. These chains are chains starting at column s_i or earlier and that remain after cleaning in column c. These chains *may or may not* be present in the final matrix.
- There are $m^d_{s_i,c}$ *doom chains* in column c. These are chains starting in column s_i or earlier, remaining after cleaning in column c but *are not present* in the final matrix.

We first introduce in Proposition 7.6 the probability that the starting element of the attack chain is not part of a surviving chain (Equation (7.2)) in a given column.

Proposition 7.6. *Given a column c in which a search is performed, the probability that the starting element of the attack chain is not among the elements of surviving chains starting in $s_{k^A(c)}$ in column c is:*

$$p_{notsurviving}(c) = 1 - \frac{m_{s_{k^A(c)},c}}{N}.$$

Proof. The number of surviving chains in column c is, by construction, $m_{s_{k^A(c)},c}$, thus the probability that a random element in N is not one of the $m_{s_{k^A(c)},c}$ elements of those chains is straightforward. □

Next, we introduce in Proposition 7.7, the probability that the starting element of the attack chain is an element of a doom chain (the number of doom chains in a column c, $m^d_{s_i,c}$ has been defined in Equation (7.3)).

Proposition 7.7. *Given a column c in which a search is performed, the probability that the starting element of the attack chain is among doom chains is:*

$$p_{doom}(c) = \frac{m^d_{s_{k^A(c)},c}}{N}.$$

Proof. The probability that the starting element of the attack chains is among the elements of surviving chains in column c is by definition, the average number of doom chains in column c, $m^d_{s_i,c}$, divided by the number N, of elements in the searched space. □

Lemma 7.8 defines the probability that the attack chain does not merge with a chain starting in a precise step between two columns c and c'. Lemma 7.8 is obtained by application of Lemma 6.3 for the ASRT structure.

Lemma 7.8. *Given two columns c and c' and a step of index j with $c < c' \leqslant t$, the probability that the attack chain does not merge with any chain starting in step s_j or before by c', given it had not merged in or before c, is:*

$$p^A_{subnomrg}(c, c', j) = \prod_{i=c+1}^{c'} \left(1 - \frac{m_{s_j,i}}{N}\right).$$

Cost The cost of the false alarm depends on the column in which the alarm is detected. Thus, the cost to rule out a false alarm is η_i, with s_i the step at which the matrix chain that merged with the attack chain starts.

Probability When performing a search in column c, the probability of a false alarm depends, among other factors, on the value of the starting element of the attack chain. There are two possibilities:

a If the starting element is among the elements of doom chains, a false alarm will occur, but the probability of occurrence will vary for different steps.

b If the starting element is not one of the $m_{s_{k^A(c)},c}$ elements of surviving chains, either no alarm will be raised or a false alarm will occur.

These two possibilities have different probabilities of false alarm, and thus require separate propositions. Proposition 7.9 provides the probability of a false alarm in case (a), while Proposition 7.10 provides the probability of a false alarm in case (b).

Proposition 7.9. *Given an attack chain starting in c, the probability $p_{fa}(c,i)$ to raise a false alarm due to merge with chains of length η_i and given that the starting element of the attack chain is not an element of a surviving chain in column c is:*

$$p_{fa}(c,i) = \begin{cases} p^A_{subnomrg}(c,t,i-1) - p^A_{subnomrg}(c,t,i) & i > 0 \\ 1 - p^A_{subnomrg}(c,t,i) & i = 0 \end{cases}$$

Proof. The attack chain merging with any chain of length at least η_i is the complementary event of Lemma 7.8, for parameters (c,t,i), thus this probability is $1 - p^A_{\text{subnomrg}}(c,t,i)$. For the special case of $i = 0$, the attack chain can only merge with a chain of length η_i.

For $i > 0$, we define events E_1 and E_2 as "no merge occurs between c and t with a chain of length at least η_{i-1}" and "no merge occurs between c and t with a chain of length at least η_i", respectively. As $E_2 \subset E_1$, we deduce that $\Pr(E_1 \wedge E_2) = P(E_2)$. Therefore, we have:

$$\begin{aligned} p_{\text{fa}}(c,i) &= \Pr(E_1 \wedge \bar{E_2}) \\ &= \Pr(E_1) - \Pr(E_1 \wedge E_2) \\ &= \Pr(E_1) - \Pr(E_2) \\ &= p^A_{\text{subnomrg}}(c,t,i-1) - p^A_{\text{subnomrg}}(c,t,i) \end{aligned}$$

□

Proposition 7.10. *Given an attack chain starting in c, the probability of raising a false alarm due to merge with chains of length η_i and given that the starting element of the attack chain is among the elements of doom chains in column c:*

$$p'_{fa}(c,i) = \begin{cases} p^A_{subnomrg}(c,t,k^A(c)-1) & i = k^A(c) \wedge i \neq 0 \\ p^A_{subnomrg}(c,t,i-1) - p^A_{subnomrg}(c,t,i) & i \neq k^A(c) \wedge i \neq 0 \\ 1 - p^A_{subnomrg}(c,t,i) & i = 0 \end{cases}$$

Proof. In the cases of $i \neq k^A(c)$ or $i = 0$ (second and third cases), since c is among the doom chains present in column c, it follows that $\eta_{k_A(c)} < \eta_i$. If the starting element of the attack chain is among a doom chain in column c, this implies that the starting element is not among a surviving chain of any step starting to the left of column $s_{k_A(c)}$. Given that $\eta_{k_A(c)} < \eta_i$ when $i \neq k^A(c)$ or $i = 0$, the second and third case are obtained exactly as demonstrated in Proposition 7.9.

For the specific case of $i = k^A(c)$ and $i \neq 0$, we again define events E_1 and E_2 as "no merge occurs between c and t with a chain of length at least $\eta_{k^A(c)-1}$" and "no merge occurs between c and t with a chain of length at least $\eta_{k^A(c)}$". By the definition of a doom chain, a chain starting with an element among a doom chain in column c will merge with a chain of

length at least $\eta_{k^A(c)}$, therefore $\Pr(E_2) = 0$. Consequently, we have $\Pr(E_1 \wedge E_2) = 0$. From this, we deduce that, when $i = k^A(c)$ and $i \neq 0$, we have:

$$\begin{aligned} p_{\text{fa}}(c, i) &= \Pr(E_1 \wedge \bar{E_2}) \\ &= \Pr(E_1) - \Pr(E_1 \wedge E_2) \\ &= \Pr(E_1) \\ &= p^A_{\text{subnomrg}}(c, t, i-1) \end{aligned}$$

□

7.3.5.6. Cost of the Search in One Column

The number of operations needed to perform a search in a column c is given by Theorem 7.11. It is obtained by multiplying the probabilities of the three possible events (true alarm, false alarm, no alarm) by their respective costs and sum these results.

Theorem 7.11. *For a given column c, the average number of cryptographic operations C^A_c needed to perform a search is:*

$$\begin{aligned} C^A_c = p_{find}(c) &\sum_{i=0}^{k^A(c)} \rho_{i,k^A(c)}\eta_i \\ + p_{notsurviving}(c) &\sum_{j=0}^{k^A(c)} p_{fa}(c,j)\eta_j \\ + p_{doom}(c) &\sum_{j=0}^{k^A(c)} p'_{fa}(c,j)\eta_j \\ + (t-c)\, &p_{noalarm}(c). \end{aligned}$$

Proof. **(a)** The probability of a true alarm, denoted as p_{find}, is given by Proposition 7.5. Its corresponding cost is $\sum_{i=0}^{k^A(c)} \rho_{i,k^A(c)}\eta_i$ as stated in Proposition 7.4. Hence, the cost of a true alarm in column c can be expressed as $p_{\text{find}}(c) \sum_{i=0}^{k^A(c)} \rho_{i,k^A(c)}\eta_i$.

(b) A false alarm can occur under two scenarios: (E_1) "The starting element of the attack chain is not equal to an element of a surviving chain in column c". (E_2) "The starting element of the attack chain matches an element of a doom chain in column c".

Hence, the cost of a search in case of a false alarm is the sum of the probabilities of these two events multiplied by their respective costs.

- (E_1): The probability of event E_1 is provided by Proposition 7.6. The cost of a false alarm in this case, analogous to the true alarm, is, for all j with $0 \leqslant j \leqslant k^A(c)$, the probability of a merge with a chain of length exactly η_j multiplied by its cost (η_j). The

probability of merging with a chain of length exactly η_j under event E_1 is given by Proposition 7.9 as $p_{\text{fa}}(c,j)$. Hence, the cost of event E_1 is $\sum_{j=0}^{k^A(c)} p_{\text{fa}}(c,j)\eta_j$.

- (E_2): The probability of event E_2 is provided by Proposition 7.7. Analogous to E_1, the probability of a merge with a chain of length exactly η_j under event E_2 is given by Proposition 7.10 as $p'_{\text{fa}}(c,j)$. Hence, the cost of event E_2 is $\sum_{j=0}^{k^A(c)} p'_{\text{fa}}(c,j)\eta_j$.

Thus, the cost of a false alarm in column c is:

$$p_{\text{notsurviving}}(c) \sum_{j=0}^{k^A(c)} p_{\text{fa}}(c,j)\eta_j + p_{\text{doom}}(c) \sum_{j=0}^{k^A(c)} p'_{\text{fa}}(c,j)\eta_j.$$

(c) The probability of no alarm, denoted as $p_{\text{noalarm}}(c)$, is given by Proposition 7.3. Its cost is $t-c$, as shown in Section 7.3.5.3. Hence, the cost of no alarm in column c can be expressed as $(t-c)\,p_{\text{noalarm}}$.

By summing up the cost of the events (a), (b), and (c), C_c^A is obtained.

□

7.3.5.7. Average Attack Time

The average attack time using ℓ ASRTs is given in Theorem 7.12. As for RTs and DSRTs, this attack time is computed in two parts. The first part is the cost of the attack if the searched element is in the table, multiplied by the probability that the searched element is in the table. The second part is the cost of the attack if the searched element is not in the table multiplied by the probability that the searched element is not in the table.

Theorem 7.12. *Given N, ℓ ASRT with τ steps, the average number T of hash operations required to perform an attack is:*

$$T = \ell \sum_{c=1}^{t} \left(\frac{m_{s_{k^A(c)},t}}{N} \prod_{i=1}^{c-1} \left(1 - \frac{m_{s_{k^A(c)},t}}{N}\right) \sum_{j=1}^{c} C_{t-j+1}^A \right) + \ell \prod_{i=1}^{t} \left(1 - \frac{m_{s_{k^A(c)},t}}{N}\right) \sum_{c=1}^{t} C_c^A.$$

Proof. This formula is a generalization of Theorem 2.3 as it has been done for DSRT in Theorem 6.10. The number T is obtained by adding, on the one hand, the success probability of the attack using ℓ tables, multiplied by its average cost, and on the other hand, the failure probability of the attack using ℓ tables, multiplied by the cost of a failed search.

The first term is obtained by multiplying for each column c, the probability of a true alarm in the column, with the probability of no true alarm in all earlier iterations:

$$\frac{m_{s_{k^A(c)},t}}{N} \prod_{i=1}^{c-1} \left(1 - \frac{m_{s_{k^A(c)},t}}{N}\right).$$

This is multiplied by the cost of all searches performed until reaching this column: $\sum_{j=1}^{c} C_{t-j+1}^A$.

The second term is obtained by multiplying the failure probability using ℓ tables, namely

$$\ell \prod_{i=1}^{t} \left(1 - \frac{m_{s_{k^A(c)},t}}{N}\right),$$

with the cost of performing a search in all columns of a table, namely $\sum_{c=1}^{t} C_c^A$. □

7.4. Comparison

In this section, we compare ASRT, DSRT and RT. We firstly present in Section 7.4.1 the experimental validation of the analysis provided in Section 7.3. We then provide in Section 7.4.2 the methodology used for the comparison, and we finally present our results in Section 7.4.3.

In what follows, we call *configuration* a list of parameters describing a set of either RTs, DSRTs, or ASRTs. For RTs, a configuration is composed of one maximality factor α, a number of columns t, and a number of table ℓ. For DSRT, in addition to these three parameters a configuration is also composed of the steps positions $\{s_1, s_2, \ldots, s\tau\}$. Finally, the ASRT configuration is composed of the same parameters as DSRT plus the quasi-maximality factors considered at each step, defined by $\{\alpha_0, \alpha_1, \ldots, \alpha_\tau\}$.

7.4.1. Experimental Validation

In order to validate the formulas characterizing the different variants, RT, DSRT, and ASRT were implemented. A series of experiments were conducted to verify the close alignment between the theoretical results and the practical outcomes observed in concrete examples. The experiments were carried out on small-sized problems ($N = 2^{24}$ and $N = 2^{32}$) to facilitate a large number of attacks and generate multiple sets of tables for the different variants. Larger simulations have then been made on space $N = 2^{42}$.

This section provides an overview of the tests performed to assess the success probability, memory requirement, precomputation time, and attack time of the implemented variants.

7.4.1.1. Success Probability

To evaluate the success probability of each variant, we generated multiple sets of tables for various success probabilities and configurations. For each variant, a large number of attacks were carried out using these tables, typically ranging from 100,000 to 1,000,000 attacks per configuration in order to obtain accurate results. The observed success probabilities were consistent with the theoretical predictions given by Equation (2.10) and Theorems 6.1, and 7.1, with differences below 0.1%.

7.4.1.2. Precomputation Time

We assessed the precomputation time by generating tables for the three variants: RT, DSRT, and ASRT, with a fixed number of filters (typically around 20 which is a fair balance).

Additionally, we conducted tests on smaller spaces ($N = 2^{32}$) using one filter per column. We do not conduct tests using one filter per column on bigger spaces because as the computations are distributed, it costs too much time to filter in every column when considering bigger spaces. Indeed, for larger spaces, employing a filter in every column can achieve the theoretical lower bound in terms of the number of hash operations, but this approach considerably slows down the real time process due to the filtration time becoming predominant over the hashing time, as discussed in Chapter 4.

Our experimental results demonstrate that the precomputation time with one filter per column ($P_{\min}$) closely aligns with the theoretical predictions, with a maximum difference of less than 0.1%. Moreover, the experiments demonstrate that employing approximately 20 filters (including those used for steps) results in precomputation times that closely approach the theoretical lower bound, given by Equation (7.5), for all variants.

7.4.1.3. Memory Requirements

We did not implement point compression for memory testing since compression is independent from the variants chosen. Instead, we adapted the original formula provided in [AC13] for each variant. We tested the values involved in these formulas and verified that we obtained the expected numbers of EPs and SPs according to the theory. When applying the formulas to our experimental results versus the theoretical number of SPs and EPs to store, the differences were smaller than 0.05% across all tested configurations.

7.4.1.4. Attack Phase

The attack phase was tested in a manner analogous to the success probability evaluation. Tables were generated for various configurations and target success probabilities, followed by conducting a substantial number of attacks for each variant and configuration (typically between 100 000 and 1 000 000 attacks to ensure the accuracy of the average attack time measured). The average attack time closely adhered to the theoretical predictions, with a difference of less than 0.8% for all variants. The results were well distributed around the theoretical mean, with no significant difference based on the configuration used. It is worth noting, however, that the variance of DSRT attack time is larger than for RTs, and the variance of ASRTs is slightly larger than the variance of DSRT. This is further discussed in Section 7.5.

7.4.2. Comparison Methodologies

In Section 7.4.3, the precomputation and attack times of the RT, DSRT, and ASRT variants are compared, using the same targeted memory and targeted success probability. This approach allows for an evaluation of RT, DSRT and ASRT in a manner consistent with the one used in Chapter 6. Furthermore, comparing the variants at fixed success probability and memory settings highlights the trade-off between precomputation and attack times for each case.

This section outlines the evaluation process for the precomputation time, attack time, memory, and success probability of each variant, and provides justifications for the chosen methodology.

7.4.2.1. Precomputation Time

We chose $P_{\min}$ for comparison due to several reasons. Firstly, our tests demonstrated that filter usage gives results near the theoretical lower bound for RT, DSRT, and ASRT, thus offering a suitable comparison basis. Secondly, to perform the evaluation of tens of thousands of configurations, we favored $P_{\min}$ over a more time-consuming filter optimization evaluation. Finally, the use of $P_{\min}$ avoids any bias that could arise from selecting filter-based comparisons and, as discussed in Section 7.4.1.2, $P_{\min}$ approximates filter results for all variants.

7.4.2.2. Memory

For memory requirements, we could have used delta encoding. However, we chose the memory lower bound, which closely approximates (within 1%) the delta encoding, but offers simpler associated formulas. To ensure no bias towards DSRT or ASRT over RT, each step of DSRT and ASRT was treated as a separate table for memory computation which tends to favor RTs over DSRTs and ASRTs. This is fair as DSRT and ASRT share similar step-by-step storage methodologies. We also checked that the difference between delta encoding applied to ASRT and DSRT and their minimal memory lower bound remains under 0.7%, giving us the confidence to use the memory lower bound for comparison.

7.4.2.3. Success Probability and Attack Time

To compare the success probability and attack time of each variant, we simply applied Equation (2.11), and Theorems 6.1, 7.1, 6.9, 6.10, and 7.11. As mentioned in Section 7.4.1, the success probabilities and attack times estimated using these formulas closely align with the success probabilities and attack time obtained in practice for each variant.

7.4.3. Results

7.4.3.1. Parameters

The parameters chosen for the comparison are $N = 2^{42}$, which allows for an easy comparison with the results from [ACLA21, AC17, AC13], and from Chapter 6, and an arbitrary memory $M = 32GB$, representing a practical use case. The variants are then compared for various success probabilities, ranging from 80% to 99.95%. To maintain brevity, only results for some of the tested success probabilities are presented; however, the conclusions drawn are valid for all success probabilities.

> **Reminder:** A *configuration* is a list of parameters describing a set of either RTs, DSRTs, or ASRTs. The parameters of configurations for each variant are:
>
> **RTs:** $\{\ell, t, \alpha\}$,
>
> **DSRTs:** $\{\ell, t, \tau, \{s_1, \ldots, s_\tau\}, \alpha\}$,
>
> **ASRTs:** $\{\ell, t, \tau, \{s_1, \ldots, s_\tau\}, \{\alpha_0, \alpha_1, \ldots, \alpha_\tau\}\}$.

For each success probability, possible configurations for RTs, DSRTs, and ASRTs are computed. For RTs, the number of possible configurations for fixed memory and fixed success probability is limited, as the only variable left free is the number of tables. For DSRT,

in addition to the number of tables, the positions of the steps can vary according to the configurations, leading to many possible configurations as extensively explained in Chapter 6. When considering ASRT, the position of steps that remain free, and the number of elements to add in each step (determined by $\{\alpha_0, \alpha_1, \ldots, \alpha_\tau\}$) are additional parameters to set.

In total, the number of possible configurations for a given number of tables, given probability, and given memory is 1 for RT, bounded by $(t-1)^\tau$ for DSRT, and bounded by $(t-1)^\tau \times N^\tau$ for ASRT.

Given the number of possible configurations for DSRTs and ASRTs, the number of steps is set to four for DSRTs and to two for ASRTs. This choice is justified by the fact that using more than four steps for DSRTs does not significantly increase their performance, as stated in Chapter 6 using more than two steps for ASRT does not allow a significant gain compared to the computational cost needed to find possible configurations with three steps.

Following the method outlined in Chapter 6, we used Algorithm 3 to determine the DSRT configurations. For ASRT, we performed an exhaustive search, adjusting the steps for each α and t. We varied both t and the columns $\{s_1, s_2\}$ in steps of 100 columns, α_0 in steps of 0.003, α_1 in steps of 0.002, and α_2 in steps of 0.001. This strategy offers a balance between precision and computational efficiency in discovering configurations. The step size for α_0 is larger than that for α_1, and the step size for α_1 is larger than that for α_2. This is due to the fact that α_0, the maximality factor of the leftmost step, is associated with longer chains. Therefore, the impact of changes of α_0 is less pronounced in the resulting number of chains. This phenomenon, where the number of surviving chains quickly declines before stabilizing, is illustrated in Figure 3.1 of Chapter 3. The same conclusion holds for choosing the step variation of α_1 larger than those for α_2.

7.4.3.2. Figures Interpretation

Figures 7.3a 7.3b, 7.3c depict, for various success probabilities, the attack time of RTs, DSRTs and ASRTs best configurations according to their precomputation time, while Table 7.1, presents some interesting results. The attack time and precomputation time are expressed in the number of hashes to perform. The precomputation time is obtained by computing the corresponding minimal precomputation time P_{min} of each configuration either RTs, DSRT or ASRTs. The attack time is the average attack time for each configuration of each variant.

For each variant the *best configurations* are the configurations for which there is no existing configuration that is better both in precomputation and in attack.

In each plot, ASRT configurations are represented by red dots points, DSRT configurations are denoted by green dots points, and RT configurations are indicated by orange dots points. The black line signifies the *optimal configurations* among all the configurations of each variant.

7.4.3.3. ASRTs Versus RTs

In the following sections, the focus will be on comparing ASRT solely to DSRT, since as illustrated in Figures 7.3a, 7.3b, and 7.3c, there always exists an ASRT configuration superior to the RT configurations. Furthermore, similar to DSRT, the number of possible configurations for a given success probability and specified memory is higher when using ASRT than when using DSRT or RT. As a result, employing ASRTs allows for better trade-off between precomputation and attack compared to RTs.

The explanation of why ASRTs outperform RT is provided in Section 7.5.

7.4.3.4. ASRTs Versus DSRTS

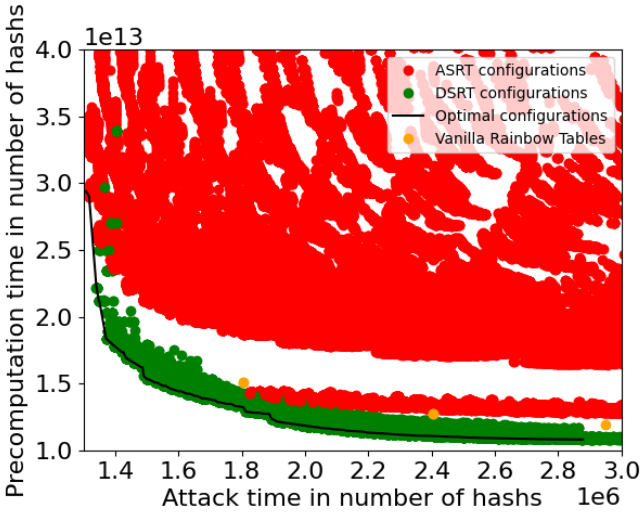
The subsequent paragraphs highlight noteworthy results from Table 7.1 and Figure 7.3, which help illustrate the differences between the use of ASRTs and DSRTs. We will discuss and provide interpretations for these results in Section 7.5. For simplicity, we present only three representative cases, but it should be noted that these results are applicable for other coverage and do not depend on the available memory, as discussed in Section 7.5.3.1.

Table 7.1. – Expected gain illustrated on several examples with ASRT and DSRTs. Precomputation and attack phase numbers are quantity of cryptographic operations.

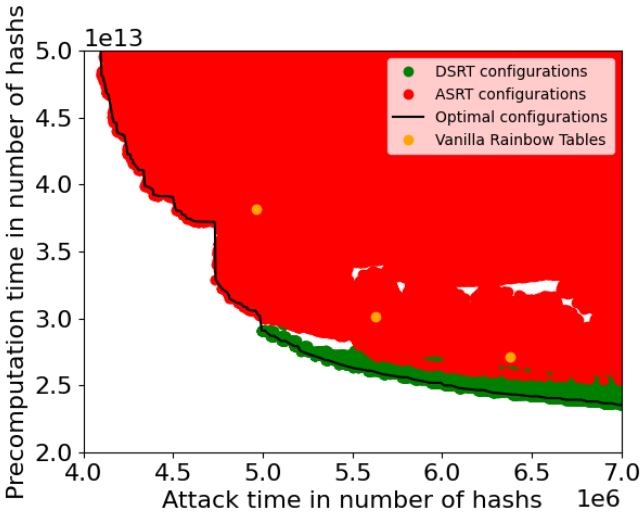
Success probability: 90%		
	Precomputation	Attack
1 ASRT	1.48×10^{13}	1.82×10^{6}
1 DSRT	1.21×10^{13}	
Gain	+22%	
1 ASRT	2.9×10^{13}	1.3×10^{6}
1 DSRT	2.2×10^{13}	1.35×10^{6}
Gain	+31%	−4%
Success probability: 99%		
	Precomputation	Attack
ASRT	5×10^{13}	4.11×10^{6}
DSRT	2.89×10^{13}	4.98×10^{6}
Gain	+73%	−17%
ASRT	3.23×10^{13}	4.7×10^{6}
DSRT	2.89×10^{13}	4.98×10^{6}
Gain	+12%	−6%
Success probability: 99.95%		
	Precomputation	Attack
ASRT	4.8×10^{13}	8.71×10^{6}
DSRT	7.2×10^{13}	
Gain	−33%	
ASRT	7.2×10^{13}	7.57×10^{6}
DSRT		8.66×10^{6}
Gain		−13%

Case 1: DSRTs more Efficient than ASRTs

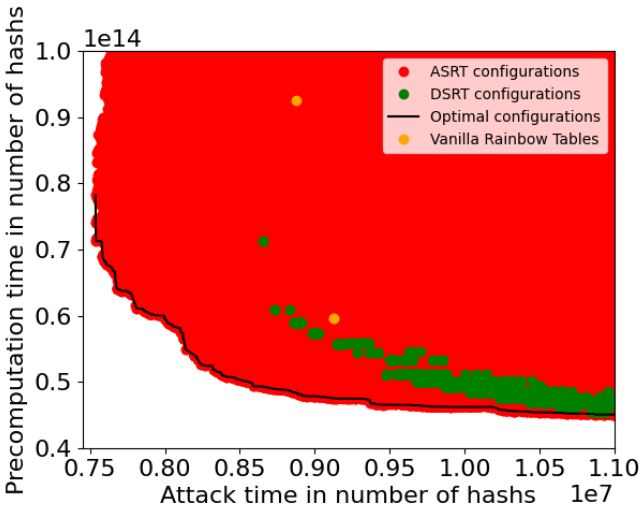
There are instances, especially when the targeted coverage is low enough to require only a single DSRT or ASRT, where DSRT configurations are more advantageous than ASRT configurations. This is demonstrated in Figure 7.3a and the first sub-Table of Table 7.1. For a given attack time achievable with DSRT, the corresponding ASRT needs considerably more precomputation time, rendering the variant less interesting. The configurations on the left of Figure 7.3a may be worthwhile in some cases, as they permit a reduction in attack time by 4% compared to DSRT, but at the cost of a 31% increase in precomputation time.



(a) Trade off between precomputation time and attack time **90%** of success, $N = 2^{42}$ and a 31.99 GB memory.



(b) Trade off between precomputation time and attack time **99%** of success, $N = 2^{42}$ and a 31.99 GB memory.



(c) Trade off between precomputation time and attack time **99.95%** of success, $N = 2^{42}$ and a 31.99 GB memory

Figure 7.3. – Trade-off between precomputation and attack.

Case 2: DSRTs and ASRTs Efficient

Figure 7.3b illustrates a typical scenario where ASRT may be preferred over DSRT if the attack time is the most important factor for the attacker. ASRT configurations achieve nearly identical trade-offs as the fastest DSRT configurations, and additionally offer a range of faster attack configurations at the cost of increased precomputation time. For instance, compared to the fastest DSRT configurations, it is possible to reach trade-offs 6% quicker in attack but requiring 12% additional precomputation time, or trade-offs that are 17% faster in attack at the expense of a 73% increase in precomputation time compared to DSRT (only 24% slower than the fastest RT configuration).

These results are observable for different coverage values greater than 97%. When targeting coverage higher than 99.5%, ASRT configurations outperform DSRT configurations.

7.4.3.5. Case 3: ASRTs more Efficient than DSRTs

For high targeted coverage, typically coverage requiring three or more tables, ASRTs outperform DSRTs. Figure 7.3c presents results for a common case discussed in the literature: the use of four quasi-maximal vanilla RTs, which allows to achieve a coverage of 99.95%.

As depicted in Figure 7.3c and Table 6.1, the optimal configurations are all ASRT configurations. Compared to the fastest DSRT configuration, an ASRT configuration can achieve the same attack time with 33% less precomputation time, or can reach a configuration 13% faster in attack for the same precomputation time.

7.5. Discussion

We initiate the discussion by comparing ASRT with RT exclusively in Section 7.5.1. This comparison facilitates the comprehension of the critical factors that render ASRT effective in a straightforward manner. Subsequently, in Section 7.5.2, we use the arguments developed in comparison with RT to contrast DSRT and ASRT, explaining why ASRT outperforms DSRT under certain circumstances and why ASRT is inferior to DSRT in others.

7.5.1. Comparison with RT

In this section, we compare two configurations: one corresponding to the RT method and the other one representing the ASRT method. In the particular example being discussed, the ASRT configuration overpass the RT configuration both in precomputation and attack efficiency.

To simplify the explanation for the reader, we have opted to compare RT and ASRT using a representative example. Although the comparison uses a specific example, the insights presented can be generalized to a broad range of cases.

Figure 7.4 illustrates a Rainbow matrix (depicted in green) and its corresponding ASRM. For the sake of clarity, this figure is constructed for a small space ($N = 2^{24}$), but the density of matrices remain consistent for larger spaces. Both configurations aim for a coverage of 98% and with the same memory.

Precomputation

> **Reminder:** The value $m_{c,c'}$ depict the number of surviving chains in column c' that starts in column c or before.
> It follows that the value m_{s_i,s_i}, is the number of chains considered in column s_i starting in column s_i itself or before.

For the same coverage and memory, the initial m_0 considered for the RT is substantially larger than the m_{s_0,s_0} of the ASRT (almost four times larger). This observation holds for all significant configurations and constitutes a key factor exploited by both ASRT and DSRT. Selecting four times fewer elements at the beginning of precomputation does not lead to a reduction by a factor of four in the final number of elements obtained at the end of the precomputation, but slightly less than twice as many. It is important to note that when employing filtration, as is our case, choosing four times more elements at the beginning of the precomputation does not quadruple the precomputation time, but increases it slightly less than three time. The saved time can be effectively employed to add more elements later in the precomputation phase.

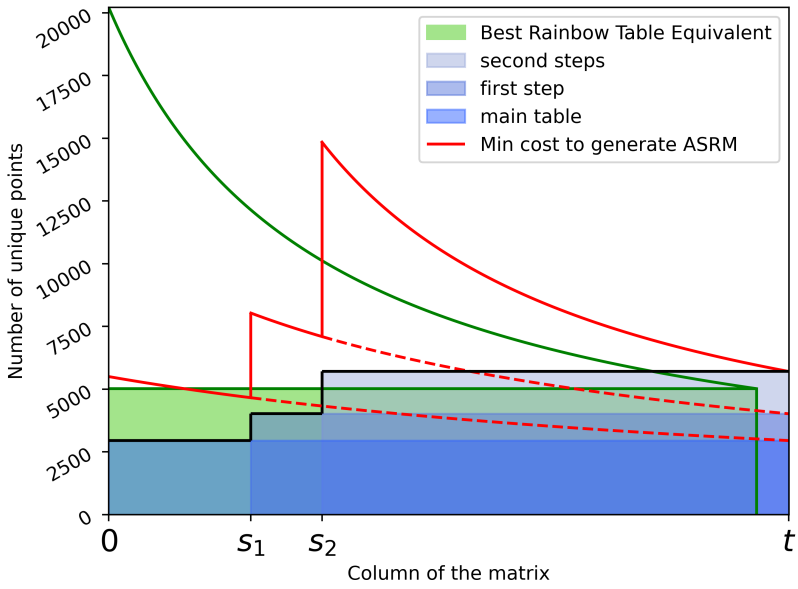


Figure 7.4. – ASRM with 2 steps versus the corresponding Vanilla Rainbow Table for same memory and same coverage (98%).

In column s_1, adding more elements to the ASRM still keeps the number of hashes required during the precomputation phase much lower for ASRM than for Rainbow matrix, and allows for increasing the number of elements in the final table to about 75% of the number of elements in the final Rainbow matrix.

The number of those SPs considered in column s_2 provides crucial insight into how the ASRT outperforms vanilla RT. The number of SPs considered in column s_2 is much larger than those considered in s_0 and s_1 and, at the same time, is significantly shifted to the right of the table. This enables to keep more elements in column t than in the case of vanilla

RT. Although the precomputation of the elements between s_2 and t requires more time than the precomputation of the RT chains in the corresponding columns, the time gained at the beginning of the precomputation by computing fewer chains significantly compensates for this extra time.

Memory A crucial point to understanding why the matrices depicted in Figure 7.4 occupy the same memory is based on acknowledging the impact of the $\log(m_0)$ and $\log(m_{s_0,s_0})$ factors in the memories formulas. Although they are logarithmic terms, which may give the impression of insignificance, the reality is quite the opposite. For instance, with RTs, the initial m_0 in the case of Figure 7.4 is four times larger than the number of elements obtained at the end, often even larger. This implies that each SP of the RT consumes about 15% more memory per element than the m_{s_0,s_0} elements considered in the ASRT variant. The RT SPs then take about 25% more memory per SP than the SP of the chains starting in s_1 and about 10% more memory per element than the SPs of chains starting in s_2.

In the final analysis, even though more SPs must be stored with the ASRT than with RT, and the fact that storing three batches of SPs slightly mitigates the decrease in memory required to store each SP, the amount of memory used to store the ASRT's SPs is roughly 12% lower than the amount of memory required to store the RT SPs.

This 12% memory saving is then used to "compensate" for storing the EPs, which is more optimized for the RTs due to: (a) Greater efficiency in compressing a single "large" batch of elements (as in RT EPs) as opposed to three "small" batches of elements (as in ASRT). (b) The fact that slightly fewer EPs need to be stored when using RT than when using ASRT.

Attack Even though the ASRT chains are longer than the RT chains, the attack phase is faster when using this particular ASRT configuration than when using the RT configuration.

Firstly, the chains of the ASRT starting in s_1 and s_2 are substantially shorter than those of the RT, and these chains account for about half of the ASRT chains. Thus, when performing a search in columns between s_2 and t, about half the time, the search will cost significantly less than the search in the RT. For the remaining half of the time, where a match occurs with chains starting in s_0 rather than s_1 or s_2, the cost is higher than when using RT, but less significantly. This is due to the fact that the difference between the lengths of the RT chains and the ASRT chains starting in s_0 is considerably lower than the difference in length between the RT chains and the ASRT chains starting in s_1 or s_2.

> **Reminder:** The value η_i, is the length of chains starting in column s_i.

Lastly, there are more chains between s_2 and t in the ASRM than in the Rainbow matrix. This shifts the average column in which the searched element is found, pushing it further to the right. Consequently, this decreases the number of searches before finding the searched element and increases the chances of matching with chains of length η_i and η_2 instead of t, thus increasing the chance of performing a search costing less operations.

7.5.2. ASRTs versus DSRTs

Takeaway: The key points to consider when deciding whether to use ASRT or DSRT are:

1. The larger the number of tables, the more efficient ASRT becomes, while the opposite is true for DSRT. It follows that ASRT performs best under high coverage while DSRT tends to excel at lower coverage.
2. ASRT is the preferred method when using quasi-maximal tables with $\alpha \geqslant 0.95$.
3. If the goal of the attacker is to minimize the attack time at any cost, ASRT should be used; if minimizing precomputation time is the main concern, DSRT should be selected when fewer than three tables are used.

The comparison between ASRT and DSRT is more complex than that between ASRTs and RTs. Nevertheless, the insights drawn from Section 7.5.1 regarding the superiority of ASRTs over RTs, and the discussion in Chapter 6 on the benefits of DSRT over RTs, can help elucidate the differences between DSRT and ASRTs.

To facilitate this comparison, we will separately address the comparison of ASRT and DSRT in terms of coverage, precomputation time, attack time, and memory.

7.5.2.1. Coverage

Quasi-Maximality Factor A key point when using ASRT is the need for continuously increasing quasi-maximality factors. In other words, for ASRT to be effective, the quasi-maximality factors should satisfy $\alpha_0 < \alpha_1 < ... < \alpha_s$. The intuition behind this is that the operational principle of ASRT is based on maintaining a large number of chains, ideally shorter ones, towards the right of the matrix.

One can perceive the right part of an ASRM as the segment that ensures the speed of the attack phase, and the left part as the segment that guarantees to reach targeted coverage.

To perform well in comparison with another variant, ASRT must therefore begin with an initial number of chains ms_0, s_0 significantly lower than that of the DSRTS. As the coverage increases, the quasi-maximality factor used tends to increase (to maintain an acceptable attack time by not adding an additional table), thus enhancing the use of ASRT.

However, at lower coverage, the maximality factor of RTs and DSRTs tends to be lower. This is particularly true for DSRTs, where the central idea is to generate matrices with lower quasi-maximality factors and to "compensate" for the waste of chains caused by the decrease in the quasi-maximality factor, with the steps.

Number of Tables DSRT variant tends to be less interesting, particularly regarding the attack time, as the number of tables increases. One of the factors contributing to DSRT better attack performance over RT is its general reliance on one fewer table than RT, which thereby reduces attack time. However, at higher coverage, the requirement for tables increases, and thus the benefit of using one less table diminishes, since each table has less impact when more tables are used.

Conversely, the most effective ASRT configurations can use the same number of tables as the best-performing RT configurations in the attack phase. Therefore, the attack performance of ASRT is not based mainly on the difference in the number of tables used.

Conclusion on Coverage ASRT outperforms DSRT in both attack and precomputation scenarios when an equal number of tables is used, or when sufficient tables are used such that the impact is insignificant, and if the quasi-maximality factors are sufficiently high. Under different circumstances, either DSRT performs better in both attack and precomputation, or ASRT is faster in attack but slower in precomputation. These last cases are further detailed in Sections 7.5.2.2 and 7.5.2.3.

7.5.2.2. Precomputation

> **Reminder:** While DSRT uses a single maximality factor, ASRT uses several of them. The maximality factor of DSRT determines the number of elements considered in column 0 of the DSRT matrix. Conversely, the ASRT maximality factors determine the number of chains to consider in column 0 and at each of the matrix's steps $\{s_1, \ldots, s_\tau\}$.

DSRT was designed with fast precomputation in mind. The critical element that facilitates the speed of DSRT precomputation is the choice of a lower maximality factor than RT at the beginning of precomputation, and compensate the resulting waste of chains through the use of steps.

Despite that, compared to ASRT, DSRT generally uses a higher maximality factor than ASRT initial maximality factor (while still being lower than RT). Consequently, the initial phase of precomputation is more costly when generating DSRT than ASRT. However, ASRT subsequent maximality factors will increase during the precomputation phase. In some cases, as soon as the first ASRT step is reached, the number of chains to compute becomes higher for ASRT than for DSRT. Ultimately, in a significant number of cases, the rising quasi-maximality factors of ASRT lead to a slower precomputation phase, as more chains need to be computed in the ASRT case after the first step.

When DSRT maximality factor is sufficiently high, the number of chains considered up until the final step of ASRT remains lower than the number of chains considered in the DSRT matrix. In the final step, the number of chains considered when using ASRT exceeds that of DSRT in all examples we have encountered. However, when the difference does not offset ASRT initial precomputation speed advantage, the precomputation time for the ASRT variant ends up being less than that of DSRT. On the contrary, if this is not the case, ASRT precomputation time exceeds that the one of DSRT.

7.5.2.3. Attack

Inherently, ASRTs tend to outperform DSRTs in terms of attack due to their key concept of maximizing the number of short chains on the right hand side of the matrix. As discussed in Section 7.5.2.2, this may come to the cost of a higher precomputation time.

Compared to DSRTs, ASRTs have more chains on the right of the matrix, with some parts of these chains being shorter than even the shortest DSRT chains. When ASRT shorter chains are not shorter than DSRT shortest chains, ASRT typically still comes with superior efficiency

in attack due to the slightly higher number of chains per table, and the lower cost of building the attack chain when using ASRT.

However, when ASRT shorter chains are not short enough, DSRT often surpasses ASRT in attack speed, largely due to the phenomenon discussed in Section 6.3 and illustrated in Section 6.5.4.1: when a false alarm is detected in a DSRT step, the attack chain is not rebuilt until the final table column. It acts as a sort of partial checkpoint (introduced in Section 2.6.1) and is one of the key points for maintaining the efficiency of DSRTs in attack.

In scenarios where ASRT does not have a sufficiently high last maximality-factor α_τ to guarantee a sufficient number of chains on the right side of the matrix, and where the chains of steps are not short enough, ASRT configurations are inferior to DSRT configurations in attack.

Nonetheless, at sufficiently high coverage (typically greater or equal to 90%), there always exists an ASRT configuration that outperforms the fastest DSRT in terms of attack time. However, this may come at the cost of a longer precomputation phase, particularly when three or fewer tables are used.

7.5.3. Memory

7.5.3.1. Memory Variation

It is essential to note that the relative performance of ASRT, DSRT, and RT do not depend on the memory available. By relative performance, we refer to the difference in performance between the variants, irrespective of the memory. Figures 7.5a and 7.5b illustrate the configurations of ASRTs, DSRTs, and RTs for the space $N = 2^{42}$, the coverage of 90%, and memory availabilities of 16GB and 32GB, respectively.

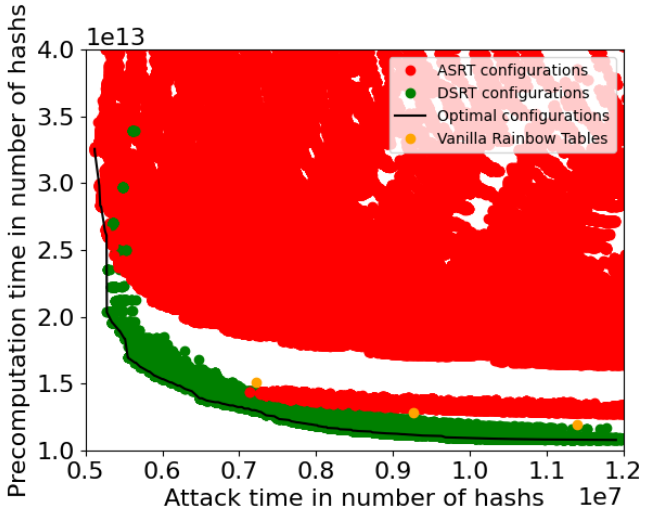
For a given coverage, it is clear that the precomputation time does not vary with the available memory, provided this memory allocation remains "reasonable". However, noticeable side effects might appear in the precomputation time when t is exceedingly low (high available memory), or when t is overly high (low available memory), resulting in an insufficient number of chains. Excluding these exceptional cases where precomputation time could marginally fluctuate with memory changes, the precomputation time does not depend on the available memory, since the "area" of the matrix to compute remains the same irrespective of the memory available. The only variation lies in the shape of the computed matrices, which may be more or less high or tall, depending on the memory available.

Regarding the attack phase, the results, though intuitive, are less clear-cut. Initially, when considering the RT, the results shown in Figures 7.5a and 7.5b align perfectly with expectations. As the memory is doubled while N remains the same and following the relation $T = N^2/M^2$, we can expect that by doubling the memory, we quarter the attack time, which is indeed the case.

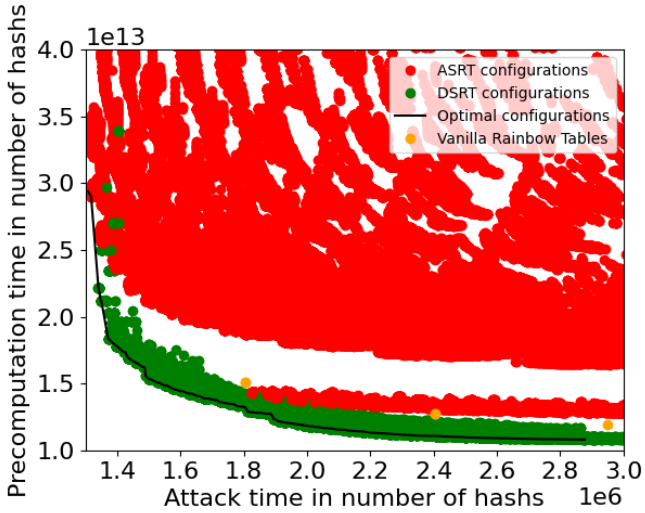
For the DSRT and ASRT variants, we do not provide proof, but we hypothesize that the behavior of these variants is equivalent to that of RTs. As will be discussed in Chapter 8, we presume that the relation $T = N^2/M^2$ may not be accurate, especially for lower coverages. Therefore, we postulate that ASRTs and DSRTs variants follow the $T = N^2/M^2$ relation for sufficiently high coverages.

We give some argument to justify our intuition:

- Numerous experiments performed on various search spaces, memory availabilities, and coverage consistently confirm this hypothesis. For instance, as presented in Figures 7.5a



(a)



(b)

Figure 7.5. – Trade-off between precomputation and attack.

and 7.5b, DSRTs and ASRTs follow this postulate and their attack time quarter when the memory is divided by two.

- The fact that RTs are, in essence, a special case of DSRT and ASRT (ASRT and DSRT formulas perfectly align with RT formulas when all steps are in the same column t.

- ASRT and DSRT can also be considered as multiple RTs sharing the same reduction functions. If each RT follows the relation $T = N^2/M^2$, we can expect the combined DSRT and ASRT to also adhere to this relation.

7.5.3.2. Access Memory

> **Reminder:** During a search, the attacker builds the so-called attack chain up to column t, and then checks for a match between the EPs and the attack chain. Searching for a match corresponds to performing one memory access.

On average, the number of memory accesses required for the ASRT attack is less than those needed for the DSRT and RT variants. This is primarily due to the greater number of chains in the right part of the table, which tends to be higher in ASRT than in the other two variants, thereby reducing the number of searches (and consequently the number of memory accesses). Nevertheless, the decrease in the number of memory accesses is not significant; it amounts to only a few percent, depending on the coverage and memory used.

7.5.4. Worst Attack Time

Like DSRTs, a drawback of ASRTs is that their worst-case attack time is longer than that of the worst-case RT attack. This is due to the fact that in a significant number of cases (almost all), the longest ASRT chain exceeds the length of the longest RT chain. Consequently, the attack time increases when the entire table must be searched through. This disadvantage can be mitigated since ASRTs are typically of interest in situations with high coverage, and thus, the worst-case scenario occurs very infrequently.

The most significant implication of this is an increase in the variability of the attack time. Similar to DSRTs, while the average attack time of ASRTs is shorter than that of RTs, it is more variable, which could be a disadvantage in some rare use cases.

7.6. Conclusion

This chapter introduces ASRTs, which demonstrate superior performance to both vanilla RTs and, under certain conditions, DSRTs. The core principle of ASRTs is the addition of chains at specified columns, referred to as steps. The strategy involves incrementally adding more chains at each step, with the goal of having more chains on the right side of the final matrix at the end of the precomputation phase. This approach implies the concept of *ascending stepped* Rainbow Tables and offers a two-fold benefit: improved matrix coverage and faster attack phase.

Owing to the larger parameter space, ASRTs afford a greater number of possible configurations compared to DSRTs. When targeting sufficiently high coverages, ASRTs outperform both DSRTs and RTs in terms of both attack and precomputation times, with the extent of gain primarily dependent on the targeted coverage.

Nonetheless, the addition of new chains during the precomputation phase can lead to increased precomputation times in scenarios involving less than three tables or targeting lower coverage. In these instances, gains in attack time often come at the cost of extended precomputation times compared to DSRTs.

In our practical experiments, when a typical coverage of 99.95% (a case frequently referenced in literature) is targeted, ASRTs can reduce precomputation time by 33% compared to DSRTs, for the same attack time, coverage, and memory. Alternatively, ASRTs can trim attack time by 13% for the same coverage, memory, and precomputation time. Compared to RTs, this represents a precomputation time reduction of 48% for the same attack time, coverage, and

memory, or an attack time reduction of 15% with a precomputation time that remains 24% lower.

When targeting lower coverage, ASRTs can reduce attack time compared to DSRTs (and thus RTs) at the cost of an increase in precomputation time. For instance, for a 99% coverage, ASRTs can decrease attack time by 17% at the cost of an additional 73% in precomputation time, or reduce attack time by 6% with a 12% increase in precomputation time.

Using both DSRTs and ASRTs instead of RTs can enable an attacker to substantially decrease both the attack and precomputation times of the used TMTOs, if the suitable variant is selected based on their targeted coverage and requirements. The potential for combining DSRTs and ASRTs in one variant remains a prospect for future work and could potentially facilitate a "best of both worlds" scenario.

8 Conclusion

Why would a PhD student never be a good meteorologist after having worked on Rainbow Tables?

Because she would always be predicting showers of hashes!

8.1. Thesis Contributions

In this thesis, we have provided significant insights and broadened the use-cases of TMTOs. One of the primary achievements was the formalization of quasi-maximality of RTs, notably introducing the quasi-maximality factor. It enables us to reach different coverages, beyond those reachable with maximal tables and has provided a metric to link formally m_0 with m_t. We also introduced the filtration method, a novel approach to optimize the precomputation phase of RTs. This method significantly reduces the precomputation time, particularly for quasi-maximal tables, and has triggered a fresh perspective on optimizing the precomputation process.

A key finding of our research was the identification of the precomputation phase as the existing bottleneck for RTs. This understanding led us to propose new RT variants that address and reduce the precomputation time.

We finally introduced two novel RT variants: Descending Stepped Rainbow Table (DSRT) and Ascending Stepped Rainbow Table (ASRT). The DSRT variant significantly improves the precomputation phase and yields a substantial reduction in attack time. On the contrary, the ASRT variant exhibits superior performance than DSRT when a sufficient number of tables is used. Both variants provide a more flexible approach compared to vanilla RTs, and improve the efficiency of the trade-off.

8.2. Research Takeaways

8.2.1. Precomputation

The research presented in this work has highlighted a new perspective on RTs and TMTOs in general. An aspect that had been largely under-explored until now is the importance of the precomputation time. This work has demonstrated that today, the precomputation time is a crucial parameter when using TMTOs, as critical as the other traditionally studied aspects, such as the attack time and memory.

The analysis and experiments conducted within this work show that the time needed to precompute sometimes outweighs the benefits provided by the TMTO algorithm, limiting its real-world applicability. This is especially true when dealing with large problem spaces, as in the case of modern cryptographic systems.

Hence, one of the research conclusion is that in order to fully understand and optimize TMTOs, it is necessary to move beyond the traditional time-memory dichotomy and to include the precomputation time as a core part of the TMTO trade-off. This will not only help to expand the applicability of TMTOs but also contribute to more accurate cost-benefit analyses of such algorithms.

8.2.2. A Whole Trade-Off

The discussion in this work brings to light an understanding of TMTOs as more than a simple trade-off between attack time and memory. Instead, it proposes viewing TMTOs through a more nuanced view by considering a trade-off among four factors: attack time, memory, precomputation time, and coverage.

Coverage, as precomputation time, is an aspect that this work identifies as being largely underestimated. Traditional approaches use maximal tables that thus have a fix coverage per table and thus neglect the flexibility and control that manipulating coverage can provide. Altering coverage can have significant impact on all other factors, including precomputation time, memory, and attack time. This interaction can result in substantial performance improvements or new approaches for algorithm optimization, opening up opportunities for future work.

8.3. Future Works

8.3.1. TMTOs Bound

The established bounds of TMTOs have been widely acknowledged, with the formula of $T = N^2/M^2$ identified as a characteristic constraint for the attack phase of TMTOs as outlined in [BBS06]. Nevertheless, throughout the process of this research, multiple elements have emerged, suggesting that these bounds should be reconsidered and refined.

A crucial aspect that requires revision is the set of assumptions underpinning the defined bounds. We believe that some assumptions do not hold for all possible TMTOs.

Moreover, the current bounds lack precision. Notably, the bound $\frac{1}{1024\ln(N)} \times \frac{N^2}{M^2}$, as formalized in [BBS06], is close to zero, which is not a realistic representation of actual TMTO performance. In our tests, the actual attack time tends to considerably exceed the computed bounds $\frac{1}{1024\ln(N)} \times \frac{N^2}{M^2}$, significantly impacting the usefulness of these bounds in assessing TMTO performance.

In addition to the above points, the applicability of these bounds across various coverage scenarios and table structures needs to be addressed. Specifically, the bound N^2/M^2 fails to hold for all coverages, particularly for lower coverages, and does not accurately reflect the performances of TMTOs when multiple tables are used or when tables are not close to maximal. In our tests, for instance, the attack time for RT with a coverage of 70% falls significantly below (approximately 50% below) the bound N^2/M^2, and is approximately 10^4 higher than the bound $\frac{1}{1024\ln(N)} \times \frac{N^2}{M^2}$, underscoring the insufficiency of the current bounds in accurately predicting TMTO performance.

It is also important to consider that the value M actually used in formulas represents the number of chains or the number of pairs stored. However, there are memory optimizations that can enhance storage efficiency (e.g., [ABC15, AC13]), and these optimizations operate differently depending on the maximality (closely tied to coverage) of the tables used. This

phenomenon is why we believe that comparing variants with equal chain numbers is not accurate, and should instead be done with equal memory lower bound. Focusing solely on the number of chains without considering maximality or coverage will inevitably result in inaccuracies.

In light of these observations, the scope for future research is vast, particularly in the redefinition and refinement of TMTO bounds. By addressing these bounds, a more comprehensive and accurate understanding of TMTOs can be achieved, which in turn could significantly enhance their practical applicability and performance.

8.3.2. Evaluation of TMTOs Across Varying Coverage

The investigation and comparison of the ASRT and DSRT variants during this research, particularly their configurations, prompted a deeper consideration of the coverage level in TMTO variants. Most of the prior works tend to focus primarily, if not exclusively, on high coverage scenarios, we suppose, due to a combination of the common use of maximum tables and the practical requirement for high coverage in passwords cracking.

Despite this, there is a noticeable lack of comparative analyses of different TMTOs variants at lower coverage. In particular, Rainbow Tables and Fuzzy Rainbow Tables have been acknowledged for their efficiency, with some researchers claiming them to be the most efficient TMTO variants. However, such comparisons have been traditionally conducted at high coverage. An intriguing question that arose during this work, albeit without accompanying evidence, is whether the Rainbow Tables and Fuzzy Rainbow Tables would maintain their superiority at lower coverage.

The relevance of this question is amplified by the emergence of applications requiring only a low success probability, such as, e.g., attacks against corporate employee passwords. Hence, a comparison of TMTOs variants at varying coverage, including low coverage, could be both insightful and impactful.

On a broader scale, an analysis comparing all significant TMTOs variants across coverages from 1% to 99.995%, according to their number of memory accesses, precomputation time, memory requirements, and attack time, would provide invaluable insights. Such a study could serve to conclusively identify the most efficient TMTO variant or, more likely, establish a framework to determine the most suitable variant according to specific attack purposes. This represents a significant and promising avenue for future research.

8.3.3. Distributed Rainbow Tables on GPU/FPGA

The research presented in [ACLA22] (Chapter 5) highlights that the precomputation phase is the primary bottleneck of RT particularly on CPUs. Consequently, the potential benefits of using GPU or FPGA are clear. A primary challenge, however, resides in the application of filters when using GPUs.

From a cost perspective, the use of multiple GPUs typically results in greater hashing speed compared to using FPGAs for the same investment. On the contrary, managing memory and facilitating effective communication is more challenging on GPUs. This brings to the forefront an interesting trade-off to consider. At lower coverage or when sufficient tables are used, the number of chains per table (and thus the number of merges) decreases, making the benefits of filtration less significant. Under such conditions, GPUs may offer a more efficient alternative compared to FPGAs and CPUs.

However, as we approach higher coverage, the situation becomes less straightforward. Our hypothesis is that even at high coverage, the use of filters on GPUs will remain advantageous when compared to not using them or relying on CPUs.

This raises several pertinent subjects for future research. The potential of FPGAs to outperform GPUs, given an equivalent cost investment, could be evaluated. A comparison of the speeds of FPGA or GPU usage with CPUs could reveal new insights. Exploring specific optimization techniques to further enhance the performance of GPUs and FPGAs is another promising subject.

8.3.4. Investigating Real-World Attacks

There are numerous potential applications for TMTOs that warrant further investigation. For instance, these could include attacks on cryptographic algorithms employed in communication networks However, we also believe there are additional practical attack scenarios that could potentially bring back the light on TMTOs.

Password cracking, in particular, is a field suitable for the application of TMTOs. Currently, password attacks primarily employ methods other than TMTOs. Yet, recent research on the topic [CFL22, MUS+16, USB+15] indicate that even when leveraging the most advanced techniques (e.g., machine learning, mask attacks, or Markov chain attacks), a substantial portion of attacked passwords remains unretrieved (between 30% and 50% of the database attacked). According to [CFL22], these passwords typically have a mask that appears only between one and six times. This suggests that these types of passwords are unlikely to be retrieved by methods other than TMTOs or brute-force attacks.

We believe that constructing TMTOs targeting specific forms of passwords (e.g., passphrase or pronounceable but random passwords) could be a fruitful line of investigation. These TMTOs will target the 30% to 50% of passwords that are not recovered by other techniques. This becomes especially relevant given the rise of password managers, which increasingly generate random passwords.

8.3.5. Other Variants

8.3.5.1. Regular Fuzzy Tables

Hellman tables and RTs could be regarded as two ends of a spectrum: while Hellman table use a single hash-reduction function for t columns, RTs employ t hash-reduction functions for the same number of columns. DSRTs serve as a generalized version of clean RTs, allowing chains to end at multiple columns. Similarly, one could propose a generalization of the RT variant that would use between 1 and t hash-reduction functions across t columns. The precise number of hash-reduction functions would depend on the targeted coverage, memory, and attack time. Using fewer than t hash-reduction functions could enhance the attack time, particularly if the same function is used on the matrix right hand side. However, this could also lead to a rapid decline in coverage, requiring an optimal trade-off.

Consider a case where t_h hash-reduction functions are used instead of t. Here, the question arises as to where to apply these functions. One possible approach could be to use the first $t_h - 1$ functions in the initial $t_h - 1$ columns, followed by a single function for the remaining $t - t_h + 1$ columns. This approach would substantially accelerate attack time on the matrix right hand side but could potentially cause a significant decrease in coverage.

Another approach is to use the t_h hash-reduction function, similar to the method used for the Fuzzy Rainbow Table, applied in batches across columns. However, without the DPs at the end of each chain, the attack efficiency could potentially be diminished.

8.3.5.2. Split Rainbow Tables

The concept of *Split Rainbow Tables* involves the *splitting* or unmerging of chains at certain columns during the precomputation process. This "split" consists in applying a different hash-reduction function on merged chains in a given column c, and then applying the same hash-reduction functions on the split chains until the precomputation end. The outcome is a matrix with an increased number of chains, considerably enhancing the attack time. A potential drawback here is the representation of duplicate chains prior to the split column c, which could be mitigated by searching within a chain batch across a selected number of columns left of the split.

Our preliminary experiments indicate that Split Rainbow Tables could outperform RTs. However, more extensive research and experimentation are required to confirm these results and determine whether this variant can significantly surpass existing ones, justifying its introduction.

8.3.5.3. Combined DASRT

A fusion of Descending and Ascending Stepped Rainbow Tables could potentially leads to a more efficient variant. The objective would be to recycle duplicate chains, particularly in the right parts of the matrix, while adding new chains across given columns during the precomputation phase.

The challenge with this combination lies in analyzing and identifying the best configurations. Given a specific memory and coverage, the possible configuration space would increase dramatically, with the main task being to find configurations for a specific memory and coverage.

Appendices A

A.1. Attack Phase with 1 step

Algorithm 1: Attack Phase (1-step)

Input : $y \in B : y = h(x^*) \in A$
Output : x^* (success) or $\perp$ (failure)

$stepsList \leftarrow [s, t]$
$EP \leftarrow [[EP_1 ... EP_{m_s - m_t}], [EP_{m_s - m_t + 1} ... EP_{m_s}]]$
$SP \leftarrow [[SP_1 ... SP_{m_s - m_t}], [SP_{m_s - m_t + 1} ... SP_{m_s}]]$
$c \leftarrow [s, t]$
$v \leftarrow [0 : \text{len}(c) - 1]$
$success \leftarrow$ False
while $c \neq [0, 0]$ **and** $success =$ False **do**
    $alarm \leftarrow$ False
    for $d = 0$ **to** $\text{len}(c) - 1$ **do**
        $v[d] \leftarrow \mu(c[d])$
    end
    $j \leftarrow c[v.\text{index}(\max(v))]$
    $i \leftarrow j$
    $x_i \leftarrow R_i(y)$
    while $i \leqslant t$ **and** $alarm =$ False **do**
        if $i \in stepsList$ **then**
            $index_i \leftarrow stepsList.\text{index}(i)$
            if $x_i \in EP[index_i]$ **then**
                $alarm \leftarrow$ True
                $x \leftarrow SP[EP[index_i].\text{index}(x_i)]$
                for $g = 1$ **to** $j - 1$ **do**
                    $x \leftarrow f_g(x)$
                end
                if $h(x) = y$ **then**
                    $success \leftarrow$ True
                end
            end
        end
        if $alarm =$ False **and** $success =$ False **then**
            $x_i \leftarrow f_i(x_i)$
            $i \leftarrow i + 1$
        end
    end
    if $success =$ False **then**
        $c[c.\text{index}(j)] \leftarrow c[c.\text{index}(j)] - 1$
    else
        return $(success, x)$
    end
end
return $(success, 0)$

A.2. Attack Phase with τ steps

Algorithm 2: Attack Phase with τ-Steps DSRTtable

input : $y \in B$ s.t. $y = h(x^*) \in A$
output : x^*, if it belongs to the DSRTmatrix

$stepsList \leftarrow [s_1, s_2, ...s_\tau, t]$
$EP \leftarrow [[EP_1...EP_{m_{s_1}-m_{s_2}}],$
$[EP_{m_{s_1}-m_{s_2}+1}...EP_{m_{s_1}-m_{s_3}}]...$
$[EP_{m_{s_1}-m_t+1}...EP_{m_{s_1}}]]$
$SP \leftarrow [SP_1...SP_{m_{s_1}-m_{s_2}}],$
$[SP_{m_{s_1}-m_{s_2}+1}...SP_{m_{s_1}-m_{s_3}}]...$
$[SP_{m_{s_1}-m_t+1}...SP_{m_{s_1}}]]$
$c \leftarrow [s_1, s_2, ..s_\tau, t]$
$v \leftarrow [0 : len(c) - 1]$
$success \leftarrow False$
while $c \neq [0, 0, .., 0]$ *and* $success = False$ **do**
  $Alarm \leftarrow False$
  for $d = 0$ *to* $len(c) - 1$ **do**
    $v[d] \leftarrow \mu(c[d])$ *#See Definition 6.3*
  end
  $j \leftarrow c[v.index(max(v))]$
  $i \leftarrow j$
  $x_i \leftarrow R_i(y)$
  while $i \leqslant t$ *and* $Alarm = False$ **do**
    if i *in* $stepsList$ **then**
      $index_i \leftarrow stepsList.index(i)$
      if x_i *in* $EP[index_i]$ **then**
        $Alarm \leftarrow True$
        $x \leftarrow SP[EP[index_i].index(x_i)]$
        for $g = 1$ *to* $g = j - 1$ **do**
          $x \leftarrow f_g(x)$
        end
        if $h(x) = y$ **then**
          $success \leftarrow True$
        end
      end
    end
    if $Alarm = False$ *and* $success = False$ **then**
      $x_i \leftarrow f_i(x_i)$
      $i \leftarrow i + 1$
    end
  end
  if $success = False$ **then**
    $c[c.index(j)] \leftarrow c[c.index(j)] - 1$
  else
    return $(success, x)$
  end
end
return $(success, 0)$

A.3. Configurations

Function *adjust* used in Algo. 3, takes a number of column t, and a list of *steps*, with a relative step position between 30% and 100% of t. It returns the positions of steps according to t.

Algorithm 3: Algorithm used to find valid configurations

input : The success probability *TargetProba*
The targeted memory
TargetMemory

output : A list *validConfig* of configurations that reaches the targeted success probability and memory

Procedure:
$t \leftarrow 1$
$\alpha \leftarrow 0.001$
for $ell = 1$ *to* $\ell = 6$ **do**
  for $nbSteps = 1$ *to* $nbSteps = 5$ **do**
    for *pos in all possible steps combinations* **do**
      $config \leftarrow [t, alpha, \ell, steps]$
      $end \leftarrow valid\_criteria(config)$
      while $end \neq 1$ **do**
        while $proba(config) < TargetProba$ **do**
          $\alpha \leftarrow \alpha + 0.001$
          $config \leftarrow [t, alpha, \ell, steps]$
        end
        while $memory(config) < 0.999 TargetMemory$ **do**
          $t \leftarrow t - 1$
          $steps \leftarrow adjust(t, pos)$
          $config \leftarrow [t, alpha, \ell, steps]$
        end
        while $memory(config) > 1.001 TargetMemory$ **do**
          $t \leftarrow t + 1$
          $steps \leftarrow adjust(t, pos)$
          $config \leftarrow [t, alpha, \ell, steps]$
        end
        $end \leftarrow valid\_criteria(config)$
      end
      $validConfig.append(config)$
    end
  end
end
return *validConfig*

Bibliography

[ABC15] Gildas Avoine, Adrien Bourgeois, and Xavier Carpent. Analysis of rainbow tables with fingerprints. In Ernest Foo and Douglas Stebila, editors, *Information Security and Privacy - 20th Australasian Conference, ACISP 2015, Brisbane, QLD, Australia, Proceedings*, volume 9144 of *Lecture Notes in Computer Science*, pages 356–374. Springer, 2015.

[AC13] Gildas Avoine and Xavier Carpent. Optimal storage for rainbow tables. In Hyang-Sook Lee and Dong-Guk Han, editors, *Information Security and Cryptology - ICISC 2013 - 16th International Conference, Seoul, Korea, Revised Selected Papers*, volume 8565 of *Lecture Notes in Computer Science*, pages 144–157. Springer, 2013.

[AC17] Gildas Avoine and Xavier Carpent. Heterogeneous rainbow table widths provide faster cryptanalyses. In Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi, editors, *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates*, pages 815–822. ACM, 2017.

[ACKT17] Gildas Avoine, Xavier Carpent, Barbara Kordy, and Florent Tardif. How to handle rainbow tables with external memory. In Josef Pieprzyk and Suriadi Suriadi, editors, *Information Security and Privacy - 22nd Australasian Conference, ACISP 2017, Auckland, New Zealand, Proceedings, Part I*, volume 10342 of *Lecture Notes in Computer Science*, pages 306–323. Springer, 2017.

[ACL15] Gildas Avoine, Xavier Carpent, and Cédric Lauradoux. Interleaving cryptanalytic time-memory trade-offs on non-uniform distributions. In Günther Pernul, Peter Y. A. Ryan, and Edgar R. Weippl, editors, *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, Proceedings, Part I*, volume 9326 of *Lecture Notes in Computer Science*, pages 165–184. Springer, 2015.

[ACLA21] Gildas Avoine, Xavier Carpent, and Diane Leblanc-Albarel. Precomputation for rainbow tables has never been so fast. In Elisa Bertino, Haya Shulman, and Michael Waidner, editors, *Computer Security – ESORICS 2021*, pages 215–234. Springer International Publishing, 2021.

[ACLA22] Gildas Avoine, Xavier Carpent, and Diane Leblanc-Albarel. Rainbow Tables: How Far Can CPU Go? *The Computer Journal*, page bxac147, 10 2022.

[ACLA23] Gildas Avoine, Xavier Carpent, and Diane Leblanc-Albarel. Stairway to rainbow. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '23, page 286–299, New York, NY, USA, 2023. Association for Computing Machinery.

[AJO05] Gildas Avoine, Pascal Junod, and Philippe Oechslin. Time-memory trade-offs: False alarm detection using checkpoints. In Subhamoy Maitra, C. E. Veni Madhavan, and Ramarathnam Venkatesan, editors, *Progress in Cryptology - INDOCRYPT 2005*, volume 3797 of *Lecture Notes in Computer Science*, pages 183–196. Springer, 2005.

[AJO08] Gildas Avoine, Pascal Junod, and Philippe Oechslin. Characterization and improvement of time-memory trade-off based on perfect tables. *ACM Trans. Inf. Syst. Secur.*, 11(4):17:1–17:22, 2008.

[Bab95] S.H. Babbage. Improved "exhaustive search" attacks on stream ciphers. In *European Convention on Security and Detection, 1995.*, pages 161–166, 1995.

[BBS06] Elad Barkan, Eli Biham, and Adi Shamir. Rigorous bounds on cryptanalytic time/memory tradeoffs. In Cynthia Dwork, editor, *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference, Santa Barbara, California, USA, Proceedings*, volume 4117 of *Lecture Notes in Computer Science*, pages 1–21. Springer, 2006.

[BD00] Eli Biham and Orr Dunkelman. Cryptanalysis of the A5/1 GSM stream cipher. In Bimal K. Roy and Eiji Okamoto, editors, *Progress in Cryptology - INDOCRYPT 2000, First International Conference in Cryptology in India, Calcutta, India, Proceedings*, volume 1977 of *Lecture Notes in Computer Science*, pages 43–51. Springer, 2000.

[BMS05] Alex Biryukov, Sourav Mukhopadhyay, and Palash Sarkar. Improved time-memory trade-offs with multiple data. In Bart Preneel and Stafford E. Tavares, editors, *Selected Areas in Cryptography, 12th International Workshop, SAC 2005, Kingston, ON, Canada, Revised Selected Papers*, volume 3897 of *Lecture Notes in Computer Science*, pages 110–127. Springer, 2005.

[BPV98] Johan Borst, Bart Preneel, and Joos Vandewalle. On the time-memory tradeoff between exhaustive key search and table precomputation. In *Symposium on Information Theory in the Benelux*, pages 111–118. TECHNISCHE UNIVERSITEIT DELFT, 1998.

[BS00] Alex Biryukov and Adi Shamir. Cryptanalytic time/memory/data tradeoffs for stream ciphers. In Tatsuaki Okamoto, editor, *Advances in Cryptology - ASIACRYPT 2000, 6th International Conference on the Theory and Application of Cryptology and Information Security, Kyoto, Japan, Proceedings*, volume 1976 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2000.

[BSW00] Alex Biryukov, Adi Shamir, and David Wagner. Real time cryptanalysis of a5/1 on a pc. In *International Workshop on Fast Software Encryption*, pages 1–18. Springer, 2000.

[CFL22] Alessia Michela Di Campi, Riccardo Focardi, and Flaminia L. Luccio. The revenge of password crackers: Automated training of password cracking tools. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen, and Weizhi Meng, editors, *Computer Security - ESORICS 2022 - 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, Proceedings, Part II*, volume 13555 of *Lecture Notes in Computer Science*, pages 317–336. Springer, 2022.

[CK08] Guanhan Chew and Khoongming Khoo. A general framework for guess-and-determine and time-memory-data trade-off attacks on stream ciphers. In Eduardo Fernández-Medina, Manu Malek, and Javier Hernando, editors, *SECRYPT 2008, Proceedings of the International Conference on Security and Cryptography, Porto, Portugal, SECRYPT is part of ICETE - The International Joint Conference on e-Business and Telecommunications*, pages 300–305. INSTICC Press, 2008.

[Den82] Dorothy E. Denning. *Cryptography and Data Security.* Addison-Wesley, 1982.

[DK08a] Orr Dunkelman and Nathan Keller. A new attack on the LEX stream cipher. In Josef Pieprzyk, editor, *Advances in Cryptology - ASIACRYPT 2008, 14th International Conference on the Theory and Application of Cryptology and Information Security, Melbourne, Australia. Proceedings*, volume 5350 of *Lecture Notes in Computer Science*, pages 539–556. Springer, 2008.

[DK08b] Orr Dunkelman and Nathan Keller. Treatment of the initial value in time-memory-data tradeoff attacks on stream ciphers. *Inf. Process. Lett.*, 107(5):133–137, 2008.

[DKRS21] Orr Dunkelman, Nathan Keller, Eyal Ronen, and Adi Shamir. Quantum time/memory/data tradeoff attacks. *IACR Cryptol. ePrint Arch.*, page 1561, 2021.

[EK15] Muhammed F. Esgin and Orhun Kara. Practical cryptanalysis of full sprout with TMD tradeoff attacks. In Orr Dunkelman and Liam Keliher, editors, *Selected Areas in Cryptography - SAC 2015 - 22nd International Conference, Sackville, NB, Canada, Revised Selected Papers*, volume 9566 of *Lecture Notes in Computer Science*, pages 67–85. Springer, 2015.

[Ers58] Andrei P. Ershov. On programming of arithmetic operations. *Communications of the ACM*, 1(8):3–6, 1958.

[FN91] Amos Fiat and Moni Naor. Rigorous time/space tradeoffs for inverting functions. In Cris Koutsougeras and Jeffrey Scott Vitter, editors, *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing, New Orleans, Louisiana, USA*, pages 534–541. ACM, 1991.

[Gol97] Jovan Dj Golić. Cryptanalysis of alleged a5 stream cipher. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 239–255. Springer, 1997.

[Hel80] Martin E. Hellman. A cryptanalytic time-memory trade-off. *IEEE Trans. Inf. Theory*, 26(4):401–406, 1980.

[HJK+08a] Jin Hong, Kyung Chul Jeong, Eun Young Kwon, In-Sok Lee, and Daegun Ma. Variants of the distinguished point method for cryptanalytic time memory trade-offs. In Liqun Chen, Yi Mu, and Willy Susilo, editors, *Information Security Practice and Experience, 4th International Conference, ISPEC 2008, Sydney, Australia, Proceedings*, volume 4991 of *Lecture Notes in Computer Science*, pages 131–145. Springer, 2008.

[HJK+08b] Jin Hong, Kyung Chul Jeong, Eun Young Kwon, In-Sok Lee, and Daegun Ma. Variants of the distinguished point method for cryptanalytic time memory trade-offs (full version). *IACR Cryptol. ePrint Arch.*, page 54, 2008.

[HJMM08] Martin Hell, Thomas Johansson, Alexander Maximov, and Willi Meier. The grain family of stream ciphers. 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 179–190. Springer, 2008.

[HK05] Jin Hong and Woo-Hwan Kim. Tmd-tradeoff and state entropy loss considerations of streamcipher MICKEY. In Subhamoy Maitra, C. E. Veni Madhavan, and Ramarathnam Venkatesan, editors, *Progress in Cryptology - INDOCRYPT 2005, 6th International Conference on Cryptology in India, Bangalore, India, Proceedings*, volume 3797 of *Lecture Notes in Computer Science*, pages 169–182. Springer, 2005.

[HKR83] Martin E. Hellman, Ehud D. Karnin, and Justin M. Reyneri. On the necessity of cryptanalytic exhaustive search. *SIGACT News*, 15(1):40–44, 1983.

[HLM11] Jin Hong, Ga Won Lee, and Daegun Ma. Analysis of the parallel distinguished point tradeoff. In Daniel J. Bernstein and Sanjit Chatterjee, editors, *Progress in Cryptology - INDOCRYPT 2011 - 12th International Conference on Cryptology in India, Chennai, India, 2011. Proceedings*, volume 7107 of *Lecture Notes in Computer Science*, pages 161–180. Springer, 2011.

[HM13] Jin Hong and Sunghwan Moon. A comparison of cryptanalytic tradeoff algorithms. *J. Cryptol.*, 26(4):559–637, 2013.

[Hoc09] Yaacov Zvi Hoch. *Security analysis of generic iterated hash functions.* PhD thesis, 2009.

[Hon10] Jin Hong. The cost of false alarms in hellman and rainbow tradeoffs. *Designs, Codes and Cryptography*, 57:293–327, 2010.

[Hon16] Jin Hong. Perfect rainbow tradeoff with checkpoints revisited. *PLoS ONE*, 11(11), 2016.

[KCGL09] Khoongming Khoo, Guanhan Chew, Guang Gong, and Hian-Kiat Lee. Time-memory-data trade-off attack on stream ciphers based on maiorana-mcfarland functions. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 92-A(1):11–21, 2009.

[KGL06] Khoongming Khoo, Guang Gong, and Hian-Kiat Lee. The rainbow attack on stream ciphers based on maiorana-mcfarland functions. In Jianying Zhou, Moti

Yung, and Feng Bao, editors, *Applied Cryptography and Network Security, 4th International Conference, ACNS 2006, Singapore, Proceedings*, volume 3989 of *Lecture Notes in Computer Science*, pages 194–209, 2006.

[KH13] Byoung-Il Kim and Jin Hong. Analysis of the non-perfect table fuzzy rainbow tradeoff. In Colin Boyd and Leonie Simpson, editors, *Information Security and Privacy - 18th Australasian Conference, ACISP 2013, Brisbane, Australia. Proceedings*, volume 7959 of *Lecture Notes in Computer Science*, pages 347–362. Springer, 2013.

[KH14] Byoung-Il Kim and Jin Hong. Analysis of the perfect table fuzzy rainbow tradeoff. *J. Appl. Math.*, 2014:765394:1–765394:19, 2014.

[KHP13] Jung Woo Kim, Jin Hong, and Kunsoo Park. Analysis of the rainbow tradeoff algorithm used in practice. *IACR Cryptol. ePrint Arch.*, page 591, 2013.

[KM96] Koji Kusuda and Tsutomu Matsumoto. Optimization of time-memory trade-off cryptanalysis and its application to des, feal-32, and skipjuck. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 79(1):35–48, 1996.

[Knu63] Donald E Knuth. Notes on "open" addressing. *Unpublished memorandum*, pages 11–97, 1963.

[KSH+12] Jung Woo Kim, Jungjoo Seo, Jin Hong, Kunsoo Park, and Sung-Ryul Kim. High-speed parallel implementations of the rainbow method in a heterogeneous system. In Steven D. Galbraith and Mridul Nandi, editors, *Progress in Cryptology - INDOCRYPT 2012, 13th International Conference on Cryptology in India, Kolkata, India. Proceedings*, volume 7668 of *Lecture Notes in Computer Science*, pages 303–316. Springer, 2012.

[KSH+15] Jung Woo Kim, Jungjoo Seo, Jin Hong, Kunsoo Park, and Sung-Ryul Kim. High-speed parallel implementations of the rainbow method based on perfect tables in a heterogeneous system. *Softw. Pract. Exp.*, 45(6):837–855, 2015.

[LH16a] Ga Won Lee and Jin Hong. Comparison of perfect table cryptanalytic tradeoff algorithms. *Designs, Codes and Cryptography*, 80(3):473–523, Sep 2016.

[LH16b] Ga Won Lee and Jin Hong. Comparison of perfect table cryptanalytic tradeoff algorithms. *Designs, Codes and Cryptography*, 80(3):473–523, Sep 2016.

[Li16] Zhen Li. Optimization of rainbow tables for practically cracking gsm a5/1 based on validated success rate modeling. In *Topics in Cryptology-CT-RSA 2016: The Cryptographers' Track at the RSA Conference 2016, San Francisco, CA, USA, Proceedings*, pages 359–377. Springer, 2016.

[LLH15] Jiqiang Lu, Zhen Li, and Matt Henricksen. Time-memory trade-off attack on the GSM A5/1 stream cipher using commodity GPGPU. In Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis, editors, *Applied Cryptography and Network Security - 13th International Conference, ACNS 2015, New York, USA, Revised Selected Papers*, volume 9092 of *Lecture Notes in Computer Science*, pages 350–369. Springer, 2015.

[MBPV06] Nele Mentens, Lejla Batina, Bart Preneel, and Ingrid Verbauwhede. Time-memory trade-off attack on FPGA platforms: UNIX password cracking. In Koen Bertels, João M. P. Cardoso, and Stamatis Vassiliadis, editors, *Reconfigurable Computing: Architectures and Applications, Second International Workshop, ARC 2006, Delft, The Netherlands, Revised Selected Papers*, volume 3985 of *Lecture Notes in Computer Science*, pages 323–334. Springer, 2006.

[MC86] J Ian Munro and Pedro Celis. Techniques for collision resolution in hash tables with open addressing. In *Proceedings of 1986 ACM Fall joint computer conference*, pages 601–610, 1986.

[MFI07] Miodrag J. Mihaljevic, Marc P. C. Fossorier, and Hideki Imai. Security evaluation of certain broadcast encryption schemes employing a generalized time-memory-data trade-off. *IEEE Commun. Lett.*, 11(12):988–990, 2007.

[MH09] Daegun Ma and Jin Hong. Success probability of the hellman trade-off. *Inf. Process. Lett.*, 109(7):347–351, 2009.

[ML75] Ward Douglas Maurer and Ted G Lewis. Hash table methods. *ACM Computing Surveys (CSUR)*, 7(1):5–19, 1975.

[MUS+16] William Melicher, Blase Ur, Sean M. Segreti, Saranga Komanduri, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Fast, lean, and accurate: Modeling password guessability using neural networks. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, Austin, TX, USA, 2016*, pages 175–191. USENIX Association, 2016.

[Nas00] Stephen G. Nash. A survey of truncated-newton methods. volume 124, pages 45–59, 2000. Numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations.

[NLAB+21] Cyrius Nugier, Diane Leblanc-Albarel, Agathe Blaise, Simon Masson, Paul Huynh, and Yris Brice Wandji Piugie. An upcycling tokenization method for credit card numbers. In Sabrina De Capitani di Vimercati and Pierangela Samarati, editors, *Proceedings of the 18th International Conference on Security and Cryptography, SECRYPT 2021*, pages 15–25. SCITEPRESS, 2021.

[Oec03] Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 617–630. Springer, 2003.

[Pet57] W Wesley Peterson. Addressing for random-access storage. *IBM journal of Research and Development*, 1(2):130–146, 1957.

[Saa02] Markku-Juhani Olavi Saarinen. A time-memory tradeoff attack against LILI-128. In Joan Daemen and Vincent Rijmen, editors, *Fast Software Encryption, 9th International Workshop, FSE 2002, Leuven, Belgium, Revised Papers*, volume 2365 of *Lecture Notes in Computer Science*, pages 231–236. Springer, 2002.

[SRQL02] François-Xavier Standaert, Gaël Rouvroy, Jean-Jacques Quisquater, and Jean-Didier Legat. A time-memory tradeoff using distinguished points: New analysis & FPGA results. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*, pages 593–609. Springer, 2002.

[Sur00] A survey of truncated-newton methods. *Journal of Computational and Applied Mathematics*, 124(1):45–59, 2000. Numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations.

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley Educational, Boston, MA, 4 edition, March 2011.

[TO] Cedric Tissières and Philippe Oechslin. Objectif sécurité, ophcrack. Accessed: June 2023, https://ophcrack.sourceforge.io/.

[USB+15] Blase Ur, Sean M. Segreti, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Saranga Komanduri, Darya Kurilova, Michelle L. Mazurek, William Melicher, and Richard Shay. Measuring real-world accuracies and biases in modeling password guessability. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, Washington, D.C., USA, 2015*, pages 463–481. USENIX Association, 2015.

[Van22] Mathy Vanhoef. A time-memory trade-off attack on WPA3's SAE-PK. In Jason Paul Cruz and Naoto Yanai, editors, *APKC '22: Proceedings of the 9th ACM on ASIA Public-Key Cryptography Workshop, Nagasaki, Japan*, pages 27–37. ACM, 2022.

[vdBP13] Fabian van den Broek and Erik Poll. A comparison of time-memory trade-off attacks on stream ciphers. In Amr M. Youssef, Abderrahmane Nitaj, and Aboul Ella Hassanien, editors, *Progress in Cryptology - AFRICACRYPT 2013, 6th International Conference on Cryptology in Africa, Cairo, Egypt. Proceedings*, volume 7918 of *Lecture Notes in Computer Science*, pages 406–423. Springer, 2013.

[VGB12] Roel Verdult, Flavio D. Garcia, and Josep Balasch. Gone in 360 seconds: Hijacking with hitag2. In Tadayoshi Kohno, editor, *Proceedings of the 21th NIX Security Symposium, Bellevue, WA, USA*, pages 237–252. USENIX Association, 2012.

[VMG+13] Roel Verdult, Wei Meng, Flavio D Garcia, Dan Doozan, Baris Ege, William Enck, Alex C Snoeren, Giovanni Vigna, Tao Xie, Nick Feamster, et al. Dismantling megamos crypto: Wirelessly lockpicking a vehicle immobilizer. In *22nd NIX Security Symposium*, pages 687–702, 2013.

[Wie04] Michael J. Wiener. The full cost of cryptanalytic attacks. *J. Cryptol.*, 17(2):105–124, 2004.

[WL13] Wenhao Wang and Dongdai Lin. Analysis of multiple checkpoints in non-perfect and perfect rainbow tradeoff revisited. In Sihan Qing, Jianying Zhou, and Dongmei Liu, editors, *Information and Communications Security - 15th International Conference, ICICS 2013, Beijing, China. Proceedings*, volume 8233 of *Lecture Notes in Computer Science*, pages 288–301. Springer, 2013.

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

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Titre : Compromis Temps-Mémoire Cryptanalytique

Mots clés : Compromis Temps-Mémoire, Table Arc-en-Ciel, Tables Escaliers, Précalculs

Résumé : Cette thèse analyse les améliorations des Compromis Temps-Mémoire Cryptanalytiques (TMTO), avec un accent sur les tables arc-en-ciel (RT). Elle revisite la phase de précalcul des RT, une étape souvent délaissée, en proposant des avancées notables. Une contribution majeure est la méthode de filtration, qui réduit le temps de précalcul des RT, en particulier celles dites parfaites ou propres, permettant de diminuer ce temps par un facteur de 6 pour des tables quasi-maximales. La thèse explore également la distribution de la phase de précalcul utilisant la méthode de filtration et fournit une comparaison du temps de précalcul des TMTO sur divers environnements. Ce travail révèle que le goulet d'étranglement actuel des TMTO sont les précalculs, et non le temps d'attaque ou la mémoire. Par ailleurs, elle présente deux nouvelles variantes des RT : les tables escaliers descendantes et ascendantes. Ces deux variantes, à couverture et mémoire fixes, surpassent les RT en précalcul et en attaque. Notamment, la première permet de réduire significativement le temps de précalcul par rapport aux RT avec filtration, tandis que la seconde permet un gain significatifs pour les tables quasi-maximales. La thèse conclut que les TMTO doivent être envisagés comme un compromis entre quatre facteurs : temps de précalcul, temps d'attaque, mémoire et couverture, ouvrant la voie à une optimisation future des TMTO.

Title: Cryptanalytic Time-Memory Trade-off

Keywords: Time-Memory Trade-off, Rainbow Tables, Stepped Tables, Precomputation

Abstract: This manuscript analyzes enhancements to Time-Memory Trade-Off (TMTO) techniques, focusing on the Rainbow Tables (RTs) variant. It revisits the often-overlooked precomputation phase of RTs, proposing significant advances. A key contribution is the filtration method, which reduces precomputation time for RTs, particularly for so-called 'perfect' or 'clean' RTs, allowing this time to be reduced by a factor of 6 for quasi-maximal tables. The thesis also investigates the distribution of the precomputation phase using the filtration method, providing a comparison of TMTO precomputation time across different environments. This research reveals that the current bottleneck for TMTOs lies within the precomputation phase, not the attack time or memory for TMTO storage. Moreover, it introduces two new variants of RTs: Descending and Ascending Stepped Rainbow Tables. These variants, with fixed coverage and memory, outperform vanilla RTs in precomputation and attack. Notably, the first allows to significantly reduce the precomputation time compared to RTs with filtration, while the Ascending variant offers significant gains for quasi-maximal tables. The thesis concludes that TMTOs should be viewed as a trade-off among four factors: precomputation time, attack time, memory, and coverage, paving the way for future TMTO optimization.