Practical Considerations on Cryptanalytic Time-Memory Trade-Offs

Florent Tardif

▶ To cite this version:

Florent Tardif. Practical Considerations on Cryptanalytic Time-Memory Trade-Offs. Cryptography and Security [cs.CR]. Université Rennes 1, 2019. English. NNT : . tel-02882523

HAL Id: tel-02882523
https://hal.science/tel-02882523v1

Submitted on 26 Jun 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1
COMUE UNIVERSITÉ BRETAGNE LOIRE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies
de l'Information et de la Communication
Spécialité : Informatique

Par

Florent TARDIF

Practical Considerations on Cryptanalytic Time-Memory Trade-Offs

Thèse présentée et soutenue à Rennes, le 26 novembre 2019
Unité de recherche : IRISA

Rapporteurs avant soutenance :

Dr Olivier HEEN	Technicolor, France
Prof. Julio HERNANDEZ-CASTRO	University of Kent, Royaume-Uni

Composition du Jury :

Président :	Prof. Pierre-Alain FOUQUE	IRISA et Université de Rennes 1, France
Examinateurs :	Dr Xavier CARPENT	University of California San Diego, États-Unis
	Prof. Aurélien FRANCILLON	EURECOM, France
	Dr Olivier HEEN	Technicolor, France
	Prof. Julio HERNANDEZ-CASTRO	University of Kent, Royaume-Uni
Directeur de thèse :	Prof. Gildas AVOINE	IRISA et INSA Rennes, France
Co-encadrante :	Dr Barbara FILA	IRISA et INSA Rennes, France

Practical Considerations on Cryptanalytic Time-Memory Trade-Offs

Florent TARDIF

Résumé en français

Motivation

Sur les dernières décennies, on peut observer une dépendance encore et toujours plus accrue envers les ordinateurs, qui nous fournissent les informations dont nous avons besoin, tout en gardant à portée de main une quantité grandissante de nos données personnelles : photos, communications électroniques, informations bancaires, données médicales, etc... Pour sécuriser ces données, il est nécessaire d'utiliser l'équivalent numérique des clefs du monde réel, pour restreindre l'accès aux données aux seules personnes autorisées. Plus généralement, ces clefs virtuelles sont utilisées partout sur internet pour sécuriser un grand nombre d'activités en ligne telles que le e-commerce, l'accès en ligne à la banque ou encore les messageries, instantanées ou non.

La cryptographie, véritable science du secret, fournit les moyens, sous forme d'algorithmes, dits de chiffrement, qui assurent la confidentialité des données pour les seuls possesseurs des clés. Ces derniers sont alors les seuls à pouvoir *déchiffrer* les données. La cryptographie se décompose en deux disciplines : d'un côté la cryptologie, domaine d'étude et de développement de nouvelles primitives cryptographiques, et de l'autre la cryptanalyse, dont le but est d'attaquer les primitives existantes. Cela consiste à rechercher des vulnérabilités dans l'algorithme qui permettent de *décrypter* les données chiffrées, c'est-à-dire y accéder sans connaissance de la clé. Historiquement, la cryptographie a été l'apanage des militaires, au point d'être considérée comme une arme de guerre, jusqu'en 2004 en France[1]. La cryptographie moderne est basée sur le principe de Kerckhoffs qui stipule qu'un système cryptographie doit rester sûr même si absolument tout en est connu (algorithme, paramètre) à l'exception de la clé utilisée pour chiffrer les données. C'est pourquoi les attaques cryptographiques se concentrent autant sur la recherche de clés.

Les algorithmes cryptographiques sont de deux types, ceux dont la clé de chiffrement est celle de déchiffrement, qui font partie de la cryptographie symétrique, et les systèmes à clé publique qui utilisent une clé différente pour les deux opérations, et qui appartiennent à la cryptographie asymétrique. Ces derniers sont souvent utilisés pour des situations plus complexes que le simple chiffrement de données, par exemple des protocoles d'échanges de clé sécurisés ou des schémas de signature. Nous nous concentrons sur le chiffrement de données, donc seule la cryptographie symétrique est considérée ici.

Dans un contexte cryptographique, une donnée non chiffrée est appelée *clair*, et on nomme *chiffré* la sortie de cette même donnée par l'algorithme de chiffrement pour une clé donnée. Un scénario courant de cryptanalyse est l'attaque à clair connu, qui correspond à la situation où l'attaquant possède un chiffré d'une donnée qu'il connaît, et où son but est de déterminer la clé qui a chiffré cette donnée.

[1] Article 30 de la loi du 21 juin 2004 pour la confiance dans l'économie numérique. « I. L'utilisation des moyens de cryptologie est libre. »

Afin de mener à bien cette attaque, une idée possible est de parcourir tout l'espace de toutes les clés possibles, c'est-à-dire essayer de chiffrer la donnée que l'attaquant possède (en fait le clair que l'on connaît, d'où le nom du scénario) avec toutes les clés possibles, jusqu'à obtenir un chiffré identique à celui que l'on possède. Cette méthode de recherche exhaustive de clé est souvent appelée attaque par force brute, traduction littérale de l'anglais "brute force attack". Cette attaque ne requiert qu'une quantité négligeable de mémoire, puisque les clés sont essayées à la volée. Dans une situation idéale, il n'existe pas d'attaque plus efficace que l'attaque par force brute contre une primitive cryptographique donnée. Si d'autres attaques moins coûteuses sont trouvées, on considère que la sécurité proposée par la primitive est diminuée. À un certain point, une primitive dont la sécurité est trop diminuée est considérée comme cassée et ne doit pas être utilisée. Lors de la conception d'une primitive cryptographique, l'un des objectifs est donc de s'immuniser contre toutes les attaques qui ont été recensées contre ce genre d'algorithme.

Un des outils utilisés pour attaquer les primitives cryptographiques sont les compromis temps-mémoire cryptanalytiques (TMTOs, de l'anglais *Time-Memory Trade-Offs*), qui permettent d'effectuer une recherche de clé plus rapidement qu'une attaque par force brute. Pour ce faire, ils reposent sur un précalcul dont le résultat est stocké en mémoire, et dont le coût est souvent élevé. Parfois, ce coût excède même le coût d'une attaque par force brute. L'utilité d'une telle technique est d'effectuer plusieurs attaques par force brute. Si le nombre d'attaques est élevé, le coût du précalcul est alors amorti. Plus particulièrement, les compromis temps-mémoire cryptanalytiques sont les plus utiles là où une recherche exhaustive est extrêmement coûteuse, mais pas impossible, et le stockage en mémoire de l'espace des clés n'est pas possible avec le matériel disponible.

Les compromis temps-mémoire cryptanalytiques ont été exploités dans divers domaines de la cryptographie symétrique. Leur usage pour la cryptanalyse des chiffrements par flot, qui sont des primitives cryptographiques destinées à fonctionner avec des flux de bits, plutôt que des quantités fixées de données, a été particulièrement prolifique, au point qu'ils font désormais partie de la boîte à outil des concepteurs d'algorithmes de chiffrement par flot. Des exemples d'application incluent par exemple le système de chiffrement des communications de téléphonie mobile A5/1, dont la cryptanalyse a pu être suffisament accélérée par les compromis temps-mémoire cryptanalytiques qu'il soit possible d'attaquer, et décrypter une communication en temps réel. D'autres algorithmes de chiffrement par flot ont également été cryptanalysés comme par exemple les candidats aux compétitions internationales de cryptographie, dont le but est de soumettre à la communauté des propositions de primitives qui sont attaquées par les experts du domaine afin de déterminer celles qui sont potentiellement les plus sûres. Les compromis temps-mémoire cryptanalytiques ont également eu du succès dans la cryptanalyse de chiffrements intégrés dans des puces de radio-identification, qui possèdent peu de capacités de calcul, ce qui implique une sécurité limitée. Les compromis temps-mémoire cryptanalytiques permettent d'accélérer des attaques qui, faites à vitesse normale, ne constitueraient pas un danger pour les appareils. De telles puces vulnérables ont été trouvées dans des systèmes de paiement de station essence, ou encore dans des appareil d'immobilisation à distance de véhicules. Néanmoins, une des applications les plus connues des compromis temps-mémoire cryptanalytiques est la récupération de mots de passe à partir d'empreintes cryptographiques produites par des fonctions de hachage. Nous nous intéressons plus particulièrement à cette application et reviendrons plus en détail sur le cassage de mot de passe dans la section suivante. Enfin, une autre application des compromis temps-mémoire cryptanalytiques est la mise en défaut d'un système de consensus alternatif à la preuve de travail telle qu'utilisée dans des cryptomonnaies comme le Bitcoin. Traditionnellement, le consensus est établi en proposant une solution à un problème difficile qui demande beaucoup de ressources de calculs. Cette méthode alternative propose, en lieu et place de calcul,

de s'engager à stocker une ressource arbitraire en mémoire et de le prouver en répondant rapidement à certaines opérations spécifiques. La quantité de données stockées agit comme métrique de l'engagement et donne d'autant plus de poids dans le consensus qu'elle est importante. Cependant, les compromis temps-mémoire cryptanalytiques permettent de réduire l'espace de stockage réellement utilisé, tout en limitant le temps d'accès aux données pour les preuves. De nouveaux systèmes de consensus, exploitant les compromis temps-mémoire cryptanalytiques eux-mêmes, ont été mis en place afin de résoudre ce problème.

Dans cette thèse, nous nous intéressons au fonctionnement des compromis temps-mémoire cryptanalytiques. Le travail présenté dans ce manuscrit est indépendant des différentes applications que l'on peut trouver aux compromis temps-mémoire cryptanalytiques et dont certaines viennent d'être listées. Cependant, connaître l'usage qu'il sera fait de la technique est souvent avantageux en terme d'optimisations. Il est plus facile d'établir des paramètres de fonctionnement adaptés si la taille du problème, par exemple la taille de l'espace des clés, ou la mémoire disponible de la machine ciblée, sont connues. C'est pourquoi nous choisissons de fournir un cas pratique pour notre étude, qui donne un aspect plus concret à notre propos. Nous nous concentrons donc plus particulièrement sur l'une des applications présentées : le cassage de mots de passe.

Cas d'étude : le cassage de mots de passe

Un grand nombre de systèmes sécurisés, et particulièrement les services en ligne, reposent sur une gestion de mots de passe pour assurer l'authentification. Les paramètres de connexion sont une cible de choix pour les attaquants. En effet, une fois en possession d'un mot de passe, et en l'absence de mécanismes de protections supplémentaires tels que le fingerprinting de l'utilisateur pour détecter une activité inhabituelle, l'accès de l'attaquant utilisant un mot de passe subtilisé est indiscernable d'un accès légitime. On pourrait considérer des techniques plus avancées telles que la reconnaissance faciale ou l'utilisation de l'empreinte digitale qui sont moins vulnérables au vol d'identifiants. En réalité, ces techniques biométriques peuvent faillir à cause par exemple de la déformation du visage ou du doigt, et ces systèmes ont souvent, en plus du système biométrique, un système d'authentification par mot de passe pour pallier aux situations où ce dernier ne fonctionnerait pas correctement, ce qui fait qu'une attaque sur le système de mots de passe a une portée plus conséquente et supérieure à celle attendue. Si l'on considère une telle attaque, une première idée est d'envisager l'énumération de mots de passe sur le service jusqu'à obtenir le bon. C'est cependant une stratégie peu viable en pratique, du fait de la latence induite par le réseau qui limite le nombre d'essais par seconde. Il faut garder à l'esprit que la performance d'une telle attaque effectuée en local, c'est à dire sur la même machine que la base de données qui valide les mots de passe, est telle qu'il est possible de l'effectuer en un temps qui est d'au moins un ordre de grandeur plus petit qu'une attaque en ligne. En plus de la latence du réseau, certains services ajoutent un délai artificiel entre deux essais consécutifs d'un mot de passe, pour rendre impossible, dans un temps raisonnable, l'essai d'un grand nombre de mots de passe. Il est donc plus avisé de considérer le cas local pour une attaque qui fonctionne en pratique.

Bien que fortement déconseillée, tant par la plupart des politiques de mots de passe en entreprises, que par les experts en sécurité, la réutilisation d'un même mot de passe pour différents services est malheureusement un phénomène récurrent. De fait, une stratégie efficace pour un attaquant est de se concentrer sur des services peu sécurisés afin de mettre la main sur des bases de données de mots de passe. L'idée est simplement d'associer une identité avec un mot de passe afin d'essayer ce même mot de passe sur les autres services, potentiellement plus sécurisés, que celui sur lequel la personne s'est enregistrée.

De telles bases de données de systèmes peu sûrs font régulièrement l'objet de « fuites » publiques, orchestrées par des groupes de hackers anonymes, et sont mises à disposition en téléchargement libre sur Internet. Le phénomène est suffisamment fréquent qu'il existe des sites web spécialisés dans le suivi des fuites de mots de passe, tels que la page « Have I Been Powned », du chercheur en sécurité Troy Hunt.

Dans les bases de données, les mots de passe ne sont pas toujours lisibles. Une technique classique pour le stockage de mots de passe est l'utilisation d'une empreinte issue d'une fonction de hachage cryptographique. L'idée est de permettre le stockage du mot de passe sans le révéler au service. A partir de sa connaissance du mot de passe, l'utilisateur peut générer une empreinte qui sera comparée à celle de la base de données. Cependant, obtenir le mot de passe à partir de l'empreinte est difficile étant donné que l'action de générer l'empreinte n'est en principe pas réversible. Il est néanmoins possible, moyennant un nombre souvent important d'essais, de parvenir à retrouver un mot de passe dont l'empreinte est égale à une empreinte donnée. Un procédé consistant à systématiquement retrouver un mot de passe à partir de son empreinte est appelé « cassage de mots de passe ». Il s'agit d'une activité indispensable pour assurer la sécurité par mots de passe d'un système : le meilleur moyen de savoir si les mots de passe d'utilisateurs sont résistants aux attaques connues est d'appliquer ces mêmes attaques sur la liste des mots de passe.

La plupart des serveurs de données, et plus particulièrement ceux qui sont accessibles depuis internet, sont sous la menace permanente d'attaquants. Ces derniers leur font subir en permanence des manœuvres hostiles, telles que des attaques par déni de service ou des tentatives d'intrusions, le but étant de mettre à jour une faille dans le système qui soit exploitable, afin de parvenir à une élévation de privilèges. Bien évidemment, les méthodes parfois très sophistiquées des attaquants, et au-delà de l'état de l'art documenté, sont un secret bien gardé. Sécuriser un système est donc un jeu du chat et de la souris entre les attaquants d'une part, et les spécialistes en sécurité de l'autre. Un des enjeux de la recherche dans le domaine de la sécurité est de se tenir à jour des techniques utilisées par les attaquants.

En documentant les attaques, qu'il s'agisse de leur méthodologie ou de leur efficacité, les attaquants peuvent ainsi informer la communauté du niveau de danger qui guette les systèmes d'information, et par la même, de leur possible fragilité. Ainsi les différents acteurs sont encouragés à mettre à jour leurs systèmes au vu de la menace, ce qu'ils seraient plus réticents à faire dans un cadre plus théorique. En effet, les publications théoriques se rapportant à la sécurité n'ont pas toujours l'impact souhaité, en particulier quand le coût d'une mise à jour où le temps nécessaire pour corriger une vulnérabilité dans un produit déployé est pris en compte. Il est donc primordial que les vulnérabilités aient une dimension pratique, pour susciter la réaction escomptée. Même dans le cas d'une vulnérabilité avérée, l'effet peut être atténué par le fait que les attaques sont coûteuses à mettre en place. Cependant, avec le temps, du fait de l'amélioration du matériel, il arrive un moment où les attaques au prix exorbitant du passé deviennent possibles avec du matériel peu coûteux, stade auquel il faut considérer que le système vulnérable ne remplit plus du tout son rôle.

Notons qu'il existe une technique qui peut protéger les mots de passe contre les compromis temps-mémoires : le *salage* de mot de passe. Il s'agit d'ajouter une donnée connue et non secrète à tous les mots de passe avant de générer leur empreinte. Si cette donnée est suffisamment longue, attaquer une base de données de mots de passe salés devient impossible de manière traditionnelle. Utiliser un compromis temps-mémoire cryptanalytique pour une telle base, en prenant en compte le sel, implique de refaire le précalcul, et donc annule un des intérêts de la technique qui consiste à proposer l'équivalent d'une attaque par force rapidement. Avec cette technique, leur rôle se réduit donc alors à fournir une estimation de la force d'un mot de passe non salé. Cependant, cette technique n'est pas encore utilisée partout,

comme le prouvent les différentes fuites de bases de données de mots de passe non salés, et il est de toute manière impossible de vérifier la technique utilisée pour stocker les mots de passe lors de la création d'un compte utilisateur sur un site web. Les compromis temps-mémoire cryptanalytiques ont donc encore leur place dans un contexte moderne, en tant qu'un des outils d'évalutation de la sécurité des mots de passe.

On peut définir deux cas d'usage courants pour les attaques par compromis temps-mémoire cryptanalytiques. Premièrement, quand il est nécessaire d'effectuer de nombreuses recherches exhaustives sur une même fonction à sens unique, il est sensé de recourir au précalcul de tables à partir du moment où ce dernier n'est pas plus coûteux que le nombre correspondant de recherches exhaustives, car les phases d'attaque en utilisant les tables sont négligeables par rapport à même une seule recherche exhaustive. Le deuxième cas d'usage commun est « l'attaque de la femme de ménage ». Il s'agit du cas de figure où l'attaquant a du temps pour préparer l'attaque, mais ne connaît pas a priori l'empreinte à inverser, bien qu'il souhaite l'inverser très rapidement. L'exemple classique consiste à considérer une cible qui séjourne à l'hôtel et qui s'absente de sa chambre brièvement tout en laissant son ordinateur portable. L'attaquant, usurpant l'identité de la femme de ménage, pénètre dans la chambre et dispose d'un temps limité pour son attaque. Pour tous les cas où une attaque par dictionnaire n'est pas suffisante, les compromis temps-mémoire cryptanalytiques peuvent être considérés.

Problème étudié

Avec les compromis temps-mémoire cryptanalytiques, il est possible de réduire arbitrairement le temps nécessaire pour effectuer une attaque par force brute. Ce n'est cependant possible qu'en payant le coût d'un précalcul, qui n'est effectué qu'une fois, mais qui requiert des ressources de calcul considérables. Le travail de cette thèse est d'améliorer les compromis temps-mémoire cryptanalytiques, en réduisant l'empreinte mémoire qu'il est nécessaire de stocker à la fin du précalcul. La question de recherche de ce manuscrit est la suivante : *Comment peut-on améliorer la complexité spatiale des compromis temps-mémoires ?* Une telle amélioration nous permettrait de s'attaquer à des problèmes plus importants et d'élargir le champ d'application des compromis temps-mémoire cryptanalytiques.

Les résultats de notre analyse nous ont menés aux trois principales contributions de cette thèse, décrites ci-après.

Mémoire externes Quand la taille du problème étudié est très grande, la mémoire nécessaire pour stocker les résultats du précalcul excède ce qui peut être stocké en RAM par un ordinateur. L'utilisation de mémoires externes, telles que les disques durs peut être envisagée. Nous proposons d'étudier ce problème, inédit dans la littérature, en évaluant l'impact de l'usage du disque sur un compromis temps-mémoire habituellement effectué en mémoire RAM. Nous comparons notre approche avec les implémentations existantes et considérons à la fois des disques dur mécaniques et des disques SSD plus modernes. Une validation expérimentale est également proposée.

MPHF Nous introduisons une nouvelle méthode pour effectuer la recherche de pré-image avec les compromis temps-mémoires cryptanalytiques, basée sur les fonctions de hachage parfaites minimales (MPHF, de l'anglais *Minimal Perfect Hash Functions*). Avec les MPHFs, notre technique propose une approche complètement différente des précédents travaux sur le sujet, et les surpasse en terme de complexité spatiale.

Comparaison Nous passons en revue les améliorations proposées dans la littérature et établissons leur taxonomie, en distinguant les améliorations liées à la mémoire de celles qui réduisent le temps d'attaque. Nous introduisons ensuite un modèle pour les comparer toutes, ainsi que notre nouvelle technique MPHF. Notre comparaison est plus complète et plus précise que les travaux existants, et inclut des combinaisons d'améliorations jamais étudiées.

Résumé

Le Chapitre 1 introduit le contexte des compromis temps-mémoire cryptanalytiques ainsi que le problème étudié dans cette thèse, tels que présentés ci-dessus. Les contributions sont détaillées et un plan du manuscrit est établi.

Dans le Chapitre 2, nous introduisons formellement les compromis temps-mémoire cryptanalytiques ainsi que le vocabulaire relatif au domaine, ultérieurement réutilisés dans tout le manuscrit. Le problème de l'inversion d'une fonction à sens unique sur une partie choisie de son espace de départ est étudié en détail. La construction des tables à partir de chaînes, constituées d'itérations de la fonction à sens unique est présentée : on calcule des chaînes d'éléments de l'espace du problème et on conserve dans les tables le point de départ et le point de fin. La nature de compromis temps-mémoire est illustrée, avec une attention particulière sur la place en pratique des différents paramètres en jeu. La taille de l'espace du problème ainsi que la probabilité de succès constituent la base du problème résolu par un compromis temps-mémoire cryptanalytique et sont des paramètres fixes. La taille des tables, représentée par deux paramètres, constitue le cœur du compromis : à problème égal, plus une table est petite, plus la longueur des chaînes nécessaires à sa construction est longue. Les deux phases du compromis temps mémoire sont introduites : la phase de précalcul et la phase en ligne. Il est noté que la phase en ligne est directement dépendante de la longueur des chaînes ce qui expose le compromis : en réduisant la taille d'une table, on allonge la durée de l'attaque en ligne, et vice versa. Les différents algorithmes proposés dans la littérature pour faire office de compromis temps-mémoire cryptanalytiques sont ensuite exposés un à un. Il s'agit de la proposition originale de Hellman en 1980, qui est nommée de manière éponyme dans le manuscrit, de la variante « DP » utilisant des points distingués comme éléments de fin de chaîne, et de la variante « arc-en-ciel » introduite par Oeschlin en 2003, et dès lors appliquée au cassage de mots de passe. Ces trois variantes restent les principales en usage aujourd'hui. Nous mentionnons aussi une variante moins utilisée, mais intéressante, dite « floue ». Toutes ces variantes sont introduites avec leurs caractéristiques principales, auxquelles nous rappelons leur temps moyen en phase en ligne. Le problème de fusion de chaînes qui impacte négativement la probabilité de succès des tables sous la forme de fausses alarmes est considéré, son impact est également quantifié, le tout de manière unifiée.

Nous continuons l'exposé de l'état de l'art dans le Chapitre 3 qui se concentre sur les améliorations proposées dans la littérature sur les variantes algorithmiques mentionnées dans le chapitre précédent. Les améliorations présentées dans ce chapitre visent à faire évoluer positivement un des paramètres du compromis sans faire évoluer les autres de manière significative. Notons que, comme il s'agit de compromis temps-mémoire, si la mémoire est optimisée, il est toujours possible de contre-balancer les paramètres de manière à ce que la mémoire reste la même, mais que le temps de la phase en ligne soit réduit. Nous commençons par présenter les améliorations concernant la mémoire, c'est à dire les techniques qui optimisent les compromis temps-mémoire cryptanalytiques en « compressant » les tables. Une compression classique n'étant pas compatible avec la phase en ligne, particulièrement si la table est déjà prévue pour occuper toute la mémoire disponible, un travail supplémentaire est nécessaire pour s'assurer que la fonctionalité de la phase en ligne n'est pas altérée. Il s'avère que les points de départ

sont peu compressibles et les améliorations se concentrent sur les points de fin. Deux techniques de compression sans perte, compatibles avec la phase en ligne, sont passées en revue : la technique de l'indexation et l'encodage des différences entre les points de fin. Une mention est faite de la troncation des points de fin qui permet une amélioration, car un gain en mémoire est plus impactant que la perte due à la détermination du point non tronqué. Concernant l'amélioration du temps de phase en ligne, la technique des points de contrôle et celle de l'empreinte de chaîne sont discutées. Ces deux techniques stockent plus d'informations sur la chaîne que seulement le point de départ et le point de fin afin de réduire les fausses alarmes. Une technique d'entrelacement des tables est également passée en revue. Cette technique est exploitée par deux publications et permet de donner le temps moyen de la phase en ligne d'une combinaison de plusieurs compromis temps-mémoire, dont les paramètres diffèrent, ainsi que dans le cas d'un même compromis avec des tables de taille différente. Une mention est faite de travaux de la littérature concernant l'utilisation d'un processeur graphique pour accélérer la phase en ligne. Les différents problèmes induits par ce changement d'architecture sont mis en valeur.

Une fois l'état de l'art présenté, on s'intéresse, dans le Chapitre 4 à l'étude des compromis temps-mémoire cryptanalytiques dans le cadre d'une mémoire externe. La plupart des analyses considéraient jusqu'alors que la table était stockée en mémoire RAM, ce qui induisait une simplification des temps de la phase en ligne. En effet, quand la table est en RAM, les temps d'accès sont négligeables. Cependant, les implémentations existantes traitent depuis un certain temps des problèmes de grande taille, de sorte que les tables ne peuvent plus être stockées seulement en RAM. Nous définissons formellement les deux modèles étudiés : le modèle de machine à accès aléatoire utilisé dans la littérature sur les compromis temps-mémoire et le modèle à mémoire externe qui prend en compte l'utilisation d'un disque dur en plus de la mémoire RAM. Nous passons en revue une analyse théorique de l'algorithme utilisé en pratique, destiné à maximiser le transfert des tables en RAM, par les implémentations afin d'effectuer une comparaison avec notre approche qui consiste à effectuer une transition plus directe de l'algorithme en RAM en mémoire externe. Nous établissons une méthodologie expérimentale pour déterminer les paramètres matériels à utiliser dans les algorithmes. Nous établissons ensuite une comparaison entre les deux approches en utilisant un disque dur mécanique, mais aussi un disque dur récent exploitant les technologies modernes de mémoire non volatile. Nos résultats montrent que l'approche utilisée en pratique n'est optimale que dans un contexte limité avec un disque dur mécanique. Notre approche est plus performante pour travailler sur des problèmes de grandes tailles, avec un disque dur récent. Nous la recommandons donc pour de grands problèmes sur des architectures récentes. Nous proposons en conclusion une validation expérimentale de notre modélisation en mémoire externe, avec un problème pratique de taille modérée.

Nous proposons dans le Chapitre 5 une nouvelle approche pour effectuer la phase en ligne, basée sur des fonctions de hachage minimales parfaites. Ce sont des fonctions qui permettent d'associer de manière bijective un ensemble choisi d'éléments dans un espace de nombres de la taille de cet ensemble choisi et dont la description est optimale en terme de complexité spatiale. Nous nous concentrons sur les tables arc-en-ciel pour cette amélioration. Notre technique permet de se passer des points de fin dans la table, ce qui induit un gain substantiel en terme de stockage de tables. Cependant, un nouveau type de fausses alarmes est introduit, qui augmente le nombre de faux positifs et donc ralentit la phase en ligne. Nous introduisons donc un système de signatures, conceptuellement analogues à des points de fin tronqués mais avec un rôle différent (car il n'y a plus de points de fin), afin de minimiser ce problème. Notre technique, en incluant l'espace requis pour stocker les signatures s'avère plus performant que chacune des améliorations de stockage existantes. Notre méthode dépend de l'état de l'art dans le domaine des fonctions de hachage minimales parfaites, dont une borne minimale de

stockage est connue. Des avancements dans ce domaine de recherche adjacent peuvent améliorer la complexité spatiale des tables obtenues. Nous proposons également une implémentation de la technique afin de confirmer la modélisation faite dans ce chapitre. L'aspect novateur de notre proposition remet en cause les suppositions émises jusqu'alors, sur le rôle indispensable des points de fin. L'absence de cette supposition pourrait être une base d'étude dans la recherche de nouvelles améliorations de la phase en ligne.

Les améliorations proposées pour les compromis temps-mémoire ont tendance à être analysées séparément et il est difficile de les comparer intuitivement. Par ailleurs, certaines de ces améliorations sont cumulables, et à ce jour, aucune analyse n'a été faite des combinaisons possibles de plusieurs améliorations. Le Chapitre 6 propose une telle analyse, en prenant en considération les améliorations évoquées dans ce manuscrit. Nous proposons d'effectuer cette analyse dans un contexte compatible avec le cassage de mots de passe qui est plus particulièrement considéré dans cette thèse. Cette étude de contexte montre qu'il est plus judicieux d'utiliser des tables avec une probabilité de succès élevée. Nous avons présenté les variantes principales des compromis temps-mémoire précédemment. Plusieurs comparaisons de ces variantes existent dans la littérature. Nous les passons en revue en exposant leurs limites. D'une part les améliorations ne sont souvent pas considérées, et de l'autre certaines approximations sont trop fortes pour la précision nécessaire lorsqu'elles le sont. Cependant, dans le cadre d'une haute probabilité de succès, nous pouvons statuer sur le fait que les compromis arc-en-ciel sont supérieurs aux autres variantes et nous nous concentrons sur cette variante pour l'analyse avec améliorations. Nous introduisons alors un modèle à mémoire fixée, qui nous permet de comparer de manière égale les différentes améliorations du Chapitre 3, possiblement combinées, ainsi que notre technique introduite au Chapitre 5, avec une évaluation précise du stockage requis pour chacune d'elles. Nous déterminons les paramètres optimaux des améliorations, ainsi que la taille des tables, qui satisfont la condition de mémoire fixée du modèle et évaluons le temps moyen attendu de la phase en ligne. On observe que plusieurs des améliorations de mémoire peuvent être combinées ; par exemple la troncation des points de fin est compatible avec la compression. La technique des points de contrôle est appliquée en sus de cette combinaison d'améliorations de mémoire pour obtenir une configuration optimale, que nous évaluons en parallèle de notre technique basée sur les fonctions de hachage minimales parfaites. Les résultats sont très serrés, notre méthode et la combinaison des améliorations ont des performances très similaires. La conclusion dépend en fait de la qualité de la fonction de hachage minimale parfaite. Notre technique, avec les paramètres utilisés dans les Chapitres 5 et 6, n'est pas plus performante que la combinaison des améliorations. Une amélioration de la fonction de hachage minimale parfaite changerait donc cette conclusion.

En conclusion, dans le Chapitre 7, nous exposons le cadre de notre travail et proposons ce qu'il nous semble être les axes les plus pertinents de poursuite du travail de cette thèse. Nos résultats se concentrent sur la phase en ligne, visant à réduire son temps d'exécution sans impacter négativement la mémoire requise par les tables. L'autre phase des compromis temps-mémoire – le précalcul des tables – est marginalement étudié dans cette thèse et dans la littérature en général. En pratique, le précalcul reste un gros frein à la construction de tables du fait du lourd investissement matériel qu'il requiert. Bien que l'on considère que ce calcul est effectué pendant une période arbitrairement longue par l'attaquant, il ne peut pas être négligé en pratique, et une optimisation du processus de précalcul peut amener à une économie de temps de l'ordre de plusieurs mois pour de gros problèmes. Pourtant, tous les travaux de la littérature utilisent des algorithmes naïfs pour effectuer le précalcul de manière distribuée. La phase de précalcul implique souvent une dé-duplication de chaînes qui aboutissent à un même point de fin. Il est clair qu'en effectuant des contrôles intermédiaires, toute une partie du calcul pourrait être

évitée. Cependant, ce travail n'est pas trivial dans un contexte distribué, où les temps de latence et la bande passante entre les machines doivent être étudiés attentivement, faute de quoi ces paramètres pourraient rapidement devenir un facteur limitant du précalcul, ce qui n'apporterait aucun gain par rapport à l'approche naïve. Par ailleurs, l'utilisation de processeurs graphiques pour accélérer la phase de précalcul mériterait une étude plus approfondie pour atteindre des tailles de problème encore plus larges.

Publication Une partie des travaux réalisés dans cette thèse, en particulier le traitement des compromis temps-mémoire dans le modèle de mémoire externe a été publiée dans l'article suivant :

- Gildas Avoine, Xavier Carpent, Barbara Kordy et Florent Tardif. *How to Handle Rainbow Tables with External Memory.* In : *ACISP 17, Part I.* Sous la dir. de Josef Pieprzyk et Suriadi Suriadi. T. 10342. LNCS. Auckland, New Zealand : springer, juil. 2017, p. 306-323

La présentation de la technique basée sur les MPHFs, ainsi que la comparaison des améliorations des TMTOs est le sujet d'un autre article à paraître :

- Gildas Avoine, Xavier Carpent, Barbara Fila et Florent Tardif. *Cryptanalytic Time-Memory Trade-Offs: Recommended Configurations.* unpublished. 2019

Contents

Remerciements iii

Résumé en français v
Motivation . . . v
Cas d'étude : le cassage de mots de passe . . . vii
Problème étudié . . . ix
Résumé . . . x

---

1 Introduction 3
1.1 Context . . . 4
1.2 Password cracking . . . 6
1.3 Problem statement . . . 7
1.4 Plan . . . 8
1.5 Publications . . . 9

2 Background and preliminaries 13
2.1 Preliminaries . . . 14
2.2 Hellman TMTO . . . 24
2.3 Distinguished points . . . 30
2.4 Rainbow trade-off . . . 35
2.5 Fuzzy rainbow trade-off . . . 40
2.6 Time-Memory-Data trade-offs . . . 42
2.7 Applications . . . 44
2.8 Summary . . . 46

3 Improvements on TMTOs 51
3.1 Improvements on TMTOs . . . 52
3.2 Storage improvements . . . 53
3.3 Checkpoints . . . 63
3.4 Fingerprints . . . 73
3.5 Dealing with non-uniform problem spaces . . . 77
3.6 Heterogeneous tables . . . 81
3.7 Rainbow tables on GPU . . . 84

3.8 Conclusion 85

4 Rainbow tables with external memory **91**
4.1 Motivation 92
4.2 Models 94
4.3 Related work on TMTOs in external memory 97
4.4 Performance of the algorithms 100
4.5 Establishing the constants 106
4.6 Analysis 109
4.7 Experimental results 117
4.8 Conclusion 121

5 MPHF-based rainbow tables **125**
5.1 Introduction 126
5.2 Related work on MPHFs 128
5.3 MPHF-based perfect rainbow trade-off 134
5.4 Analysis 137
5.5 Optimal parameters 141
5.6 Experimental validation 143
5.7 Conclusion 146

6 TMTOs' optimisation **151**
6.1 Introduction 152
6.2 Preliminary comparison 154
6.3 Fixed memory model 159
6.4 Comparison 170
6.5 Conclusion 178

7 Conclusion **183**
7.1 Summary 184
7.2 Results 185
7.3 Open problems 186

Bibliography **189**

The whole notion of passwords is based on an oxymoron. The idea is to have a random string that is easy to remember. Unfortunately, if it's easy to remember, it's something nonrandom like "Susan". And if it's random, like "`r7U2*Qnp,`" then it's not easy to remember.

Bruce Schneier

1 Introduction

Cryptanalytic time-memory trade-offs have been around since the 80s, with steady evolutions in the field. We begin by giving some context in which they have been used, how relevant they are nowadays, and for which applications. Some of these uses involve passwords, and we briefly discuss password cracking, an area of interest of this thesis, though not its main focus.

Contents

1.1 Context 4
1.2 Password cracking 6
1.3 Problem statement 7
1.4 Plan 8
1.5 Publications 9

1.1 Context

Over the last decade, we have witnessed an impressive surge in our reliance on computers to help us handle an ever-increasing amount of private data: photos, communications, banking information, medical records, etc. To secure this data, we need the virtual equivalent of physical keys, to prevent unauthorised third parties from accessing it. More generally, virtual keys are needed all over the Internet to secure common activities such as online banking, shopping or private messaging.

Cryptography, the science of secrecy, provides a means, in the form of algorithms, called ciphers, to ensure that nobody without the right key is able to recover any protected – or more precisely *encrypted* – data. It consists of, on the one hand cryptology, which designs such algorithms, and on the other hand cryptanalysis, which aims at breaking the cipher, *i.e.*, accessing – *decrypting* – the data without the knowledge of the key. While, historically, cryptography has mainly been restricted to military use, it is now employed in various civil applications, as indicated by the prevalence of the aforementioned virtual keys. A reminder of this fact is the occasional mention "military-grade encryption" in some applications that aim at proving some kind of secrecy as a feature. Modern cryptography applies the so-called Kerckhoffs's principle, which states that a cryptosystem should remain secure if everything about its design is known, except for the key used to encrypt the data, hence the usual focus on the keys during the cryptanalysis.

Cryptographic algorithms come in two kinds: those whose encryption key is also their decryption key, which belong to *symmetric* cryptography, and the public-key cryptosystems, which have distinct keys for the two operations, and are referred to as *asymmetric cryptography.* The latter are often used in situations that are more complex than the mere protection of data secrecy, *e.g.*, they are involved in key-exchange protocols or signature schemes, and are not discussed here.

In the context of cryptography, an unencrypted data is called a *plaintext*, and its encrypted counterpart is a *ciphertext.* During the cryptanalysis, a common scenario is the known plaintext attack, where the cryptanalyst has access to a ciphertext whose plaintext is known. Then, in order to perform the cryptanalysis of a cipher, one idea is to exhaust the key space, in a so-called *brute-force* attack, by trying to decrypt the ciphertext with every possible key, until decryption that matches the known plaintext is found. Such an attack does not require any significant memory, as the key sequence can be generated on the fly. Note that, due to the symmetric nature of the cipher, doing things the other way around is equally efficient: one can try to encrypt the plaintext with every possible keys. Ideally, in order to achieve maximum security, there should be no attack against the cipher that is more efficient that the brute force attack. If other attacks are found, the security of the cipher is reduced, down to a point where it is considered broken if there is a discrepancy that is too large between the security claim and the actual security. Therefore, and very logically, cryptographers tend to design ciphers in such a way that they are resistant against known attacks.

One of the tools used in cryptosystem attacks are cryptanalytic time-memory trade-offs (TMTOs). TMTOs are a set of generic probabilistic techniques that allow an attacker to perform an exhaustive search of a given space in less time than what would be required by a brute force attack. In order to speed-up the exhaustive search, they rely on a costly one-time pre-computation whose result is stored in memory, and whose cost in time is often high. In particular, for a single key recovery, the cost of a TMTO is sometimes even higher than that of the one of a brute force attack of a single key. Due to this cost, the key concept of a TMTO is to perform repetitive tasks, either by trying several keys, or relying on several known ciphertext–plaintext pairs. Indeed, if a single key recovery is attempted, using a single plaintext-ciphertext, a brute force attack will cost less than a TMTO pre-computation. TMTOs'

sweet spot lies in the area where a brute force attack is feasible, although expensive, and the memory required to hold the key space size is beyond the reach of the hardware capabilities. Nevertheless, if the pre-computation is not a concern, either because a very long period of time can be allocated to perform it, or because it has been made by someone else, the TMTOs are an interesting alternative to brute force, as long as one possesses enough memory to store the pre-computation results.

The TMTO technique has been used in various fields in cryptography. Their usage in the cryptanalysis of stream ciphers, which are ciphers designed to be used with bit streams of data, has been particularly fruitful. Cryptographers actually take TMTOs into account when designing stream ciphers. For instance, the A5/1 stream cipher is part of the Global System for Mobile Communications (GSM) standard, in which it was used to ensure on-the-air privacy of cellular phones communication. The TMTO technique, combined with some weakness of the cipher, allowed to perform a real-time cryptanalysis of the cipher [BSW00; LLH15], and therefore a real-time eavesdropping of communication, on a commodity computer. The LILI-128 stream cipher, submitted to NESSIE[1], was proven to be unsecure by using TMTOs [Saa02]. In a similar vein, the Toyocrypt stream cipher, which was intended to be used within the Japanese government, along with more general class of Maiorana-McFarland functions, which were designed to be resistant to powerful attacks against many stream ciphers – algebraic attacks – underwent a successful cryptanalysis by TMTOs [KGL06; KL07]. Finally, three eSTREAM [Kle13] candidates were weakened by TMTOs-related attacks: Grain [HJMM08], Lex [DK08a], and MICKEY [HK05]. These attacks led to significant modifications of the design of the ciphers.

Another field in which TMTOs were applied is lightweight cryptography, as found on embedded devices with low computational capacity, such as Radio-Frequency Identification (RFID) devices. Digital Signature Transponders (DSTs) are RFID devices securing electronic payment such as the Exxon-Mobil SpeedPass®, or vehicle immobilisers. With TMTOs, it was possible to make the attack fast enough that it is practical in real-life scenarios [BGS+05]. Two other vehicle immobilizers, the Hitag2 [VGB12], and the Megamos Crypto [VGE15], benefited from TMTOs to speedup the key recovery, in the same way.

One of the most well-known applications of TMTOs is to facilitate preimage recovery from cryptographic hash function digests, especially in the context of password cracking. They were employed to crack Windows LAN Manager (LM) passwords [Oec03], and Unix passwords [MBPV05]. Password cracking as an application of TMTOs is a particular focus in this thesis, and is further discussed in Section 1.2. When considering the use case of hash inversion, a typical setup which benefits from TMTOs is the daily work of a penetration tester, who wants to quickly test a set of presumably weak passwords on a given client database. Similarly, in the forensic field, it can sometimes be required to invert hashed data to recover information from a dump. Finally, custom security systems may employ cryptographic hash functions as a security measure to obfuscate some secrets. Reverse engineering of such systems can be helped by a tool allowing quick hash inversions.

Finally, TMTOs were used to break cryptocurrencies' Proofs of Space [DFKP15]. In a distributed linked chain system such as Bitcoin, the consensus is secured by a proof of work preventing anyone from rewriting the chain, requiring some work which grows exponentially with the length of the chain. Proofs of space aim at taking the place of proofs of work by replacing the necessity of work by a commitment to store arbitrary data. TMTOs allow to reduce the storage required for such proofs, defeating their purpose. This fact led to new proofs of space constructions involving TMTOs themselves [AAC+17].

We are interested in the study of the TMTO technique. The work in this thesis concern the algorithms themselves and is application-agnostic. However, knowing the problem space helps us to choose practical

[1] *New European Schemes for Signatures, Integrity and Encryption* was a European project aimed at establishing secure cryptographic primitives.

parameters. That is why we detail hereafter one common application for TMTOs, which is password cracking.

1.2 Password cracking

Let us now focus on password authentication, and how TMTOs have been, and still are, used to strengthen it. Many security systems, particularly online services, use passwords under the hood. Authentication credentials in general are a target for attackers, since they allow an illicit access to a service, which is often indistinguishable from a legitimate one. There exist methods, such as multi-factor authentication, where another form of authentication is required in addition to the password, which prevents most attacks involving credential stealing. However these are still nowadays not used as often as they should be. Other advanced methods such as facial recognition, or authentication by fingerprint, are sometime used, *e.g.*, by smartphones, as authentication mechanisms. However, it is often possible to use a password or a PIN as a fallback. The security of a system being that of its weakest link, these advanced methods do not really help in the case of a successful password attack.

Users tend to reuse passwords across different services. An effective strategy for attackers is to get access to the password database of a lowly secured service. Then, once an identity is associated with a password, one can try the same password or a variant of it on other, more secure, services. Password databases are periodically recovered from weakly secured systems by malicious third parties. They are also from time to time leaked on the Internet where anyone can download them. Such leaks triggered the emergence of services such as Have I Been Powned [Hun13], which keep track of every public breach.

Passwords are not usually stored in clear text in databases. Otherwise, an access to the hard-drive would be sufficient to learn the passwords. In particular, the entity hosting and operating the database can easily access them. Unfortunately, password are still sometimes stored this way, as indicated by the services that are able to send your lost password by e-mail upon request. The idea to improve the situation is to allow the storage of the password without the service having knowledge of it. A classical technique is to store a digest obtained by applying a cryptographic hash function of the password alongside the identifier of the user. *Password cracking* consists in the action of recovering a password from this digest.

Public-facing services, in particular on the Internet, are under permanent threat by malicious third parties. There are indeed attackers who relentlessly stress security systems, in pursuit of any flaw to be exploited in order to gain privileged access to them. Their methods are, for obvious reasons, often undocumented. This makes the task of securing a system a perpetual cat-and-mouse game. One of the objectives of security researchers is to keep up with these attackers. Disclosing attacks' methodology and performance, and documenting them, helps raise awareness on the frailty of obsolete systems. It is then possible to design practical counter-measures to help reinforce these systems. In the industry, it is often costly to upgrade systems. Moreover, the security is not always among the priorities when considering monetary constraints. For these reasons, academic publications discussing theoretical attacks are not impacting the industry enough. To alarm the community, one has to prove that an attack is practical, even if expensive, There is then a last stage, when techniques or hardware capabilities have improved so much, that it is possible to perform the attack with an off-the-shelf setup.

That is why, even though there exist password techniques such as salting (see Section 2.7.1), which can make the TMTO technique arbitrarily costly to invert digests, TMTOs can still be used to assess the strength of a password. As shown by the occasional leaks, there are still services that do not

salt passwords, and there is no possibility of knowing if it is the case in general when creating an account. Therefore, using a password which is beyond the cracking capabilities of a TMTO is among the prerequisites of a strong security.

In general, the costly pre-computation needed to perform a TMTO narrows their usage to some specific settings. In particular, three attack situations encompass most of the scenarios considered when securing a system.

Numerous attacks If one is given over time several batches of digests to invert, one can consider performing as many brute force attacks as there are batches. If the number of required attacks is significant enough to outweigh the pre-computation cost, then a TMTO can become advantageous because it requires only one pre-computation. The cost of processing a digest in search of a preimage is negligible compared to a full brute force attack. Therefore, if one is confident in the need of searching for many digest preimage, the cost of the pre-computation can quickly be compensated for.

Lunchtime attack If we consider that the attack can be performed using brute force, but would require too much time, TMTOs can be used to arbitrarily reduce the attack time, with the only requirement of providing pre-computed tables during the attack. This permits schemes such as the lunchtime attack which refers to a situation, also known as the evil maid attack, where an attack has to be performed in a short amount of time at a given moment, but is planned beforehand. A common illustration of this attack, as the name suggests, is a context where the attacker waits for an opportunity to get access to the victims' computer, possibly for a short period of time, for example when the victim leaves her office, without her computer, during lunch.

Downloadable tables The pre-computation and the online attack can be performed by different entities, possessing different computational resources. This is the case when the results of a pre-computation are shared online. The availability of already computed tables allows individuals to perform attacks without the need for huge computing power. There are initiatives to provide such tables for common hash functions. An example is the RainbowCrack [Shu09] project which makes use of crowd-computing by distributing the pre-computation to participants, similarly to what is done with projects such as Folding@Home [Pan00] or SETI [ACK+02]. Other examples include the Ophcrack tool [TO05], which comes with several tables, and can be used quickly once downloaded, whereas performing the pre-computation of some of the tables would require a significant cost in terms of time and computational resources.

1.3 Problem statement

With the TMTO method, it is possible to arbitrarily reduce the time needed to perform a brute-force attack. However, this comes at the expense of needing a large memory to store the result of the pre-computation. The work in this thesis aims at improving the trade-off algorithm, in the form of an optimisation after the pre-computations, which is the subject of Chapter 5, so that this memory requirement is reduced. The research question studied in this manuscript is the following: *How can we improve the space efficiency of the TMTO method?* Such an efficiency improvement allows to tackle larger problems with the TMTOs technique, *i.e.*, ciphers with larger key spaces, or inverting hash function on a larger input space, than what is possible today.

We will keep the particular application of password cracking in mind when studying the practicality of the problems considered, in order to provide a better sense of how practical the technique is with regard to the memory requirement. We are interested in the range of the capabilities of TMTOs as far as practical attacks are concerned, by studying them in a relevant, realistic context. The password cracking application will help us do that, but the results can easily be generalised to other suitable problems, such as the ones described in the previous section.

The results of our analysis on the TMTOs lead to the three main following contributions.

TMTOs on disk When the problem space size is very large, the memory required to perform an inversion, even using cryptanalytic time-memory trade-offs, outweighs the available RAM. Therefore, the use of external memory, such as hard-drives, is considered. We study the impact of using disk, instead of RAM, as memory for the trade-off. This problem has hardly ever been discussed in the literature before our work. We propose a different approach to performing TMTOs on disk than what is usually done in practice, in the implementation landscape. We compare our approach to the implemented algorithms, when considering both classical mechanical hard drives and newer types of disks, such as SSDs. An experimental validation is also proposed to corroborate our findings.

MPHF trade-off We introduce a novel method to perform the pre-image search with TMTOs, based on minimal perfect hash functions (MPHFs), so-called MPHF trade-offs. MPHFs are a data structure mainly used to construct hash tables, and reusing them in order to perform the trade-off attack allows us to discard information previously needed in the description of the algorithm. Our technique takes a different approach to what was done in most existing implementations of TMTOs, and outperforms them in terms of memory footprint.

TMTO comparison We review all of the improvements applied on the TMTOs presented in the literature and propose their taxonomy, distinguishing between memory-related improvements and those that reduce the online time. We then introduce a model for comparing all of these improvements, as well as our newly introduced MPHF trade-off, in a more precise manner than what was previously proposed in the literature. We also give the first comparison of some combinations of TMTO improvements, which were only analysed in isolation so far.

1.4 Plan

The outline of this thesis is as follows. The first two chapters build the theoretical ground which we rely on in the rest of the document. In Chapter 2, we formally present the problem tackled by TMTOs. The terminology is introduced, and the key points of interest of the trade-offs are presented. There are several variants of cryptanalytic trade-offs, which are introduced along with their main characteristics. We conclude by giving some problem examples, which are among the most frequently used in practice to perform TMTO attacks.

Chapter 3 is a survey of improvements on TMTOs. We give a thorough review of the techniques from the literature that provide a significant enough gain on the TMTO technique, that they are worth to be considered. Each method is presented in a unified way so that the different methods can be easily combined later on, and the gain obtained can be quantified.

We present, in Chapter 4, the transition of TMTOs from the classical RAM model, where the memory is fast and the time of numerous operations can be neglected, to the external memory model, which

accounts for slower memories such as disks. The two models are formally introduced, then the algorithm that is used in practice is presented. We provide an analysis for both this algorithm and our approach, which consists in a more straightforward tailoring of the algorithm used with RAM. An evaluation of the two algorithms, using practical parameters, is performed. In our comparison, we consider mechanical disks as well as SSDs.

In Chapter 5, we introduce a novel technique to perform TMTOs, the MPHF-based trade-off. This technique is based on a particular type of hash functions which present an interesting memory property, the MPHFs. We formally define MPHFs. We then clarify their inner workings, by reviewing the related work. Then, we provide an analysis of the method, and show how it performs memory-wise. Finally, we validate our analysis experimentally.

The goal of Chapter 6 is to provide a comprehensive comparison of TMTOs, with a secondary objective of finding which of the TMTOs variants, with which improvements, provides the best performance, in a practical context. In order to determine the best trade-off, we review the previous comparisons of TMTO variants from the literature. We then introduce our comparison model based on a fixed amount of memory, similarly to what is done in practice when implementing TMTOs. This model allows us to extend the previous analysis' works on some of the improvements, and to provide some comparison of improvement combinations that have not been considered until now.

We conclude in Chapter 7, where we state what we consider to be the main open problems in the TMTO field. We then describe the future work that would be interesting to pursue, following what was done in this thesis.

1.5 Publications

The main part of the results of Chapter 4, concerning TMTOs in external memory, can be found in the following publication:

- Gildas Avoine, Xavier Carpent, Barbara Kordy, and Florent Tardif. *How to Handle Rainbow Tables with External Memory*. In: *ACISP 17, Part I*. ed. by Josef Pieprzyk and Suriadi Suriadi. Vol. 10342. LNCS. Auckland, New Zealand: springer, July 2017, pp. 306–323

The introduction of the MPHF trade-off described in Chapter 5, as well as the comparison in the fixed memory model of the MPHF technique with various improvements proposed in the literature, performed in Chapter 6, constitute the main part of the following article, to be submitted:

- Gildas Avoine, Xavier Carpent, Barbara Fila, and Florent Tardif. *Cryptanalytic Time-Memory Trade-Offs: Recommended Configurations*. unpublished. 2019

References

[AAC+17] Hamza Abusalah, Joël Alwen, Bram Cohen, Danylo Khilko, Krzysztof Pietrzak, and Leonid Reyzin. *Beyond Hellman's Time-Memory Trade-Offs with Applications to Proofs of Space*. In: *Advances in Cryptology – ASIACRYPT 2017*. Ed. by Tsuyoshi Takagi and Thomas Peyrin. Vol. 10625. LCNS. Cham: Springer International Publishing, 2017, pp. 357–379. ISBN: 978-3-319-70697-9 (cit. on p. 5).

[ACFT19] Gildas Avoine, Xavier Carpent, Barbara Fila, and Florent Tardif. *Cryptanalytic Time-Memory Trade-Offs: Recommended Configurations.* unpublished. 2019 (cit. on pp. xiii, 9, 126).

[ACK+02] David P Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. "SETI@ home: an experiment in public-resource computing". In: *Communications of the ACM* 45.11 (2002), pp. 56–61 (cit. on p. 7).

[ACKT17] Gildas Avoine, Xavier Carpent, Barbara Kordy, and Florent Tardif. *How to Handle Rainbow Tables with External Memory.* In: *ACISP 17, Part I.* Ed. by Josef Pieprzyk and Suriadi Suriadi. Vol. 10342. LNCS. Auckland, New Zealand: springer, July 2017, pp. 306–323 (cit. on pp. xiii, 9).

[BGS+05] Steve Bono, Matthew Green, Adam Stubblefield, Ari Juels, Avi Rubin, and Michael Szydlo. *Security Analysis of a Cryptographically-Enabled RFID Device.* In: *In 14th USENIX Security Symposium.* USENIX, 2005, pp. 1–16 (cit. on p. 5).

[BSW00] Alex Biryukov, Adi Shamir, and David Wagner. *Real Time Cryptanalysis of A5/1 on a PC.* In: *Fast Software Encryption – FSE'00.* Ed. by Bruce Schneier. Vol. 1978. Lecture Notes in Computer Science. New York, USA: Springer, Apr. 2000, pp. 1–18 (cit. on pp. 5, 56, 62, 63).

[DFKP15] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. *Proofs of Space.* In: *Advances in Cryptology – CRYPTO 2015.* Ed. by Rosario Gennaro and Matthew Robshaw. Vol. 9216. LCNS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 585–605. ISBN: 978-3-662-48000-7 (cit. on p. 5).

[DK08a] Orr Dunkelman and Nathan Keller. *A New Attack on the LEX Stream Cipher.* In: *Advances in Cryptology - ASIACRYPT 2008.* Ed. by Josef Pieprzyk. Vol. 5350. LCNS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 539–556. ISBN: 978-3-540-89255-7 (cit. on p. 5).

[HJMM08] Martin Hell, Thomas Johansson, Alexander Maximov, and Willi Meier. *The Grain Family of Stream Ciphers.* In: *New Stream Cipher Designs: The eSTREAM Finalists.* Ed. by Matthew Robshaw and Olivier Billet. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 179–190. ISBN: 978-3-540-68351-3 (cit. on p. 5).

[HK05] Jin Hong and Woo-Hwan Kim. *TMD-Tradeoff and State Entropy Loss Considerations of Streamcipher MICKEY.* In: *Progress in Cryptology - INDOCRYPT 2005.* Ed. by Subhamoy Maitra, C. E. Veni Madhavan, and Ramarathnam Venkatesan. Vol. 3797. LCNS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 169–182. ISBN: 978-3-540-32278-8 (cit. on p. 5).

[Hun13] Troy Hunt. *Have I Been Pwned.* https://haveibeenpwned.com. Accessed: 2019-01. 2013. Link. (Cit. on p. 6).

[KGL06] Khoongming Khoo, Guang Gong, and Hian-Kiat Lee. *The Rainbow Attack on Stream Ciphers Based on Maiorana-McFarland Functions.* In: *Applied Cryptography and Network Security.* Ed. by Jianying Zhou, Moti Yung, and Feng Bao. Vol. 3989. LCNS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 194–209. ISBN: 978-3-540-34704-0 (cit. on p. 5).

[KL07] Guang Gong Khoongming Khoo Guanhan Chew and Hian-Kiat Lee. *Time-Memory-Data Trade-off Attack on Stream Ciphers based on Maiorana-McFarland Functions.* Cryptology ePrint Archive, Report 2007/242. `https://eprint.iacr.org/2007/242`. 2007 (cit. on p. 5).

[Kle13] Andreas Klein. *The eStream Project.* In: *Stream Ciphers.* London: Springer London, 2013, pp. 229–239. ISBN: 978-1-4471-5079-4 (cit. on p. 5).

[LLH15] Jiqiang Lu, Zhen Li, and Matt Henricksen. *Time-Memory Trade-Off Attack on the GSM A5/1 Stream Cipher Using Commodity GPGPU - (Extended Abstract).* In: *ACNS 15.* Ed. by Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis. Vol. 9092. LNCS. New York, NY, USA: Springer, Heidelberg, Germany, June 2015, pp. 350–369 (cit. on pp. 5, 42, 84).

[MBPV05] Nele Mentens, Lejla Batina, Bart Preneel, and Ingrid Verbauwhede. *Cracking Unix Passwords using FPGA Platforms.* SHARCS - Special Purpose Hardware for Attacking Cryptographic Systems. Paris, France: Ecrypt, Feb. 2005 (cit. on p. 5).

[Oec03] Philippe Oechslin. *Making a Faster Cryptanalytic Time-Memory Trade-Off.* In: *CRYPTO 2003.* Ed. by Dan Boneh. Vol. 2729. LNCS. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 2003, pp. 617–630 (cit. on pp. 5, 35, 36, 56, 107, 153, 156, 157).

[Pan00] Vijay Pande. *Folding@ home.* Accessed: Jun 2019. 2000. Link. (Cit. on p. 7).

[Saa02] Markku-Juhani Olavi Saarinen. *A Time-Memory Tradeoff Attack Against LILI-128.* In: *Fast Software Encryption.* Ed. by Joan Daemen and Vincent Rijmen. Vol. 2365. LCNS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 231–236. ISBN: 978-3-540-45661-2 (cit. on p. 5).

[Shu09] Zhu Shuanglei. *RainbowCrack.* Accessed: Apr 2017. 2009. Link. (Cit. on pp. 7, 93, 187).

[TO05] Cedric Tissières and Philippe Oechslin. *Ophcrack.* Accessed: Apr 2017. 2005. Link. (Cit. on pp. 7, 35, 56, 93, 99, 162).

[VGB12] Roel Verdult, Flavio Garcia, and Josep Balasch. *Gone in 360 Seconds: Hijacking with Hitag2.* In: *In 21st USENIX Security Symposium.* USENIX, Jan. 2012, pp. 237–252 (cit. on p. 5).

[VGE15] Roel Verdult, Flavio D. Garcia, and Baris Ege. *Dismantling Megamos Crypto: Wirelessly Lock-picking a Vehicle Immobilizer.* In: *Supplement to the Proceedings of 22nd USENIX Security Symposium (Supplement to USENIX Security 15).* Washington, D.C.: USENIX Association, 2015, pp. 703–718. ISBN: 978-1-931971-232 (cit. on p. 5).

Knowing is not enough; we must apply. Willing is not enough;
we must do.

Johann Wolfgang von Goethe

Background and preliminaries 2

A formal introduction of the TMTO technique will be required for our analysis. This is the objective of this chapter, which also introduces the terminology that we will use in our presentation throughout this document. We extend this background to include the popular variants of the technique, along with some of the main results regarding their efficiency.

Contents

2.1 Preliminaries **14**
- 2.1.1 One-way function inversion problem 14
- 2.1.2 Random functions 15
- 2.1.3 Trade-off 17
- 2.1.4 Chains 19
- 2.1.5 TMTO matrix 20
- 2.1.6 Success rate 22
- 2.1.7 Tables 23

2.2 Hellman TMTO **24**
- 2.2.1 Offline phase 24
- 2.2.2 Online phase 27
- 2.2.3 Analysis of the Hellman online phase 28
- 2.2.4 Variants 30

2.3 Distinguished points **30**
- 2.3.1 Pre-computation 31
- 2.3.2 Success rate 32
- 2.3.3 Online phase 33
- 2.3.4 Analysis 34

2.4 Rainbow trade-off **35**
- 2.4.1 Offline phase 35
- 2.4.2 Online phase 37

2.5 Fuzzy rainbow trade-off **40**

2.6 Time-Memory-Data trade-offs **42**
- 2.6.1 BG attack 42
- 2.6.2 TMDTO attack 43

2.7 Applications **44**
- 2.7.1 Password cracking 44
- 2.7.2 Cipher cryptanalysis 45
- 2.7.3 Pseudo-random number generators 46

2.8 Summary **46**

2.1 Preliminaries

This section introduces and defines the core notions of the time-memory trade-offs (TMTOs). These notions will be leveraged in the description of the TMTOs themselves in Sections 2.2 to 2.6. In particular, we present the inversion problem, which TMTOs are designed to tackle, and TMTO tables which are the trade-off data structure in memory.

2.1.1 One-way function inversion problem

A TMTO's goal is to provide a quick solution to the inversion problem, which is to find a pre-image of a function in a given subset. The inversion problem revolves around the idea of one-way functions which are easy to compute, and very difficult to invert, hence the interest of the problem.

Definition 2.1 (One-way function)**.** *Let us consider a function $f : \Omega \to \mathcal{H}$. If the two following conditions are verified:*

1. *the computation of $f(x)$ for any x is at most of polynomial time complexity in the size of x, and*
2. *there is no known algorithm that can compute f^{-1} in a time complexity lower than $O(|\mathcal{H}|)$,*

then f is said to be a one-way *function.*

Most one-way functions encountered in TMTOs follow an even tighter set of constraints. Indeed, the typical one-way function f for which a TMTO make sense, can be computed in $O(1)$ time complexity. Note that usually $\Omega \gg \mathcal{H}$, which precludes the one-way function from being injective.

Let us consider a one-way function $f : \Omega \to \mathcal{H}$ such as the one presented in Figure 2.1. One can think of Ω as a password space or a key space, and the image space $\mathcal{H}$ corresponds, informally, to a digest or ciphertext space. In practice, Ω is so large that it is not practical to even enumerate its elements, therefore we have to consider a smaller input space for the inversion problem. We choose a subset $\mathcal{N} \subset \Omega$ which we call the *problem space*, *i.e.*, $\mathcal{N}$ is the set of elements on which we want to apply our TMTO. Contrarily to Ω, the set $\mathcal{N}$ is always small enough that its elements can be enumerated. This is a requirement on the problem space in order to be able to perform the TMTO. This means that each TMTO is associated with a given problem space.

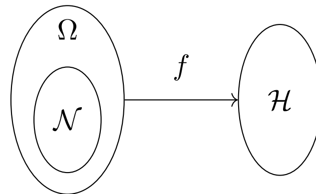


Figure 2.1 – One way function f

Formally, the problem solved by the TMTO, is to find preimages of the one-way function f in our problem space $\mathcal{N}$.

Definition 2.2. *Given an element $y \in \Omega$, the* inversion problem *consists in finding efficiently a $x \in \mathcal{N}$ such that $f(x) = y$, when such an x exists.*

Depending on the context of application of the TMTO, when f is not injective, it may be sufficient to find any of the several preimages $x \in f^{-1}(y)$. However, there are cases were the specific x is required. This is a minor concern, as in practice $|\mathcal{N}| \ll |\mathcal{H}|$, and so $|\mathcal{N}| \ll |\Omega|$, in such a way that a given element in $\mathcal{H}$ will often only have one of its preimages in $\mathcal{N}$.

2.1.2 Random functions

The TMTO is a probabilistic technique which relies on functions which associate elements of $\mathcal{N}$ together in a random manner. It often leverages the randomness of the one-way function itself, as one-way functions used in applications where TMTOs are used often happen to have this characteristic. We now formally define what we mean by a random function. For simplicity[1], we choose to define random functions by the distribution of their output, which is the characteristic we are interested in, and which the subsequent results rely on.

Definition 2.3 (Random function)**.** *A function $f : \mathcal{N} \to \mathcal{H}$ is said to be* random *if its output in $\mathcal{H}$ follows a discrete uniform distribution. If for some $x \in \mathcal{N}$, we consider the random variable X taking the possible values of $f(x)$, we have*

$$\forall y \in \mathcal{H}, \quad \Pr(X = y) = \frac{1}{|\mathcal{H}|}. \tag{2.1}$$

We also consider pseudo random functions which behave like random functions, *i.e.*, for which the equality in Equation (2.1) becomes an approximation: $\Pr(X = y) \approx \frac{1}{|\mathcal{H}|}$. For simplicity, we will omit the term "pseudo" and simply refer to such function as random functions.

In [FN91], Fiat and Naor have shown that it is possible to construct a TMTO that can use any one-way function, not necessarily a random one. Unfortunately, their solution comes at a price – a significantly worse performance than with a random function. We will therefore assume that we are always dealing with a random one-way function, without losing the practicality of the method, since many applications of TMTOs involve such functions (see Section 2.7).

The complexity in the analysis of TMTOs lies mostly in the fact that we are dealing with random functions. A point of interest is the behaviour of random functions when the input space is the same as the output space, and in particular the probability that several input elements map to the same output element. We now discuss the concept of collisions, and their implications on iterated random function image spaces.

Definition 2.4 (Collision)**.** *For a random function $F : \mathcal{N} \to \mathcal{N}$, the situation where*

$$\exists (x, x') \in \mathcal{N}^2, x \neq x', \quad \textit{s.t.} \quad F(x) = F(x').$$

is called a collision *on F.*

Given enough elements of $\mathcal{N}$, we can expect collisions to happen. This is the subject of the well-known birthday paradox, and quantified hereafter.

Birthday paradox If we constitute two subsets $S_1, S_2 \subset \mathcal{N}$, by picking elements at random in $\mathcal{N}$, there is a high probability that $S_1 \cap S_2 \neq \emptyset$, as soon as $|S_1| \times |S_2| > N$. Indeed, if we already picked k elements at random in $\mathcal{N}$, the probability that a newly picked element is not in $\mathcal{N}$ is

$$\frac{N-1}{N}\frac{N-2}{N}\cdots\frac{N-k}{N} = \prod_{i=1}^{k}\left(1 - \frac{i}{N}\right),$$

[1] A more complete definition would describe a stochastic process, which involves dealing with the average of a family of all functions with the same domain and image, but the additional concepts required would clutter the developments unecessarily.

which can be approximated when $k \ll N$ by

$$\prod_{i=1}^{k} e^{-\frac{i}{N}}$$

to finally obtain

$$e^{-\frac{k(k+1)}{2N}} \approx e^{-\frac{k^2}{2N}}.$$

We can see that this probability starts to become small when k^2 exceeds $2N$, that is when $\sqrt{k} > \sqrt{2}N$. In other words, on average, in a set $\mathcal{N}$ of size N, it is probable to get a collision on F by picking about $\sqrt{N}$ elements in $\mathcal{N}$.

Let $F : \mathcal{N} \to \mathcal{N}$ be a function. We will denote with F^k the iterated function

$$\underbrace{F \circ F \circ \cdots \circ F \circ F}_{k \text{ times}}.$$

This collision property implies that for a function $F : \mathcal{N} \to \mathcal{N}$ which is also random, we have, for $D \subseteq \mathcal{N}$:

$$D \supseteq F(D) \supseteq F^2(D) \supseteq F^3(D) \supseteq \dots,$$

because of the collisions. Let us first specify the amount by which the successive application of a random function decreases the size of the iterated image spaces.

Proposition 2.1 (Iterated images size). *Let D_0 be a set of m_0 distinct elements in $\mathcal{N}$, and $F : \mathcal{N} \to \mathcal{N}$, where $|\mathcal{N}| = N$, be a random function. $\forall k > 0$, we note $D_k = F^k(D_0)$, and $m_k = |D_k|$. Then,*

$$m_k \approx \frac{2N}{\frac{2N}{m_0} + k}.$$

Proof. This proof is based on a variant of the result which appeared in [AJO08] with different assumptions. We consider $i \in \mathbb{N}$. The probability of an element of D_i to be in $D_{i+1} = F(D_i)$ is $\frac{m_i}{N}$. Therefore, the proportion of the m_i elements of D_i that are not in D_{i+1} is

$$\Pr(x \notin D_{i+1} \mid x \in D_i) = \left(1 - \frac{m_i}{N}\right)^{m_i}, \tag{2.2}$$

which can be approximated for a large m_i by $e^{-\frac{m_i}{N}}$. The quantity $1 - \Pr(x \notin D_{i+1} \mid x \in D_i)$, which corresponds to the proportion $\frac{m_{i+1}}{N}$ of distinct elements in D_{i+1}, can be approximated using the first terms of the exponential power series $e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$. This gives, for $\frac{m_{i+1}}{N}$:

$$\frac{m_{i+1}}{N} = \frac{m_i}{N} + \frac{1}{2}\frac{m_i^2}{N^2} + \cdots \approx \frac{m_i}{N} + \frac{1}{2}\frac{m_i^2}{N^2},$$

and so we have a recurrence relation[2]

$$\forall i \in \mathbb{N}, \quad m_{i+1} \approx N\left(\frac{m_i}{N} + \frac{m_i^2}{2N^2}\right),$$

[2] The fact that the relation between m_{i+1} and m_i holds asymptotically requires that the distribution of the image sizes indeed stays close to the average, which is a common assumption in cryptography that has been discussed in Appendix B of [HM13], and which applies to the functions we are interested in here.

which can be viewed as a differential equation by taking $\frac{dm_i}{di} = m_{i+1} - m_i$:

$$\frac{dm_i}{di} \approx -\frac{m_i^2}{2N}.$$

Then, we have

$$\int -\frac{dm_i}{m_i^2} \approx \int \frac{di}{2N}$$

which gives

$$\exists C \in \mathbb{R}, \quad \frac{1}{m_i} \approx \frac{1}{2N}(i + C),$$

so the solution is

$$m_i \approx \frac{2N}{i + C}. \tag{2.3}$$

We have $m_0 = \frac{2N}{C}$, and so $C = \frac{2N}{m_0}$, which we inject in the expression (2.3) to get the stated value. □

2.1.3 Trade-off

As their name suggests, TMTOs are *trade-offs*, meaning that they allow to manage resources such as the execution time and the memory requirement, by providing a choice to the user about which weight to put on either of theses resources. This also makes their study harder, compared to the more classical algorithms which possess a fixed time and memory complexity. We present hereafter the dimensions of the trade-off.

TMTOs put in relation various dimensions, and can informally be seen as a set of constraints on theses dimensions: the modification of one of them influences the others. There are three interesting dimensions to look at when dealing with the inversion problem. These are summarised in the following paragraphs. We consider a one-way function $f : \mathcal{N} \to \mathcal{H}$.

Problem size The cardinal of $\mathcal{N}$, which was denoted with N in this section, is called the problem size. In other words, it is the number of the preimages that are recoverable by the TMTO. An interesting question is how large N can be in practice, as it determines the relevance of the TMTO technique. The larger the problem space can be, the more applications TMTOs can have.

Memory The memory of the TMTO corresponds to the amount of information M that is required to perform the inversion. In the case of TMTO, it takes the form of a data structure, often called *table*, which is a part of the inversion algorithm.

Time The attack time T is the amount of time used to perform an inversion in average. The time T can be measured in either the wall clock time, or the number of applications of the one-way function f, especially when f computes with an $O(1)$ time complexity.

Using the above notation, a TMTO is a technique providing an algorithm $\mathcal{A}_{\mathcal{N}}^{f}$, described in M bits of memory, which performs the inversion. Algorithm $\mathcal{A}_{\mathcal{N}}^{f}$ attempts to find a preimage in $\mathcal{N}$ of an element in $\mathcal{H}$ by the function f in a time T.

We now describe two simple algorithms that can be considered to tackle the inversion problem. We consider some $y_0 \in \mathcal{H}$, and we want to find the x_0 such that $y_0 = f(x_0)$, or at least one of them, if f is not injective. We assume that such a x_0 exists.

Exhaustive search The first idea consists in computing $y = f(x)$, for each x of the problem space until we have $f(x) = y_0$. This does not require any memory M except for the one y_0 that will be compared, but one has to go in average through $T = \frac{N}{2}$ (up to $T = N$ in the worst case) applications of the f function to find a preimage of y_0.

Lookup search Another idea is to assume that we possess an associative data structure which maps $f(x)$ to x efficiently, for each x of the problem space. This data structure is then stored in $O(N)$ in memory.

The three parameters N, M, and T are linked in such a way that they influence each other. In particular, it becomes clear that T and M are inversely correlated to each other. If we keep a constant problem size N and we increase one of the parameters, the algorithm $\mathcal{A}_{\mathcal{N}}^f$ will need lower values for the other parameters to perform the inversion. The trade-off here corresponds to the fact that we can choose to compute either a faster and heavier algorithm, or a lightweight but slower one.

The two algorithms mentioned above are on opposite sides of the trade-off spectrum. The exhaustive search and the lookup search correspond to the configurations $(T = N, M = 0)$ and $(T \approx 0, M = N)$ respectively. The TMTO lies between these extremes, allowing us to perform the inversion in $(T < N, M < N)$, as shown in Figure 2.2.

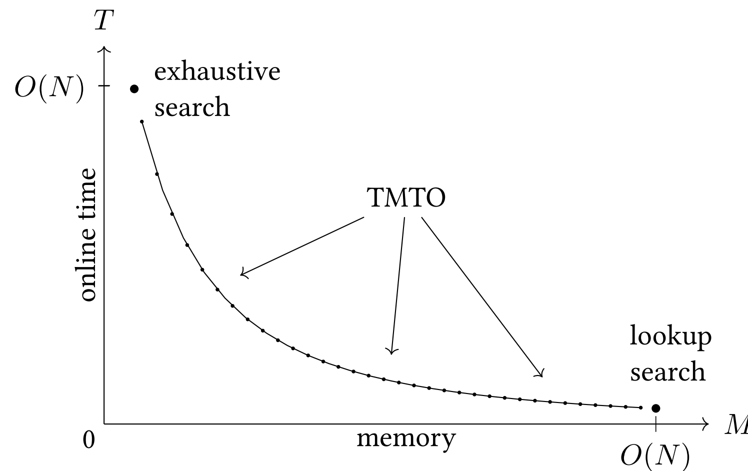


Figure 2.2 – TMTO spectrum

An important aspect of TMTOs is that they involve two phases: the construction of the algorithm $\mathcal{A}_{\mathcal{N}}^f$ on the one hand, and the execution of the algorithm for an inversion on the other hand. These two steps are called *online* and *offline* phases, to underline the fact that the construction of the algorithm is a preparation step whereas the application of the algorithm is how the TMTO is used.

Offline phase The offline phase consists mainly in doing the pre-computation of the TMTO tables, that are part of the algorithm $\mathcal{A}_{\mathcal{N}}^f$ performing the inversion. Note that in most of the cases, this step takes more time than an exhaustive search. It is stated in [HKR83] that no general TMTO can be more efficient than an exhaustive search, for a single inversion, if the pre-computation is taken into account. The rationale of considering this phase despite its huge pre-computation cost is that it is only done once and for all, and therefore can take a lot of time, while allowing quick results when an inversion is needed at a given moment.

Online phase Given an element in $\mathcal{H}$ whose pre-image by f exists in $\mathcal{N}$, the online phase consists in recovering this pre-image via the algorithm $\mathcal{A}_{\mathcal{N}}^{f}$, in expected time T. The idea is that the algorithm is very quick and uses a relatively modest amount of memory to perform the inversion.

It often happens that at least one dimension is very small compared to N, possibly both of them in a balanced configuration, hence requiring few resources for the online phase. Nevertheless, to achieve such decrease in time and memory in the online phase, the offline phase still requires a memory of order N, which is then shrunk down to M.

2.1.4 Chains

We now introduce TMTO chains that are a core concept of TMTOs. In particular, a vast majority of the time spent during a TMTO, in both phases, is used to compute chains.

Definition 2.5. *A TMTO* chain *is a sequence of elements $X_1, \ldots, X_t$ in $\mathcal{N}$:*

$$X_1 \longrightarrow X_2 \longrightarrow \cdots \longrightarrow X_t,$$

such for each $n < t$, the link X_{n+1} of the chain is derived from the link X_n via the one-way function f.

The first and the last link of a chain, hereafter called *starting point* (SP) and *endpoint* (EP) respectively, are of particular importance in TMTOs.

However, the input space $\mathcal{N}$ and output space $\mathcal{H}$ of f are often sets of different objects, *e.g.*password strings which are hashed by f to obtain bitstrings or integers. Even if we consider only integers, *i.e.*, $\mathcal{N} \times \mathcal{H} \times \Omega \subset \mathbb{N}^3$, their cardinal is not in the same order of magnitude. In this latter case, we almost always have

$$\mathcal{N} \subset \mathcal{H} \subset \Omega, \quad \text{s.t.} \quad |\mathcal{N}| \ll |\mathcal{H}| \ll |\Omega|$$

Therefore, we cannot constitute chains of elements in $\mathcal{N}$ by using the function f alone. We need functions which associate elements of $\mathcal{H}$ to elements of the much smaller space $\mathcal{N}$, in other words which “reduce” elements of $\mathcal{H}$ back to $\mathcal{N}$.

Definition 2.6 (Reduction function)**.** *Given a one-way random function $f : \Omega \to \mathcal{H}$, and $\mathcal{N} \subset \Omega$, a function*

$$r : \mathcal{H} \to \mathcal{N}$$

is a reduction function *for f in $\mathcal{N}$ if it preserves the randomness of f, that is, if $r \circ f$ is also a random function.*

Using reduction functions, we can consider a linking function $F = r \circ f$. If we denote by F^k the successive k applications of F, chains can be computed as follows:

$$X_0 \to F(X_0) = X_1 \to F^2(X_0) = F(X_1) = X_2 \to \cdots \to X_t.$$

From a starting point, we use f to get a random element in $\mathcal{H}$, then reduce it to the second link and reiterate, as illustrated in Figure 2.3. The chain’s purpose is to exploit the randomness of the one-way function f to browse the problem space, *i.e.*, to put in relation elements of $\mathcal{N}$. A key point in that regard is the fact that $\forall k > 0$, F^k is random. In particular, the reduction function is chosen so that it does not collapse the randomness of F over successive iterations.

While any function satisfying Definition 2.6 could be used in theory, more considerations, such as for instance speed of computation, are generally looked upon when choosing reduction functions. Indeed,

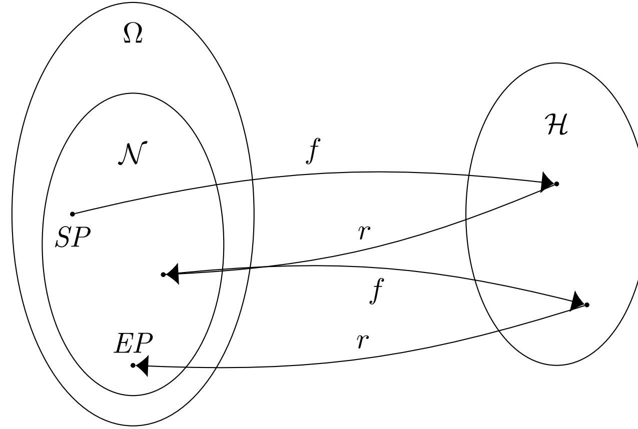


Figure 2.3 – TMTO chain

an important quality that is expected from them is to be fast to compute. It is notably expected to take a negligible amount of time to reduce an element with regard to the one-way function f. Permutations of bits are a good candidate in this regard, since most CPUs have specific instructions to perform them in a few cycles. Similarly, bit truncation and modulo operation (especially with integers in the format 2^k) are performed very quickly.

The usefulness of TMTOs resides in their implementations, and when discussing the choices of reduction functions, we will consider the fact that processors mainly deal with integers. There is no standard way to convert various inputs, such as, *e.g.*, passwords, into numbers. It is up to the developer, for each type of problem space, to come up with two bijective mappings

$$\phi : \mathcal{H} \to \{0, \dots, N-1\}, \quad \text{and} \quad \psi : \{0, \dots, N-1\} \to \mathcal{N},$$

which allow them to process any input using both a one-way function, and a reduction function dealing with integer spaces. We will assume that such mappings exist, which will allow us to conflate a given endpoint $x \in \mathcal{N}$ with its integer representation $\psi^{-1}(x)$. In particular, we consider that the elements of $\mathcal{N}$ can be ordered in the same way as their images by ϕ in $\{0, \dots, N-1\}$. We can then sort sets of elements of $\mathcal{N}$ using this order.

Several reduction functions will often be required by a single TMTO for various reasons, so we will need a family of such functions, ideally parameterised by some integer variable for practical efficiency. An example of a reduction function which can be found in the implementation landscape [ACL15; AC17] is

$$r_k : x \mapsto \psi(\phi(x) + k \mod N),$$

parameterised by k. An equivalently good function proposed in [HM13] is

$$r_k : x \mapsto \psi\left(\phi(x) \oplus k \mod N\right),$$

where $\oplus$ denotes the bitwise exclusive OR (XOR) operation.

2.1.5 TMTO matrix

Most of the pre-computation effort is spent in building a pre-computation matrix, which will serve as basis to construct the TMTO tables. In particular, the relation between the set of elements of this matrix and $\mathcal{N}$ is of particular importance when reasoning about the performance of TMTO.

A set of elements of $\mathcal{N}$ constitutes the set of starting points. Different strategies exist regarding how they should be chosen, such as selecting them at random, or incrementing a counter. Once the starting points are determined, chains are computed from each of them. These chains are then gathered to constitute a matrix as the one illustrated in Figure 2.4. Notice that the starting points constitute the 0th column of the matrix, due to the fact that unlike the rest of the matrix, all the preimage of its elements do not belong to the matrix.

Definition 2.7 (TMTO matrix)**.** *Let $X_1, \ldots, X_m$ be m chains generated in the way previously described, where $\forall k \in \{0, \ldots, m-1\}, \exists s_k \quad X_k = (x_{k,1}, x_{k,2}, \ldots, x_{k,s})$.*

1. *A* pre-computation matrix *is a collection of the chains including the starting points*

$$\overline{TM} = (x_{i,j}) \quad \text{with } i \in \{1, \ldots, m\}, \text{ and } j \in \{0, \ldots, s_i\}.$$

2. *A TMTO matrix is the part of a pre-computation matrix whose elements are invertible,* i.e., *is the restriction to columns $\{1, \ldots, t\}$ of $\overline{TM}$*

$$TM = (x_{i,j}) \quad \text{with } i \in \{1, \ldots, m\}, \text{ and } j \in \{1, \ldots, s_i\}.$$

Note that the assumption that the chains are of a fixed length does not hold for all TMTOs, so this definition accounts for the case where the chains are of variable length. When there is no ambiguity, we will simply refer to a TMTO matrix as "a matrix".

$$\begin{array}{ccccccccc}
SP_1 = x_{1,0} & \xrightarrow{F} & x_{1,1} & \xrightarrow{F} & \ldots & \xrightarrow{F} & x_{1,t-1} & \xrightarrow{F} & EP_1 = x_{1,t} \\
SP_2 = x_{2,0} & \xrightarrow{F} & x_{2,1} & \xrightarrow{F} & \ldots & \xrightarrow{F} & x_{2,t-1} & \xrightarrow{F} & EP_2 = x_{2,t} \\
\vdots & & \vdots & & \ddots & & \vdots & & \vdots \\
SP_m = x_{m,0} & \xrightarrow{F} & x_{m,1} & \xrightarrow{F} & \ldots & \xrightarrow{F} & x_{m,t-1} & \xrightarrow{F} & EP_m = x_{m,t}
\end{array}$$

Figure 2.4 – Pre-computation matrix

The effort of the pre-computation phase corresponds to the mt applications of F, which can often be approximated by f as far as the computation time is concerned, since the reduction function running time is often negligible compared to the f running time. Note that there are often several matrices to compute for a single TMTO. Therefore we have to multiply the computation effort by the amount ℓ of tables.

Due to the randomness of the chains, we cannot guarantee that all of $\mathcal{N}$ is contained in the matrix. An important characteristic of a TMTO is to quantify which percentage of the problem space is covered by a matrix.

Definition 2.8. *The* coverage *of a problem space by a TMTO is the proportion of distinct elements of $\mathcal{N}$ in the TMTO matrix:*

$$\mathcal{C} = \frac{\#\{x \in \mathcal{N} | x \in TM\}}{N}.$$

A full coverage of $\mathcal{N}$ would imply that all the elements appear in the matrix exactly once. Such a matrix is unfortunately prohibitively expensive to construct for most TMTOs.

The coverage is often used to determine the parameters t and m, *i.e.*, how large the matrix should be made during the pre-computation. Indeed, the coverage of the TMTO is completely determined by the relation between t, m, and N. This relation, which indicates when to stop the pre-computation, is introduced in [BS00] and is called the *matrix stopping rule*.

2.1.6 Success rate

TMTO's online algorithm is probabilistic in nature, so it is important to consider its success rate in order to properly evaluate its performance. We discuss this probability, and present the concept of perfect trade-off which maximises it.

We define the *success rate* as the probability of successfully recovering a preimage from a given element in $\mathcal{N}$, during the online phase. The elements that can be successfully inverted by a TMTO are, exactly, all elements $y \in H$, such that $r(y) \in \mathcal{N}$ is in the matrix, and is not a starting point. Indeed the first column cannot be inverted since the preimage of its elements by F is not stored in TM. Therefore, this corresponds to the probability P that a particular element belongs to the matrix:

$$P = \Pr(x \in TM | x \in \mathcal{N}).$$

This probability corresponds to the coverage of the TMTO.

One cannot expect to cover $\mathcal{N}$ with only a single matrix, without spending a prohibitive time in pre-computation. Indeed, F is random, and common values for m exceed $\sqrt{N}$, the well-known birthday paradox bound for collisions. There is therefore a high probability of collision for the function F, which results in duplicates in the matrix.

Merging chains The consequence of the collisions of the chains is a cornerstone of the behaviour of both the offline and the online phase of TMTOs. When a collision of the reduction function happens between two elements of distinct chains, this implies that the rest of the two chains, after the collision, will be constituted of the same elements. We call this phenomenon *chain merging*, as illustrated in Figure 2.5 showing two chains that merge at some point. The number and the position of these merges greatly alter the coverage of a given matrix.

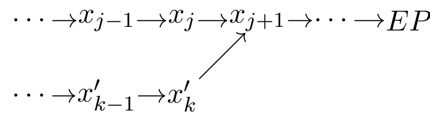


Figure 2.5 – Merging chains

A way to maximise the success rate of a TMTO is to ensure that there is no such merge in the matrix. Such a merge-free matrix is called a *perfect matrix*[3]. Due to the random nature of the TMTO, such a configuration is obtained by trial-and-discard. This works by discarding all but one of the chains leading to the same endpoint, resulting in a significantly higher pre-computation cost in order to get the same amount of chains as we would in a non-perfect matrix. Indeed, given some perfect matrix of increasing size, it is less and less probable to find a new chain that does not collide with all the chains that are already in the matrix. This means that, a given amount of computation cost yields less and less additional coverage when building a matrix.

[3]Some of the literature suggested using the more adequate term "*clean*" instead of "perfect" to designate such matrices.

2.1.7 Tables

The real form of the TMTO's online phase data structure is the *table*. This is what will actually have to be stored in memory. Tables are extracted from a given TMTO matrix by keeping the first and the last columns only. In other words, it is simply a sequence containing $(EP_i, SP_i)_{1 \leq i \leq m}$ pairs, such as depicted in Figure 2.6. The main point of a table is to associate an endpoint to a starting point which is on the same row in the matrix. Any structure that can provide this association can be used as a table.

$SP_1 = x_{11}$	$EP_1 = x_{1t}$
$SP_2 = x_{21}$	$EP_2 = x_{2t}$
$\vdots$	$\vdots$
$SP_m = x_{m1}$	$EP_m = x_{mt}$

Figure 2.6 – Table

Note that all other elements in the matrix are indeed discarded in the table. This is the reason why TMTO tables storage requires much less memory than a dictionary covering the same problem space, which would require to keep every pair $(h(x), x)$, for every $x \in \mathcal{N}$. If we suppose that the starting point and the endpoint are of similar sizes, the gain ratio compared to the matrix is about $\frac{2}{t}$, which becomes significant when the value of t increases enough.

The memory aspect of the trade-off also becomes clear, as it can already be seen that we can arbitrarily gain space for the table, *i.e.*, reducing m, while keeping a constant number of elements in the matrix. It suffices to compute longer chains, *i.e.*, to increase t, when building the table. Obviously this will affect the online running time which depends on t.

In order to evaluate the redundancy present in a table, we can rely on the coverage rate, not to be confused with the coverage of the pre-computation matrix defined above.

Definition 2.9 (Coverage rate)**.** *The coverage rate of a TMTO table is the amount of average distinct element in its tables:*

$$\mathcal{C}_r = \frac{\#\{x \in TM\}}{mt}.$$

Maximal tables One can try to maximise the coverage rate. A possible technique is to use all of the N possible starting points to produce N chains which are then compared, and where in each set of merging chains, all but one (arbitrarily chosen) are discarded. Such a matrix constructed from N points is called a *maximal matrix* and it gives a *maximum table*. Note that such a table involves Nt applications of F during the pre-computation phase, which is a significant cost when N is large.

It will be shown in the next sections, that the maximum probability of success of a TMTO is bounded by a value $P_{\max} < 1$, with $P_{\max}$ depending on the particular TMTO technique considered. In order to further increase the success probability of a given TMTO, we can compute several distinct tables on the same problem space $\mathcal{N}$. In particular, the F functions have to use different reductions functions across the tables. If we compute ℓ such tables with a success probability of P for each table, we obtain a success rate for the TMTO of

$$P^* = 1 - (1 - P)^{\ell}.$$

We can therefore obtain a success rate that is arbitrarily close to a 100%, given enough pre-computation of a sufficient number of tables.

After these general considerations on TMTOs, we will now explicitly establishing the relation between the different parameters of the trade-off. The following sections introduce the original TMTO instance, alongside some variants which improve it. Notice that we did not explain what happens in the online phase yet. This has been deferred to these sections because it differs between the variants in such a way that it was not possible to present a generic, unified algorithm that encompass them all.

2.2 Hellman TMTO

In this section, we describe the seminal work of Hellman that was published, in 1980, in [Hel80]. We introduce some of the points of interest of the study of TMTOs, as [Hel80] is a basis for of all the developments about TMTOs in the literature until now. In particular, we present the inversion procedure, which is the key of the online phase of the TMTO.

The original cryptanalytic TMTO introduced in the work of Hellman is referred to as the *Hellman TMTO*, or simply the *Hellman trade-off*, in the literature. In [Hel80], Hellman presents it in the context of the cryptanalysis of DES, in a chosen plaintext attack. The main contribution of the article is to provide a way to reduce the cost of cryptanalysis. In particular, Hellman claims that it allows to cut down the cost of a DES key recovery from \$5000, when using a traditional brute force approach, to a mere \$10 using the TMTO.

2.2.1 Offline phase

We consider a problem space $\mathcal{N}$, and denote with m our memory, *i.e.*, the number of chains we want to include in our tables, ℓ the number of tables, and t the length of the chains. The Hellman trade-off uses a different reduction function r_i for each table. We denote with Γ the following application

$$\Gamma : K \mapsto \mathit{DES}_{\mathit{enc}}(K, P),$$

which associates each DES key to the encryption of a fixed known plaintext P.

We now describe the pre-computation phase.

Pre-computation During the pre-computation, in each of the ℓ tables, m elements of $\mathcal{N}$ are chosen as starting points. It is suggested in [Hel80] to draw them uniformly in $\mathcal{N}$. Note that this choice does not influence the resulting matrix much, as the starting points themselves do not belong to the set of elements whose image by the one-way function Γ can be inverted.

From $SP_0, SP_1, \ldots, SP_{m-1}$, m chains of length t, *i.e.*, $t-1$ applications of the function F_l, were l denotes the lth table, are performed, from which we obtain m endpoints $(EP_0, EP_1, \ldots, EP_{m-1})$. A TMTO matrix TM is obtained from the pre-computation, and we denote x_{ij} the jth element of the ith chain. The starting points and the endpoints of each chain are put into pairs (SP_i, EP_i). The pairs are then sorted according to the endpoints, in such a way that, for a table

$$\{SP_i, EP_i\}_{i=1}^{m},$$

we have $EP_1 \leq EP_2 \leq \cdots \leq EP_m$. This will allow the algorithm of the online phase to efficiently access a given endpoint.

From m starting points, the application of F is expected to induce me^{-1} collisions. This means that there are only $m(1-e^{-1})$ distinct elements in the second column, which are able to constitute a chain, as the elements of the first column do not count towards the coverage of $\mathcal{N}$. In other words, a single Hellman table cannot expect to cover more than $\mathcal{C} = (1-e^{-1})$ of $\mathcal{N}$. This implies that several tables $\ell > 1$ are needed in order to get to a higher success rate. In fact, it is often a key used to determine the parameters t and m of the trade-off. In his work, Hellman also proposes a lower bound on the success rate of a table.

Theorem 2.2 (Success rate lower bound)**.** *A single Hellman table of m chains of length t is expected to cover $\mathcal{N}$ by at least*

$$P \geq \frac{1}{N} \sum_{i=1}^{m} \sum_{j=1}^{t} \left(\frac{N-it}{N} \right)^{j+1}.$$

Proof. If we pick all the mt elements of TM, from $\mathcal{N}$, at random, starting from x_{11} and going row after row, the amount of distinct elements is the sum of all the probabilities that each of the x_{ij}, for $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, t\}$, is not already in the matrix, either in a previous row or thus far in this row, *i.e.*, that it is *new*. The coverage is given by the ratio on N of this amount of distinct elements.

$$P = \Pr(x \in TM) = \frac{1}{N} \sum_{i=1}^{m} \sum_{j=1}^{t} \Pr(x_{ij} \text{ is new})$$

We have

$$Pr(x_{ij} \text{ is new}) \geq Pr(x_{ij} \text{ is new in this row})$$

We know from the Birthday paradox that, given n distinct elements in N already present, an element is new with probability $\frac{N-n}{N} = (1 - \frac{n}{N})$. Then,

$$\begin{aligned} \Pr(x_{ij} \text{ is new in this row}) = \quad & \Pr(x_{i0} \text{ is new}) \times \Pr(x_{i1} \text{ is new} | x_{i0} \text{ is new}) \times \ldots \\ & \times \Pr\left(x_{i(j-1)} \text{ new} | x_{i0}, x_{i1}, \ldots, x_{i(j-1)} \text{ are all new}\right) \\ = \quad & \prod_{k=0}^{j} \left(1 - \frac{i(t-1)+k}{N}\right) \end{aligned} \tag{2.4}$$

Each factor of (2.4) is larger that $(1 - \frac{it}{N})$ since there are at most t new elements per row, so we have

$$\Pr(x_{ij} \text{ is new in this row}) \geq \prod_{k=0}^{j} \left(1 - \frac{it}{N}\right) = \left(1 - \frac{it}{N}\right)^{j+1}.$$

□

In [Hel80], Hellman makes the choice of a balanced configuration. He suggests computing each table with parameters $t = m = N^{1/3}$ (denoted by capital letters T and M in [Hel80]). A matrix stopping rule is also given:

$$mt^2 \approx N.$$

This is justified by the lower bound established on the success rate and stated in Theorem 2.2. When t and m are large, we can approximate the lower bound by $e^{\frac{-mt^2}{N}}$, which justifies the matrix stopping rule given above: increasing the product mt^2 increases the coverage, but the gain becomes small once

the product exceeds N, whereas the pre-computation remains costly. Hence, a common rational choice is to stop the computation when we reach $mt^2 \approx N$. Note that P is independent from m, t, and N, as long as the matrix stopping rule holds. Since the coverage is assured by ℓ tables of mt elements, that is $mt\ell$ elements in total, the rule indicates that a better performance is achieved when computing a number $\ell \approx t$ of tables.

In [KM96], Kusuda and Matsumoto show that the coverage rate of a Hellman table can be computed with the formula

$$\frac{1}{t}\int_0^{\frac{mt^2}{N}} \frac{1-e^{-x}}{x}\,dx$$

which, when $mt^2 \approx N$, evaluates to 80%. A refined bound appears in [MH08], where Ma and Hong provide a closed form formula:

Theorem 2.3. *The expected coverage rate C_h of a single Hellman table of size and length parameters m and t is given by*

$$\mathcal{C}_h = \frac{N}{mt} s_t$$

where

$$s_0 = 0, \quad s_{i+1} = 1 - e^{-\frac{m}{N}} e^{-s_k}.$$

When the matrix stopping rule $mt^2 = N$ holds,

$$\mathcal{C}_h = t \cdot s_t.$$

Proof. We denote $p_k = \frac{m_k}{N}$, where m_k is the number of distinct elements at column k of TM. If we start with m starting points, we expect the first column of valid pre-images to contain a proportion

$$p_1 = \left(1 - e^{-\frac{m}{N}}\right)$$

of distinct elements. At the $(k+1)$-th column, there are $N(1 - e^{-\frac{m_k}{N}})$ distinct elements. A fraction of these can be found in any of the previous columns. The proportion of elements that have not yet appeared is $(1 - \sum_{j=1}^{k} p_k)$, so we have

$$m_{k+1} = \left(1 - \sum_{j=1}^{k} p_j\right) N(1 - e^{-\frac{m_k}{N}}),$$

which can be rewritten

$$p_{k+1} = \left(1 - \sum_{j=1}^{k} p_j\right)(1 - e^{-p_k}). \tag{2.5}$$

We now denote $s_k = \sum_{j=1}^{k} p_j$, and we have $s_1 = p_1$. According to [MH08], Equation (2.5) can be rewritten:

$$s_{k+1} = 1 - e^{-\frac{m}{N}} e^{-s_k}.$$

So we have $s_t = \sum_{j=1}^{t} \frac{m_j}{N}$, from which we can obtain the stated formula for the coverage rate.

Under the matrix stopping rule, $N = mt^2$, so $\frac{N}{mt} = 1$, which leads to the second statement. □

In order to improve the coverage of a Hellman TMTO, one may be tempted to build perfect Hellman tables. We now discuss the difficulty of building such tables.

Perfect tables The perfect version of the Hellman TMTO is barely discussed in the literature. A perfect Hellman table consists of merge free chains such that for each chain of the table, there is no other chain that merge with it. One mention of perfect Hellman tables is in [AJO08] where Avoine, Junod and Oechslin state that the success rate of a perfect Hellman table is $\frac{mt}{N}$. The computation of such a table is a difficult task. A naive way would be to compute chains and compare each new added link to the chain to all the already computed elements in the matrix. This involves in the order of N^2 comparisons which is beyond the realm of what is possible with any practical TMTO problem size. Another method, proposed in [AJO08] is to compute the longest chains possible until a merge with an already computed chain occurs. When this happens, a new starting point is chosen at random and a new chain is started. The chains are then split into smaller chains of size t which are guaranteed to be collision free. Nevertheless, no analysis of the cost of the method is made in [AJO08]. However, the authors perform an experiment which tends to show that the maximum number of chains of length t is approximately proportional to $\frac{1}{t^2}$.

Even without seeking to obtain perfect tables, one can wonder whether discarding duplicate endpoints would be beneficial when computing tables. A remark of Hong in [Hon10] tells us that this is not the case, at least when the matrix stopping rule $mt^2 = N$ is considered. Recall from Theorem 2.1 that the last column gives us

$$m_t = \frac{2N}{\frac{2N}{m_0} + t} = \frac{m_0}{1 + \frac{m_0}{2N}t} = \frac{m_0}{1 + \frac{m_0 t^2}{N}\frac{1}{2t}},$$

and with the matrix stopping $\frac{m_0 t^2}{N} = 1$, so with the power series development of m_t, we have

$$m_t \approx m_0 + \frac{m_0}{2t} + \cdots \approx m_0,$$

when $t \gg 1$, so the amount of duplicate elements is low in the Hellman table, which means that the gain obtained by discarding merging chains is low as well.

2.2.2 Online phase

We now describe the online part of the Hellman TMTO. This is the algorithmic part of $\mathcal{A}^f_{\mathcal{N}}$ used to perform the inversion, the other part being the table computed in the offline phase.

Let us first explain how we can recover elements of the matrix from the table. The idea is to determine where a given element $x_0 \in \mathcal{N}$ would be in the matrix $TM = (x_{ij})$. Assume we have $x_{i_0 j_0} = x_0$, for some (i_0, j_0), and we want to determine i_0 and j_0. We know that EP_{i_0}, which is at the rightmost end of the matrix row, is in the table. Now, the key idea is that if we can find a *potential endpoint* $PEP = F^{(t-j_0)}(x_0)$ in the table, for some j_0, the position of PEP in the table gives us i_0. So we just need to find j_0. This can be achieved by computing $F^i(x_0)$ for all $0 < i < t-1$, *i.e.*, computing all *potential chains* of all sizes i, until a potential endpoint:

$$\underbrace{x_0 \xrightarrow{F} x_1 \xrightarrow{F} \dots \xrightarrow{F} x_{i-2} \xrightarrow{F} x_{i-1}}_{\text{length } i} = PEP,$$

and check whether PEP belongs to the table. When such a match occurs, it is called an *alarm*.

False alarms Unfortunately, a potential endpoint matching an endpoint in a table is not equivalent to our element belonging to the matrix. Recall that F is random and prone to collisions, which implies that

the potential chain can be merging, like it already happens during the computation of the matrix. In particular, a collision of a potential chain with a chain in the matrix can happen. When such a collision in the chains happens, it is said to be a *false alarm*. This implies that a match of a potential endpoint is not enough to conclude the search, and additional work has to be performed to check whether the element belongs to the pre-computation table.

We now describe the inversion algorithm for a single table. When several tables are present, we process them sequentially. The full algorithm is summarised in Algorithm 2.1. We assume that we are given a $y_0 \in \mathcal{H}$ whose preimage $x_0 \in \mathcal{N}$ exists in the matrix:

$$\exists (k_0, m_0), \quad \text{s.t.} \quad F^{k_0}(SP_{m_0}) = x_0.$$

Our goal is to find x_0 while only knowing y_0. We first apply the reduction function r to get $r(y_0) \in \mathcal{N}$. Since the table is sorted by the endpoints, we can efficiently check if $r(y_0)$ belongs to the table, for example with a binary search algorithm which will find our element in $\log_2(m)$ comparisons. If $r(y_0)$ is not among the endpoints of the table, we apply F on our element, and check again if the obtained element is among our endpoints. What we are actually checking, with this second check, is whether our element was in the second to last column of the matrix. We then iterate in order to check the presence of our element in every column of the matrix. When we find our element among the endpoints, we get an *alarm*. Either we have found our $r(y_0)$ in the matrix, or we happen to have a y'_0 such that $r(y'_0) = r(y_0)$, but with $y_0 \neq y'_0$, *i.e.*, a collision[4] on r. To distinguish between false and good alarms, we have to check whether we can rebuild the chain up to y_0. In order to do so, when we find an endpoint EP_i (where we might or might not have $i = m_0$), such that it is matching our $r(y_0)$ after k iterations, we compute $x_1 = F^{t-k-1}(SP_i)$, and check whether $f(x_1) = y_0$.

Algorithm 2.1 Online inversion in the Hellman trade-off

```
x_1 ← r(y_0)
for i = 1 to ℓ do
  for k = 0 to t do
    x ← r ∘ f(x_1)
    if ∃i  s.t.  x = EP_i then
      x_0 ← r ∘ f^{t−k−1}(SP_i)
      if f(x_0) = y_0 then return x_0
      end if
    end if
  end for
end for
```

In the case the given x_0 is not in TM, the algorithm will fail at every column and stop at the ℓtth iteration, when all the columns of all the tables have been processed.

2.2.3 Analysis of the Hellman online phase

We now present some results related to the performance of the algorithm $\mathcal{A}^f_{\mathcal{N}}$. A point of interest is the average running time complexity of the online phase algorithm, which will allow us to know the expected time to invert any particular value. It especially makes sense to consider the average running

[4]Note that a collision on f is unlikely in general, if we chose f to be a cryptographic primitive.

time compared to the worst case when the success rate is close to 100%, but remains useful in the general case as a TMTO is expected to be run many times to amortize the huge pre-computation cost.

A sometime neglected consideration, in particular in early works on TMTOs, such as the original article of Hellman [Hel80], is the amount of work which is not directly useful to the cryptanalysis: the computations of F performed in order to verify whether the endpoint we found is associated with the starting point that actually led to our preimage. This work is denoted with Q_h hereafter. The original paper [Hel80] gives an upper bound to the work due to false alarms, where it is said to be at most half of the cryptanalytic effort. In [Hon10], a more precise evaluation is given: during the online phase, the expected additional computations of F that are caused by false alarms is given by

$$Q_{\mathrm{h}} = \frac{t(t+1)(t+2)}{6} \cdot \frac{m}{N}.$$

The tables are searched sequentially. The preimage has an equal probability to be in any of the tables, therefore the probability of finding the preimage in a given table, and not having found it in any of the previous tables, follows a geometric distribution. Each search is then expected to require T_h applications of the one-way function, where T_h is given in Theorem 2.4.

Theorem 2.4. *We consider a Hellman TMTO, in ℓ tables of size m and chains of length t. The expected number of applications of the one-way function, before finding the preimage, or failing to do so, is given by*

$$T_h = \sum_{i=1}^{\ell} \left(1 - \frac{\mathcal{C}_h mt}{N}\right)^{i-1} (Q_h + t).$$

Proof. Let denote $\forall i \in \{1, \ldots, \ell\}, F_i$, the one-way function used in table i. The probability of a single table to yield the answer is the probability that the answer belongs to on of the distinct points of the TMTO matrix TM. TM possesses mt elements among which $mt\mathcal{C}_{\mathrm{h}}$, where C_h is the coverage rate of the table, are distinct. So the probability is

$$\frac{mt\mathcal{C}_{\mathrm{h}}}{N}.$$

It follows that the probability that the answer was not found before the ith table is

$$\left(1 - \frac{mt\mathcal{C}_{\mathrm{h}}}{N}\right)^{i-1}.$$

In the ith table, t applications of F_i for are needed to build the potential chain, plus an expected Q_{h} applications of F_i to resolve the false alarms. □

We now discuss the asymptotic behavior of the trade-off, and in particular the relation between N, M and T.

Trade-off curve It is noted in [Hel80] that when the one-way function considered is a cyclic permutation, *i.e.*, $\mathcal{N}$ is covered by a single long chain, a table can be computed with $\sqrt{N}$ evenly distributed starting points, and we obtain a TMTO following the curve

$$TM = N.$$

A function which, on the contrary, reduces the collisions (*e.g.*, $F : x \mapsto x$), would be easy to invert, but such a function would certainly not qualify since it would not be both random and one-way. In the general case though, the Hellman TMTO follows the following asymptotic curve:

$$TM^2 = N^2.$$

Note that, since the objective of a TMTO is to perform better than an exhaustive search, we must have $T < N$ and therefore the memory of a given TMTO is bounded by

$$M = \frac{N}{\sqrt{T}} < \frac{N}{\sqrt{N}} = \sqrt{N}.$$

2.2.4 Variants

We just described the original TMTO method, introduced by Hellman in [Hel80], in 1980. Ever since, methods improving original technique were proposed in the literature. We focus on the main variants of this method that got more attention from the literature.

What is referred to as *variants* in this thesis is a particular class of improvements which modify the chain structure significantly enough that the structure of the table differs from the Hellman scheme. In particular, the coverage of $\mathcal{N}$ by the matrix cannot be computed in the same way as with the Hellman trade-off anymore. We distinguish them from the other improvements, since each variant comes with its own parameters considerations which are incompatible with the other variants, whereas general improvements can often apply to any of the variant (or at least to more than one). We reserve the review of the other improvements, which we call *optimisation improvements* for Chapter 3.

In the following sections, we introduce the two main variants of the Hellman trade-off, namely the distinguished point trade-off in Section 2.3, and the rainbow trade-off in Section 2.4, which are the most considered in the literature, with the latter being the most used in practice. The less known, more recent fuzzy trade-off is also presented in Section 2.4, for completeness.

2.3 Distinguished points

The *distinguished points* (DP) trade-off originates from an idea suggested by Rivest, which is mentioned by Denning in [Den82]. Although not introduced by itself in a publication, the concept got further attention in the literature in [SRQL03; HJK+08; HLM11], along with a proper analysis in [BPV98; AJO08; HM13] and [LH16].

The main goal of Rivest with the DP trade-off was to reduce the number of disk access operations during the online phase of the Hellman trade-off. Indeed, in the Hellman trade-off, a comparison of a potential endpoint with the endpoints in the table is made at each column. The DP trade-off allows the online phase to be performed with only a single access operation.

The technique relies, as its name suggests, on some special elements of $\mathcal{N}$, which are called distinguished points. A *distinguished point property* (DP property) is some condition on elements of $\mathcal{N}$, that can be easily verified. An example of such a condition is to require the element to be smaller than a certain target. Equivalently, the last n bits of the element must be divisible by 2^n for some integer n, so that the last n bits of its representation in memory are zeros. The main idea is to build chains iterating a function F, but to use the DP property on the elements of $\mathcal{N}$ as a condition to terminate a chain, thus a DP encountered during the creation of a chain becomes its endpoint.

This establishes a difference with the Hellman method as the chain length is in this case deterministic. When dealing with a DP trade-off, we can no longer rely on the t parameter, which was the length of a Hellman table. Instead, we denote with $\hat{t}$ the average length of a chain. This value is a target value determined during the offline time of the DP TMTO. As in the Hellman trade-off, we denote with m the number of chains in a DP table, and we assume both $m \gg 0$ and $\hat{t} \gg 0$.

A matrix stopping rule can be stated for this trade-off, which is nearly identical to that of Hellman:

$$m\hat{t}^2 \approx N,$$

where N is the size of $\mathcal{N}$. Note that, once again, we keep $\hat{t} \ll \sqrt{N}$ in order to reduce the number of collisions of the reduction function.

While the DP trade-off and the Hellman trade-off are different in nature, the asymptotic curve on which the DP trade-off lies with regard to the relation between the memory M, the problem space N, and the time T, is the same as the Hellman trade-off, *i.e.*,

$$TM^2 = N^2.$$

2.3.1 Pre-computation

The set of chains of varying length computed during the offline phase is gathered in a *pre-computation DP matrix*. During the pre-computation of the DP matrix, we chose starting points which do not have the DP property. Then, we compute chains such that all endpoints are distinguished points, and no intermediate point can be a distinguished point. The chain construction considers endpoints as elements of $\mathcal{N}$ whose application by f gives an element of $\mathcal{H}$ which has the DP property. In addition to the starting point and the endpoint, the length λ at which the DP is found is stored in the table as illustrated in Figure 2.7.

$$\begin{array}{ccc} EP_1 & \Rightarrow & (SP_1, \lambda_1) \\ EP_2 & \Rightarrow & (SP_2, \lambda_2) \\ \vdots & & \vdots \\ EP_m & \Rightarrow & (SP_m, \lambda_m) \end{array}$$

Figure 2.7 – DP Table

The fact that the chains are no longer of predictable size alters both the offline and online phases, and the analysis of both running times becomes more complex. Such an analysis is done extensively in [SRQL03], [BPV98], and [HM13]. We will only present the main results and points of interest. When relying on a DP property, the reduction function may collide with an element already in the chain. This problem can also happen in a Hellman table, with the consequence that the chain will contain redundant information. In the DP trade-off though, it implies that the computation of the chain does not end. To overcome this issue, a threshold beyond which we give up on computing that particular chain is fixed. That is, we determine a factor ν so that we perform no more than $\nu\hat{t}$ computations of F in any chain, be it during the construction of the pre-computation matrix during the offline phase, or the extension of a chain during the online phase. The parameter ν must be determined so as to minimise the computational effort. If it is too low, a lot of chains will be discarded as there will be a lot of chains which fail to reach

a DP within $\{1, \ldots, \nu\hat{t}\}$. If ν is too high, the online phase will be slower since it is in part determined by the size of the longest chain in the table, which is higher in average when ν increases. As noted in [BPV98], this provides a great advantage to the trade-off in the sense that the tables are guaranteed to be free of cycles. Furthermore, the merges are automatically detected without additional operations, as two colliding chains will have the same ending DP and therefore one only has to verify that the DP does not belong to the current table. This simplifies the creation of perfect DP tables.

2.3.2 Success rate

The success rate of the DP trade-off is determined by the coverage of $\mathcal{N}$ by the DP matrix, that is, the number of distinct elements of $\mathcal{N}$ found in the DP matrix excluding the DP. The DP property is determined to have a probability of $\frac{1}{\hat{t}}$ to happen when picking an element of $\mathcal{N}$ at random. We have a probability of $\left(1 - \frac{1}{\hat{t}}\right)$ to not get a DP at each step, meaning the probability of a chain to reach a distinguished point within $\nu\hat{t}$ iterations of F is

$$\left(1 - \frac{1}{\hat{t}}\right)^{\nu\hat{t}}.$$

We can see that the probability converge exponentially towards 1 with ν increasing, which implies that even a low value allows us to keep most of the computed chains. For example, with $\nu = 5$, more than 99% of all of the computed chains contain a DP. In any case, starting from k starting points, we can expect to get

$$k\left(1 - \left(1 - \frac{1}{\hat{t}}\right)^{\nu\hat{t}}\right)$$

candidate chains. Similarly to how we determine m_0 in the Hellman trade-off, we can achieve a target of m_0 DP chains by computing

$$m_0' = \frac{m_0}{1 - \left(1 - \frac{1}{\hat{t}}\right)^{\nu\hat{t}}}$$

starting points. Note that by the Taylor series of $x \mapsto \frac{1}{1-x}$, given by $\sum_{n=0} \infty x^n$, and considering ν to be sufficiently large to state that $\left(1 - \frac{1}{\hat{t}}\right)^{\nu\hat{t}} \approx e^{-\nu}$, we can roughly approximate the overhead of the additional needed computations by

$$\frac{m_0'}{m_0} \approx e^{-\nu},$$

which corresponds to a negligible amount of additional pre-computations, even for moderate values of ν.

In [SRQL03], Standaert, Rouvroy, Quisquater and Legat make the observation that there exists a range $[t_{\text{min}}, t_{\text{max}}]$ of chain length, such that the probability of generating a DP chain whose length is in this range is higher. Keeping chains of length l such that $t_{\text{min}} < l < t_{\text{max}}$ is more practical for the pre-computation since it minimises the ratio

$$\frac{\text{overhead due to raising } \nu}{\text{coverage of the DP matrix}},$$

while limiting the computation of useless extra chains.

The expected coverage rate of a DP table, denoted with $\mathcal{C}_{\text{dp}}$ hereafter, depends only on the matrix stopping rule constant, wich we denote γ in the following. An approximation of the coverage can be found in [HM13] which gives the following formula

$$\mathcal{C}_{\text{dp}} = \frac{2}{\sqrt{1+2\gamma}+1},$$

itself derived from a more complex evaluation of the exact value of the coverage rate in the same article. This formula does not take into account the DP themselves at the end of the chains.

A perfect matrix and table are obtained by requiring distinct DP in the matrix. Nevertheless, chains ending with the same DP are equally interesting in the trade-off due to their varying length. Indeed, if we want to maximise the coverage, we cannot simply ignore any newly found duplicate as could be done in the classical trade-off. A common strategy is then to keep the longest chain among those ending with the same endpoint.

2.3.3 Online phase

We now describe the online phase of the DP variant, whose procedure is given in Algorithm 2.2. As usual, we start with an element $y_0 \in \mathcal{H}$, which is first reduced to get some $x_1 = r(y_0) \in \mathcal{N}$. We then apply the $r \circ f$ function until we find a distinguished point. Either the distinguished point is not among the endpoints, and we know for sure that our x_1 is not covered by the table, or there is at least one matching endpoint, in which case we got an alarm that we have to resolve. Note that the length of the longest chain λ_l computed in a given table l can be stored as metadata, which allows us to save computations during this part of the algorithm as $\lambda_l < \nu\hat{t}$. The verification then consists in rebuilding the partial chain from the corresponding starting point, and the associated DP length. We find either x_1, a different element of $\mathcal{N}$, or another distinguished point earlier in the chain. In both latter cases, we know that the preimage is not covered by the DP matrix of this particular table, and we perform the same procedure on the next table.

Algorithm 2.2 Online inversion in the DP trade-off

```
x_1 ← r(y_0)
for i = 1 to ℓ do
  k ← 0
  while k < νt  AND  x_1 is not a DP do
    k ← k + 1
    x ← r ∘ f(x_1)
  end while
  if ∃i  s.t.  x = EP_i then
    x_0 ← SP_i
    for j = 1 to λ_i − k do
      x_0 ← r ∘ f(x_0)
    end for
    if f(x_0) = y_0 then return x_0
    end if
  end if
end for
```

2

The online phase requires only one search per table, whereas the search had to be done at each column in the Hellman version. Nevertheless, this gain comes at a cost of either an additional storage of the λ_i, or longer computations, if we were to have a potential chain of length $\nu\hat{t}$ each time.

2.3.4 Analysis

In the next paragraphs, we detail the average running time of the online phase of the DP trade-off, including the cost related to false alarms.

False alarm As in the Hellman trade-off, there are false alarms in the online phase of the DP trade-off. Collisions of the reduction function may merge DP chains the way that two elements lead to the same DP. The work [HM13] quantifies the average amount of false alarms. We develop in the following the main lines of this result.

Let us assume that we a given inversion will be successful, *i.e.*, there is at least one occurrence of the preimage in the TMTO matrix, in what we call hereafter a correct chain. Let $(x_i)_{1<i<t}$ be the elements of the correct chain which contains the answer of our current problem. The probability of a second chain $(x'_i)_{1<i<t'}$ to merge with the correct one is the probability that the last element of this chain is the same DP and that every other element is neither a DP itself, nor an element of the correct chain but with an incorrect offset. These two latter conditions have respectively a probability of

$$\frac{1}{t} \quad \text{and} \quad \frac{\text{length of overlap}}{N}$$

to happen. The probability $q_{t,t'}$ of a false alarm is therefore the sum of the probabilities of every chain of size between 1 and t' to merge:

$$q_{t,t'} = \begin{cases} \displaystyle\sum_{i=0}^{t'} \frac{\left(1 - \frac{1}{t} - \frac{t'}{N}\right)^i}{N} & \text{if} \quad t \geq t', \\ \displaystyle\sum_{i=0}^{t} \frac{\left(1 - \frac{1}{t} - \frac{t}{N}\right)^{(t'-i)}}{N} & \text{otherwise.} \end{cases}$$

We denote L_n the proportion of chains of length n among the computed chains of the table. These are the chains that do not exceed the limit $\nu\hat{t}$, and that reach their DP after the jth iteration without reaching any DP before. The first condition is similar to the formula of the success rate, the second is given by the geometric law that the chain follows:

$$L_n = \frac{1}{1 - e^{\nu}} \frac{1}{t} \left(1 - \frac{1}{t}\right)^{j-1}.$$

The amount of verification work given by each false alarm is either the distance until the DP from the start of the tried chain of length, say i, or the length j of the chain which merges, if it is longer. The total amount of work due to the false alarms is therefore the sum of all the possible merges of chains of all possible lengths:

$$Q_{\hat{t},m} = \sum_{0<i,j\leq\hat{t}} mL_j q_{i,j} \min(\hat{t} - i + 1, j).$$

Online time With the cost of false alarms established, we can now compute the total amount of time required by the online phase. In a DP trade-off with ℓ tables, the tables are processed sequentially. We get to the nth table if none of the previous tables contains our element. That happens with a probability

$$p_n = \left(1 - \frac{\mathcal{C}_{\text{dp}}}{N}\right)^{n-1}.$$

For each table, we perform the search until the next DP, which takes $t\left(1-e^{-\nu}\right)$ operations. Then, the amount of computation of f performed in the online phase is given by

$$T_{\text{DP}} = \sum_{i=1}^{\ell} p_i \left(t\left(1-e^{-\nu}\right) + Q_{\hat{t},m}\right). \tag{2.6}$$

2.4 Rainbow trade-off

We now describe the rainbow trade-off, which is a variant of the Hellman trade-off introduced by Oechslin in [Oec03]. This variant is probably the most used, both in the scientific literature, and in practice. In particular, rainbow tables are a well established name in computer science, especially in the field of password cracking.

Soon after the publication of [Oec03], Oechslin released the tool Ophcrack[Oec04; TO05]. It provides the means to assess the performance of the trade-off. In particular, the experiment in [Oec03] is implemented, and can be evaluated via the tool. It consists of tables allowing to recover each possible Windows XP password, from its LAN Manager (LM) hash. The tables themselves fit in two CDROMs of 700MB capacity each.

This technique aims, as an improvement over both the Hellman and the DP trade-off, at reducing the number of one-way applications during the online phase. Indeed, a limitation of the Hellman trade-off is that if two chains collide anywhere in a same table they merge. The rainbow trade-off partially overcomes this problem by using several reduction functions in the same table instead of a single one in the Hellman trade-off. This improvement also come without a significant overhead like in the DP trade-off which involves dealing with non constant lengths of tables.

A chain in the rainbow trade-off is computed as in the Hellman case, but the reduction function of the linking function changes at each step, so a chain of length t, using linking functions $F_i = r_i \circ f$, for $i \in \{0, \ldots, t-1\}$ looks like

$$SP = X_0 \xrightarrow{F_0} X_1 \xrightarrow{F_1} X_2 \xrightarrow{F_2} \ldots \xrightarrow{F_{t-1}} X_t = EP.$$

While not explicitly acknowledged in [Oec03], The “rainbow” name comes from the fact that for each column of the tables, a different colour (*i.e.*, reduction function) is used. This terminology is otherwise reused by Barkan, Biham and Shamir in [BBS06] with a clearer context.

2.4.1 Offline phase

The TMTO matrix is built somewhat in the same way as the Hellman chains, except for the fact that multiple reduction functions are used for each chain. A set of chains of length t is computed from arbitrary starting points, typically a counter, as in the Hellman method. Per table, instead of a single

function r, we have a set of t functions $r_0, r_1, \dots, r_{t-1}$, such that $r_i \neq r_j$ for $i \neq j$. An example of a rainbow matrix is shown in Figure 2.8.

This implies a need to generate easily distinct reduction functions, as there will be a total of ℓt different functions necessary in an application of the trade-off, in the case of ℓ tables. A common way of generating reduction functions is to increment the offset at each column. That is, the reduction function is function of both the table and the column, *i.e.*, we reduce the kth element of the nth table with the function r_{tn+k}.

$$\begin{array}{ccccccccc}
SP_1 = x_{1,0} & \xrightarrow{r_1 \circ f} & x_{1,1} & \xrightarrow{r_2 \circ f} & \dots & \xrightarrow{r_{t-2} \circ f} & x_{1,t-1} & \xrightarrow{r_{t-1} \circ f} & x_{1,t} = EP_1 \\
SP_2 = x_{2,0} & \xrightarrow{r_1 \circ f} & x_{2,1} & \xrightarrow{r_2 \circ f} & \dots & \xrightarrow{r_{t-2} \circ f} & x_{2,t-1} & \xrightarrow{r_{t-1} \circ f} & x_{2,t} = EP_2 \\
\vdots & & \vdots & & \ddots & & \vdots & & \vdots \\
SP_j = x_{j,0} & \xrightarrow{r_1 \circ f} & x_{j,1} & \xrightarrow{r_2 \circ f} & \dots & \xrightarrow{r_{t-2} \circ f} & x_{j,t-1} & \xrightarrow{r_{t-1} \circ f} & x_{j,\ t} = EP_j \\
\vdots & & \vdots & & \ddots & & \vdots & & \vdots \\
SP_m = x_{m,0} & \xrightarrow{r_1 \circ f} & x_{m,1} & \xrightarrow{r_2 \circ f} & \dots & \xrightarrow{r_{t-2} \circ f} & x_{m,t-1} & \xrightarrow{r_{t-1} \circ f} & x_{m,t} = EP_m
\end{array}$$

Figure 2.8 – Pre-computation rainbow matrix computed from m starting points.

While this is arguably not a common activity when dealing with TMTOs, one should be careful when trying to extend chains from a rainbow table, that is pursuing the computation from the endpoints to get a table of size $t' > t$ (and possibly discarding the new-found merges in the range of columns $[t, t']$). This can happen when one wants to "compress" a table after the pre-computation phase, at the expense of a higher running time. In general, consecutive reduction function offsets are used across the different tables, with the ith table using $r_{it}, r_{it+1}, \dots, r_{i(t+1)-1}$. Then the end of a table and the beginning of the next one will contain the same reduction function, and can merge on this window, resulting in duplicate chains in different tables, hence reducing the coverage of the trade-off. An easy solution in that case is to anticipate such behaviour and offset the reduction function index by a large constant K such that the ith table begins with $r_{Kit}, r_{Kit+1}, \dots$

The different reduction functions also reduce the number of merges during the building of the matrix. Indeed, in order for a collision to happen, we must have

$$\exists i, x, x' \quad \text{s.t.} \quad x \neq x' \quad \text{and} \quad r_i(x) = r_i(x').$$

Therefore, this event can now only occur on $\frac{1}{t}$th of the previous possibilities with the Hellman trade-off, which implies a reduction of false alarms during the online phase.

Success rate The success rate of a single table is given in [Oec03] At a given column k, the probability of an element not to be in the column k of a matrix is $1 - \frac{m_k}{N}$. The success rate is then the inverse probability that an element is not in any of the t columns of TM.

Proposition 2.5 (Rainbow table success rate). *The success rate of a single rainbow table of size m and length of chains t is*

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

where $\forall k \in \{1, \ldots, t\}$ *the* m_k *are the number of distinct elements at column* k*.*

A perfect version of the rainbow trade-off can be considered. As with the DP variant, a perfect table consists of a table where the endpoints are distinct pairwise. All but one of the chains having the same endpoint are discarded. Indeed if two chains collide at any column, they will reach the same endpoint. We are therefore ensured that in a perfect table, all the elements at each column are distinct. Like in the DP variant, and contrarily to the Hellman case, it is very easy to discard duplicates in a rainbow matrix, since all the chains that could possibly merge are aligned. Nevertheless, this results in a significantly higher pre-computation cost: many chains are computed during the pre-computation without being included in the matrix.

Maximum tables In [AJO08] Avoine, Junod and Oechslin give an analysis of the upper bound of the success rate of a perfect rainbow table. The problem of maximum coverage is linked to the problem of the maximum number of distinct chains that can be built with the same family of reduction functions $(r_i)_{1 \leq i < t}$. To maximise the coverage, a perfect table, which gets rid of the duplicates in every column, is considered.

Theorem 2.6 (Maximum success rate)**.** *The expected maximum success rate of a single maximum perfect rainbow table is*

$$P_{max} = 1 - \left(1 - \frac{2}{t+2}\right)^t .$$

When $t \gg 1$*,* $P_{max} \approx 1 - e^{-2} \approx 86\%$*.*

Proof. The maximum table is obtained with $m_0^{\max} = N$ starting points. From Proposition 2.1 we know that the matrix will then have

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

distinct elements in its last column. Since the table is perfect, only m_t chains are kept, and if there are $m_t^{\max}$ distinct elements in the last column, it implies that there are as many elements in each column. We can therefore rewrite the success rate from Theorem 2.5:

$$P_{\max} = 1 - \left(1 - \frac{2N}{N(t+2)}\right)^t .$$

□

Matrix stopping rule From the maximum success rate, we can infer that the matrix stopping rule of the rainbow trade-off which provides the maximum success rate is $mt \approx 2N$. More generally, the matrix stopping rule used in the rainbow trade-off in the non-maximal setting is in the order of $mt \approx N$, which differs by a factor t from both the Hellman and the DP matrix stopping rules.

2.4.2 Online phase

During the online phase of the rainbow trade-off, the goal is to find the right table but also the right column. While we could extend the same chain until we find an endpoint in both the Hellman and the DP trade-offs, this is no longer possible in the rainbow scheme. Indeed, if we are given $f(x_{ij})$, for some x_{ij} in the TMTO matrix, and we suppose that x_{ij} appears only once there, only the function r_i will

give us the next element $x_{i\ j+1}$. Furthermore, only the correct sequence of $(r_k)_{j+1<k<t}$ will lead to the endpoint EP_i which we are looking for.

The endpoints are therefore searched from each column independently, with a different potential chain each time. That is, a column k is searched by computing a chain of length $t-k$, which leads to a potential endpoint. The elements at the rightmost end of the TMTO matrix provide faster results than the first elements, since they need shorter chains to reach the potential endpoint. The algorithm of the online phase maximises the number of potential endpoints tested over the amount of computation done to finish the chains at any given moment. Given an input $y_0 \in \mathcal{H}$, and a single table, it therefore browses the table in reverse order to find the preimage. The last column is tried first. It only requires a reduction of our input element y_0 using the function r_{t-1} to access a potential endpoint. Then, the table is searched to find if $r_{t-1}(y_0)$ is among the endpoints. When the search is unsuccessful, the next step is to build a new chain which is started by computing $r_{t-2}(y_0)$, applying f on it, and then r_{t-1} to obtain a second potential endpoint. When an endpoint is found, a verification chain is started from the corresponding starting point, as in the other methods. The algorithm finishes when either a verification succeeds, in which case the element before the last on the verification chain is the correct preimage, or all the columns have been searched, unsuccessfully.

When several tables are used, they are not accessed sequentially as in the Hellman scheme. Again, in order to try a maximum of endpoints as early as possible, the search is interleaved in the different tables such that the next table always contains the highest column yet untried. That is, if we denote $\langle k, n\rangle$ the search from the kth column of the nth table, the order of searching is as follows:

$$\langle t, 1\rangle, \langle t, 2\rangle, \ldots, \langle t, \ell\rangle, \langle t-1, 1\rangle, \ldots, \langle 1, 1\rangle, \ldots, \langle 1, \ell-1\rangle, \langle 1, \ell\rangle.$$

The algorithm stops when an answer is found. The average number of steps required for this to happen is quantified in Proposition 2.7.

Proposition 2.7. *The probability that the preimage is successfully found at step k of the rainbow trade-off is given by*

$$p_k = \prod_{j=1}^{k-1}\left(1 - \frac{m_{t-j}}{N}\right),$$

where, for $i \in \{1, \ldots, t\}$, m_i denotes the number of distinct elements in column i of the rainbow matrix.

In the perfect trade-off, this same probability follows a geometric law of parameter $\frac{m}{N}$, whose probability function is

$$p_k = \frac{m}{N}\left(1 - \frac{m}{N}\right)^{k-1}.$$

Proof. The online phase stops when the answer is found at step k if it is not found in any of the previous steps. For a given step k, this happens with probability $1 - \frac{m_k}{N}$, whose product on all the column gives the first claim.

In the case of the perfect trade-off, we have $m_1 = m_2 = \cdots = m_t = m$, so the probability is the same at each step. Furthermore, the elements are all distinct in a given column, which implies that the probabilities are independent. This corresponds to a geometric distribution. □

We now consider the amount of computations required by the verifications of the alarms, which is given in Proposition 2.8.

Proposition 2.8. *In a single rainbow table of size m, computed with chains of length t, the number of applications of the one way function F due to false alarms up to the kth step is given by*

$$Q_k = \sum_{i=t-k}^{t} i q_i,$$

where

$$q_i = 1 - \frac{m}{N} - \prod_{j=i}^{t} \left(1 - \frac{m_j}{N}\right).$$

In the case of the perfect rainbow trade-off, we have

$$q_i = 1 - \frac{m}{N} - \frac{i(i-1)}{t(t+1)}.$$

Proof. The probability of the currently considered chain having merged with a chain of the TMTO matrix (a false alarm) depends on how far from the start of the table the search is performed: a search at the first column will not trigger any false alarm during the verification, whereas a search in the last columns will have a verification chain with a nearly maximal length (close to t). A false alarm happens in all cases where the right chain is not reached, which would happen with probability $\frac{m}{N}$, and if none of the elements from the end of the verification chain until the endpoint collide with an element of another chain placed at the same column. We denote with m_i the distinct elements of the ith column of the rainbow matrix. If we consider the column $k > 0$ of a perfect table of length t, this last event would happen with probability

$$\text{col}_k = \Pr(\text{collision between } k \text{ and } t) = \prod_{i=k}^{t} \left(1 - \frac{m_i}{N}\right).$$

Note that if the table is perfect, $m_i = m$ is constant and this simplifies in

$$\text{col}_k = \frac{k(k-1)}{t(t+1)}.$$

The probability of a false alarm is therefore

$$q_k = 1 - \frac{m}{N} - \text{col}_k.$$

Each column additionally costs a verification due to a false alarm with a probability q_c, for $k < c < t$. Each false alarm costs c calls to f, c corresponding to the length of the verification chain. □

We can now give the expected number of applications of the one-way function during the online phase of the rainbow trade-off.

Theorem 2.9. *If we consider that the given y_0 is present in the TMTO matrix of the trade-off, the expected number of calls to the one-way function during the online phase of the rainbow trade-off is given by*

$$T_{\textit{rainbow}} = \sum_{k=1}^{\ell t} p_k \ell \left(\frac{(t-c_k)(t-c_k+1)}{2} + Q_{c_k} \right),$$

where $c_k = \left\lfloor \frac{k-1}{\ell} \right\rfloor$, p_k is the probability to stop at step k, and Q_k is given in Proposition 2.8.

Proof. Up to a column k, there are

$$\sum_{i=1}^{t-k} i = \frac{(t-k)(t-k+1)}{2} \tag{2.7}$$

calls to f in order to build the previous and the current endpoint chain, and, according to Proposition 2.8, Q_k additional computations due to false alarms.

The browsing of the tables is represented by $c_k = \left\lfloor \frac{k-1}{\ell} \right\rfloor$, such that the sequence $c_0, c_1, \dots, c_{\ell t}$ corresponds to the order described above.

Then, the total cost of the online phase with ℓ tables is the cumulative cost at each column given by the expression (2.7) and Q_k, multiplied by ℓ, and weighted by the probability to find the preimage. □

In [Hon10], Hong, in an independent analysis, proposes approximations of the results of Theorem 2.9. These heavily exploit the approximation $(1-x)^y \approx e^{-xy}$ when $x \ll 1$, and neglect anything but the t^2 term of the polynomial in t that they obtain as an approximation of T_{rainbow}. Their results, given in Theorem 2 and Theorem 3 of [Hon10], are collected in the following theorem:

Theorem 2.10. *We denote with* $\bar{c} = \frac{mt}{N}$ *the* matrix stopping constant*, capturing the relation of the matrix stopping rule. The expected number of calls to the one-way function during the online phase of the non-perfect rainbow trade-off is approximated by*

$$T_{\textit{rainbow}} \approx \left[\left(\frac{2\bar{c}}{3} + \frac{\bar{c}^2}{12} + 2 \right) + \left(\frac{\bar{c}^2 + 2\bar{c}}{3} + \frac{\bar{c}^3}{20} \right) \right] \left(\frac{t^2}{2+\bar{c}} \right)^2 .$$

In the perfect trade-off, a similar approximation is given by

$$T_{\textit{rainbow}} \approx \left(\frac{\bar{c} - 2\bar{c}^2}{e^{\bar{c}}} + 2\bar{c} \right) \left(\frac{t}{2\bar{c}} \right)^2 .$$

These approximations highlight the online running time's strong dependence on the matrix stopping rule.

2.5 Fuzzy rainbow trade-off

We now consider the fuzzy rainbow trade-off, which was introduced by Barkan, Biham and Shamir in [BBS06] and interestingly combines ideas from both the rainbow trade-off, and the DP trade-off in a new trade-off variant. The authors of [BBS06] argue in favor of the efficiency of this trade-off, compared to the other two variants. The technique also appears in [BP13] which, however, does not reference [BBS06].

We consider a trade-off on a problem space $\mathcal{N}$ of size N, where f is the one-way function. Let ℓ be the number of tables containing m chains each. Let us consider a rainbow chain of length s. It uses a different linking function $F_i = r_i \circ f$, where the r_i are distinct reduction functions, for $i \in \{0, \dots, s-1\}$. With each of these s reduction functions, we can construct DP chains with a DP property of probability $\frac{1}{t}$, so that these chains are of length t in average. A chain in the fuzzy trade-off consist in such a rainbow chain, where each link is replaced by a DP chain, hereafter called *sub-chain*, such that the chain consist in successive distinguished points. Once a distinguished point is reached, it acts as a starting point from

a second chain, continuing the first one, with a different reduction function. With the notations above, such a chain involves the following order of linking functions:

$$\underbrace{F_0 \to F_0 \to \cdots \to F_0}_{t_0} \to \underbrace{F_1 \to \cdots \to F_1}_{t_1} \to \cdots \to \underbrace{F_s \to \cdots \to F_s}_{t_s},$$

were the $t_1, \ldots, t_t$ are the lengths of the DP sub-chains, which average to t. A chain therefore consists in average of ts iterations of the one-way function f.

A fuzzy pre-computation matrix is computed with m chains. When s is large, the chains length are close to the average ts, and the corresponding TMTO matrix is roughly rectangular. Nevertheless, the s DP are not "aligned" in the TMTO matrix, in the sense that, for all $k < s$, the column[5] of the kth DPs of each row are not positioned at the same position in the matrix. The name "fuzzy" TMTO matrix, and thus the name of the trade-off, comes from this slight displacement of the DPs in the DP column. Such a matrix can be seen as the concatenation of DP matrices, where each column of endpoints are also the column of starting point of the next DP matrix.

Fuzzy trade-off tables are extracted from the TMTO matrix, by keeping (SP, EP) pairs, as in the rainbow trade-off. In fact, we can see that the fuzzy trade-off is exactly the rainbow trade-off, of table length s, when $t = 1$. Similarly, it can degenerate in the DP trade-off by setting $s = 1$.

Since both the DP trade-off and the rainbow trade-off offer perfect versions, a perfect version of the fuzzy trade-off can also be considered. Nevertheless, the choice of which chain to remove is not clear, as the combination of the two trade-off change the distribution of the elements. In particular, Kim and Hong remark in [KH14], that in the fuzzy trade-off, lengths of the DP chains are shorter than in the classical DP trade-off and tend to follow a normal distribution instead of a geometric one. This implies that keeping the longest fuzzy chain is not necessarily the best choice as longer chains increase false alarms. In [KH14], the authors clean each DP matrix in the fuzzy TMTO matrix before computing the next one.

An analysis of the characteristics of the non-perfect trade-off is provided by Kim and Hong in [KH13]. In particular, the success rate of the trade-off, and the average online time complexity, are computed.

We denote with $(\mathrm{DP}_i)_{i \le s}$ the sequence describing the number of distinct distinguished points for each DP matrix, *i.e.*, DP_i corresponds to the number of distinct elements between the DP_i and the DP_{i+1} column in the fuzzy TMTO matrix. The probability of success of the fuzzy trade-off is:

$$P = 1 - e^{\frac{\ell S}{N}}, \quad \text{where} \quad S = \sum_{i=1}^{s} \mathrm{DP}_i.$$

During the online phase, the generation of a chain is expected to require

$$t\ell \sum_{i=1}^{s} (s - i + 1) \left(\frac{2 + \frac{mt^2 i}{N}}{2 + \frac{mt^2 s}{N}} \right)^{\frac{2\ell}{t}} \tag{2.8}$$

applications of the one-way function f. The resolution of false alarms is expected to require an additional

$$t\ell \frac{mt^2}{N} \sum_{i=1}^{s} (si - i^2 + i + 1) \left(\frac{2 + \frac{mt^2 i}{N}}{2 + \frac{mt^2 s}{N}} \right)^{\frac{2\ell}{t}} \tag{2.9}$$

[5] For simplicity, in a similar way to the fact that variable chains length constitute DP "matrices", we will refer to the set of each kth DP as a *column*.

iterations of f. The sum of the two expressions 2.8 and 2.9 gives the average online time complexity:

$$T_{\text{fuzzy}} = t\ell \sum_{i=1}^{s} \left[(si - i^2 + i + 1) \left(\frac{mt^2 i}{N} + 1 \right) + \frac{mt^2}{N} \right] \left(\frac{2 + \frac{mt^2 i}{N}}{2 + \frac{mt^2 s}{N}} \right)^{\frac{2\ell}{t}} \tag{2.10}$$

The authors of [KH13] also provide a comparison with the non-perfect version of the rainbow trade-off. A first of their conclusion is that the rainbow trade-off is able to achieve higher success rates than the fuzzy trade-off. Nevertheless, when they take the pre-computation cost associated with each trade-off into account in order to perform the comparison, they state that the fuzzy rainbow trade-off is always more efficient given a similar amount of pre-computation, for the success rates that are reachable by both trade-offs.

2.6 Time-Memory-Data trade-offs

Time-Memory-Data trade-offs (TMDTO) are a technique of application of TMTOs, which can be used on any variant of the Hellman trade-off, or on the Hellman trade-off itself. They were found widely useful in the cryptanalysis of stream ciphers. Although they are not the focus of this thesis, they are the subject of a significant part of the literature on TMTOs and should be mentioned here [BS00; BD00; MFI07; CK08; DK08b; KCGL09; LLH15; BMS06; Li16].

The main idea of TMDTO is to exploit the fact that there exist cases where there is more than one input for the inversion problem, although only a success on only one of them is sufficient to solve said problem. This is notably the case with stream ciphers.

Stream ciphers are fundamentally finite states machines producing a key stream, which is then XORed with a stream of data to be encrypted. Cryptanalysis of stream cipher often involves a certain length of output from which several known plaintext-ciphertext pairs can be extracted, such as when used in protocols with repeating headers, or when an attacker comes across an encrypted stream of null bytes. These pairs act as the multiple inputs to the inversion problem which involves recovering the secret state of the cipher from a given output of the key stream bits.

The inversion problem then becomes a *multi targets* inversion problem. The TMDTO were found interesting in the fact that they do not take any specificity of a given construction into account, contrarily to many attacks on stream cipher in the literature.

In addition to the already introduced parameters, such as the online time T, the memory M and the problem size N, the TMDTO technique introduces a new *data* dimension D. This last parameter accounts for the number of inputs with which the TMDTO is designed to deal with, in the multi target inversion problem.

2.6.1 BG attack

A first general attack in the context of a stream cipher was introduced independently by Babbage in [Bab95] and Golic [Gol97], as what was later named the *BG attack*, after the names of the authors of both works. This attack is not really a trade-off in the sense that the space is exhausted and stored in memory. The observation had already been made by Hellman in [Hel80] that a one-way function which is a cyclic permutation could be attacked with a TMTO following the curve $TM = N$. The BG attack is a generalisation of this attack to the multiple inversion setting.

We denote the set of all possible states with STATE. The attack defines a function f by

$$f : \{0,1\}^n \to \{0,1\}^n,$$

which associates the internal state of the state machine of the cipher after initialisation, to the first n bits of output of the cipher, where $n = \lceil \log_2(|\text{STATE}|) \rceil$ quantifies the size of the state space. The BG attack consists in computing all the pairs $(x, f(x))$ for $x \in$ STATE into a dictionary occupying a memory M, such that it is easy to establish x given some $f(x)$. The attacker is given at least an amount of data of size n bits $d_1 d_2 \dots d_n$, plus an additional D bits $d_{n+1} \dots d_{n+D}$ of output of the key stream. He therefore can extract D possible windows of n bit sequences $d_i d_{i+1} \dots d_{i+n}$, for $0 < i \leq D$, which represent outputs that can be inverted using the dictionary in order to obtain a preimage.

D and M are chosen according to the birthday paradox in a way that a solution can be found with high probability, that is it satisfies the relation $DM = N$. The running time is then simply the data available $T = D$. Therefore, the resulting attack places itself as a data point on the curve $TM = N$. By discarding some of the data the running time can be decreased at the expense of more memory required for the dictionnary. This gives us an idea of the performance, but that is all that can be used as a comparison, since the attack lacks the fundamental ability of a trade-off, and the conditions of its application are also different from the Hellman trade-off which only requires a single plaintext-ciphertext pair.

2.6.2 TMDTO attack

The main idea of TMDTO comes from Biryukov and Shamir in [BS00], which presents the idea of combining TMTOs and the BG attack. The concept is to give the attack the ability to trade the memory against the running time, given an amount of data D determined beforehand, but without giving up on the advantage of possessing more data than with the classical Hellman scheme.

The TMTO is built in order to invert the same one-way function that was defined for the BG attack. The function f does indeed qualify for a TMTO, since the randomness of the output is among the requirements of a stream cipher. Note that in general, pseudo-random generators are similar enough to stream ciphers, the way that a function f can be defined for them as well. The problem space $\mathcal{N}$ considered is therefore the state space of the cipher SPACE, which is expected to be covered by the TMTO with high probability, while involving only a fraction of the memory used by the dictionary.

The key point of the TMDTO, and the reason they are interesting in cryptanalysis of stream ciphers, is the observation that, since there are D inputs whose inversion is to be performed with the same set of tables, the full coverage of $\mathcal{N}$ is not necessary, and one can expect to still find at least one preimage among the D candidates, by using a fraction $\frac{N}{D}$ of $\mathcal{N}$. The relation of parameters of the Hellman trade-off with the parameters of the curve, alongside the matrix stopping rule

$$T = t^2, \quad M = mt, \quad \text{and} \quad mt^2 = N,$$

are modified with the data component in

$$T = t^2, \quad M = \frac{mt}{D}, \quad \text{and} \quad mt^2 = \frac{N}{D},$$

which gives a new curve for the TMDTO trade-off:

$$TMD^2 = N^2.$$

The DP trade-off variant is considered in [BS00] in the context of a TMDTO. It is mentioned as sampling technique in the article and is applied instead of the Hellman TMTO. The incentive is mainly to reduce the disk lookup performed during the online phase, as the performance analysis in [BS00] shows that the variant presents the exact same TMDTO curve than the curve in the above TMDTO.

2.7 Applications

After the presentation of the different TMTO variants, we now detail some of the practical contexts in which TMTOs were found interesting. A large range of functions can be inverted by a TMTO. We give below examples of such one-way problems, commonly used in cryptography, and which can benefit from the use of a TMTO in order to be solved.

2.7.1 Password cracking

As stated in Chapter 1, password cracking is a major focus of TMTOs. We go more in detail hereafter on the technical reasons making TMTOs such efficient tools in this field. We first introduce collision resistant hash functions, which are often used to process passwords in authentication mechanisms.

Definition 2.10 (Collision resistant hash function). *Let $\mathcal{X}$ be a set of elements and $l \in \mathbb{N}^*$. A function $h : \mathcal{X} \to \{0,1\}^l$ is a* collision resistant hash function *if any probabilistic polynomial algorithm which searches for $(x, x') \in \mathcal{X}^2$, such that $h(x) = h(x')$, succeeds with negligible probability.*

A class of hash functions that are collision resistant are cryptographic hash functions, which require, in addition of the collision resistance property, what is called pre-image resistance, which is basically the one-way property which was introduced at the beginning of this chapter. Cryptographic hash functions also require a second preimage resistance, which consists in making it difficult to find a collision of the hash function with a given element of the domain, *i.e.*, given $y \in \mathcal{X}$, find $y' \in \mathcal{X}$, such that $y \neq y'$ and $h(y) = h(y')$. Due to the birthday paradox, cryptographic hash function are designed with an image size l sufficiently large, so that $\sqrt{l}$ is difficult to enumerate.

With cryptographic functions, we have $N \ll \sqrt{|\Omega|}$. We consider some $y_0 = h(x_0)$ with $x_0 \in \mathcal{N}$. If we consider a second preimage $x'_0 \in \Omega$, such that $h(x'_0) = h(x_0)$, there is a negligible probability that $x'_0 \in \mathcal{N}$. In [HM13], the authors give a more precise estimation of such probability, while arriving at the same conclusion. Then, given a digest $y_0 = h(x_0)$, with $x_0 \in \mathcal{N}$ given by such a hash function, finding a preimage of y_0 becomes equivalent to the problem of finding x_0. Indeed, the probability to find a preimage, which is not x_0 implies a search in Ω rather than $\mathcal{N}$ which is impossible in practice.

Passwords are often stored as hashes in databases, to prevent them from being known by database operators, which would happen if they were stored as plaintext. Password hashes are commonly obtained by the application of a cryptographic hash function such as MD5, or a function of the SHA family (SHA1, SHA256 or SHA512). This way, any authenticating system can use $h(p)$, where p is the plaintext password, instead of p itself.

Then, a TMTO targeting passwords hashes can be designed, so that given a password hash, it can recover the plaintext password. In general, if the entropy of the input space $\mathcal{N}$ is low enough, the TMTO technique can be applied on any hash function. When computing the TMTO, the input space $\mathcal{N}$ corresponds to a password space. The attacker constructs $\mathcal{N}$ with a subset of passwords to be tested quickly by the TMTO, whose size is reasonable compared to the set of all possible passwords. This often makes sense as, since the human memory is limited, passwords tend to be made easy to memorise. That

is, people often choose words, or short gibberish, as passwords. Sets of every password consisting of only lowercase letters, or every password containing both lowercase letters and digits, of short length (< 10 characters) can make sense for a TMTO which aims at spotting weak passwords. Note that in the case of passwords, the inverse that need to be found has to be the correct password. Indeed, a second preimage of a digest is likely to contain non printable characters, which are not suitable in a password.

Salting There is a counter-measure against the TMTO on hashed passwords known as *salting*. The idea is to add random data, here denoted as SALT, to the password before hashing, and to keep this random data in plaintext next to the password hash in the database,

$$h(\mathrm{SALT}|p).$$

During the authentication, the service can easily append the given random bytes to the supplied password. The inversion task, however, is hardened by the fact that the TMTO has to be built to find a preimage of the format SALT$|p$, whose size is longer than p alone. In fact, the problem size becomes arbitrarily higher, at no cost for the user.

Unfortunately, there are still unsalted password databases in production nowadays, as can be seen in the occasional public breaches [Pag18; Whi18]:

> “The database also contained passwords, which were stored as an MD5 hash, a long-outdated algorithm that is nowadays easy to crack.”

A widely known non salted password hashing mechanism is still used in Microsoft Windows’ authentication system NTLM (NT LAN Manager), for compatibility with older versions of the operating system. Windows’ logon mechanism includes a challenge involving a *NT Hash*, which is how the password is stored in the SAM database, containing all the credentials managed by the operating system. This NT Hash is basically at its core an application of the hash function MD4 on the password. Beside the fact that this scheme allows *Pass the hash* types of attacks, where an attacker eavesdrops a hash from a user and replays it to the server to gain unauthorized access, Windows password databases are a good target for TMTOs.

2.7.2 Cipher cryptanalysis

Known-plaintext attacks on a given cipher correspond to the case when, given a known message P, and its encryption $C_K(P)$, we want to determine the key K used for this encryption. The TMTO is therefore dependent on P_0, and we use

$$f : K \to C_K(P)$$

as the one-way function. The set $\mathcal{N}$ is the key space we want to test, which may be part of or all of the key space of the cipher. Such attacks were the initial problem tackled by TMTOs with the block cipher DES, such as done in [Hel80] which first introduced them.

Note that, in general, TMTOs on ciphers only work on reduced key space such as artificial limitations put on obsolete ciphers for legal reasons. For modern cryptosystems with key spaces of large size ($\geq 2^{128}$), one would have to possess knowledge of a significant part of the key, or the key generation would have to be flawed in a way to reduces the key space, for the TMTOs to be practical.

2.7.3 Pseudo-random number generators

Pseudo-random number generators (PRNG) are deterministic functions producing a sequence of integers, based on an initial *seed* value. PRNGs are sometime used in sensitive contexts, for example when generating cryptographic keys. In such a context, knowing the seed is often equivalent to knowing the secret key. Given a sequence of successive outputs of a PRNG, it is usually difficult to recover the seed. Indeed, it is common for PRNGs to have truncated versions of their internal state as output, with said internal state large enough to make the enumeration of all possible values impractical. One can think, for instance, about the Mersenne Twister PRNG presented in [MN98], which possesses an internal state of size $2^{32^{623}}$.

A naive idea of TMTO which would be able to recover the seed, given a sufficient amount of successive outputs, is presented below. Let n be a number of outputs such that, their length in bit corresponds to the size of the internal state of the PRNG. We define a one-way function f by

$$f : x \mapsto (out_0, out_1, \ldots, out_n),$$

where out_k is the kth iteration of the PRNG initialised with the seed x. Note that, with a seed space small enough, we can modify f to include an offset k of the first output,

$$f : k|x \mapsto (out_{k+0}, out_{k+1}, \ldots, out_{k+n}),$$

where $|$ denotes the concatenation operation. This way, with outputs that are iterated in a range $\{0, \ldots, k\}$ from the initial value output by the PRNG, one can quickly recover the seed given a sufficient number $\geq n$ of outputs. A more robust TMTO can be conceived by considering PRNG in the same way as the stream ciphers, for which TMDTO techniques exists.

2.8 Summary

In this chapter, TMTOs have been introduced formally. The inversion problem which consists in inverting a function on a specific problem space has been described, using the notion of random one-way function. The trade-off nature of the TMTOs has been discussed, with a focus on the different dimensions that are involved in the inversion problem, including the time, the memory, and the problem size. The versatility of the technique, tunable between exhaustive search and lookup has been highlighted. The description of these relations serve as an introduction to the problem of evaluating the performance of a trade-off, which is a focus of this thesis.

The two phases of the TMTO have been presented: the offline and the online phase. The offline phase, performed once, constructs the tables. The cost of this pre-computation is very high, often higher than exhaustive search. The tables are afterwards exploited in the online phase by the inversion algorithm in order to solve an specific inversion problem. The inversion is done faster and with less memory then with the naive methods. The concept of chain, using reduction functions, has been introduced. The notion of TMTO matrix has been built upon it. Then, we saw that the table construction, which only keep a part of the TMTO matrix allows the reduction of the memory requirement, at the expense of more computation time during the online phase. These core notions, and the introduced vocabulary are reused extensively throughout this manuscript.

The presentation of the original trade-off by Hellman has been detailed, introducing the online phase algorithm key points, alongside the false alarm considerations. Three of the main variants, which

improve on the Hellman trade-off, have been considered: the DP trade-off, the rainbow trade-off, and the fuzzy trade-off. The average online running times that have been given in these sections will be built upon in Chapter 3 to present the state of the art in terms of performance in TMTOs.

Finally, some context of applications of TMTO has been discussed, to put a light on the popularity of the technique in practice, in particular with regard to password cracking.

References

[AC17] Gildas Avoine and Xavier Carpent. *Heterogeneous Rainbow Table Widths Provide Faster Cryptanalyses*. In: *ASIACCS 17*. Ed. by Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi. Abu Dhabi, United Arab Emirates: ACM Press, Apr. 2017, pp. 815–822 (cit. on pp. 20, 81–83).

[ACL15] Gildas Avoine, Xavier Carpent, and Cédric Lauradoux. *Interleaving Cryptanalytic Time-Memory Trade-Offs on Non-uniform Distributions*. In: *ESORICS 2015 , Part I*. Vol. 9326. LNCS. Vienna, Austria: Springer, Heidelberg, Germany, Sept. 2015, pp. 165–184 (cit. on pp. 20, 78–82).

[AJO08] Gildas Avoine, Pascal Junod, and Philippe Oechslin. "Characterization and Improvement of Time-Memory Trade-Off Based on Perfect Tables". In: *ACM Transactions on Information and System Security* 11.4 (4 July 2008), 17:1–17:22. ISSN: 1094-9224 (cit. on pp. 16, 27, 30, 37, 56, 64, 72–74, 77, 154, 156, 157, 160).

[Bab95] Steve Babbage. *A space/time tradeoff in exhaustive search attacks on stream ciphers*. In: *European Convention on Security and Detection – ECOS'95*. Vol. 408. Conference publication (Institution of Electrical Engineers). Institution of Electrical Engineers. Brighton, England: Inspec/IEE, May 1995 (cit. on p. 42).

[BBS06] Elad Barkan, Eli Biham, and Adi Shamir. *Rigorous Bounds on Cryptanalytic Time/Memory Tradeoffs*. In: *CRYPTO 2006*. Ed. by Cynthia Dwork. Vol. 4117. LNCS. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 2006, pp. 1–21 (cit. on pp. 35, 40, 58, 153, 155–157).

[BD00] Eli Biham and Orr Dunkelman. *Cryptanalysis of the A5/1 GSM Stream Cipher*. In: *Progress in Cryptology - INDOCRYPT 2000*. Ed. by Bimal K. Roy and Eiji Okamoto. Vol. 1977. LNCS. Calcutta, India: Springer, Heidelberg, Germany, Dec. 2000, pp. 43–51 (cit. on p. 42).

[BMS06] Alex Biryukov, Sourav Mukhopadhyay, and Palash Sarkar. *Improved Time-Memory Trade-Offs with Multiple Data*. In: *SAC 2005*. Ed. by Bart Preneel and Stafford Tavares. Vol. 3897. LNCS. Kingston, Ontario, Canada: Springer, Heidelberg, Germany, Aug. 2006, pp. 110–127 (cit. on p. 42).

[BP13] Fabian van den Broek and Erik Poll. *A Comparison of Time-Memory Trade-Off Attacks on Stream Ciphers*. In: *AFRICACRYPT 13*. Ed. by Amr Youssef, Abderrahmane Nitaj, and Aboul Ella Hassanien. Vol. 7918. LNCS. Cairo, Egypt: Springer, Heidelberg, Germany, June 2013, pp. 406–423 (cit. on p. 40).

[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*. Ed. by Peter de With and Mihaela van der Schaar-Mitrea. Veldhoven, The Netherlands, May 1998, pp. 111–118 (cit. on pp. 30–32).

[BS00] Alex Biryukov and Adi Shamir. *Cryptanalytic Time/Memory/Data Tradeoffs for Stream Ciphers*. In: *ASIACRYPT 2000*. Ed. by Tatsuaki Okamoto. Vol. 1976. LNCS. Kyoto, Japan: Springer, Heidelberg, Germany, Dec. 2000, pp. 1–13 (cit. on pp. 22, 42–44).

[CK08] Guanhan Chew and Khoongming Khoo. *A General Framework for Guess-and-Determine and Time-Memory-Data Trade-Off Attacks on Stream Ciphers*. In: *SECRYPT*. INSTICC Press, 2008, pp. 300–305 (cit. on p. 42).

[Den82] Dorothy E. Denning. *Cryptography and Data Security*. Boston, Massachusetts, USA: Addison-Wesley, June 1982, p. 100 (cit. on p. 30).

[DK08b] Orr Dunkelman and Nathan Keller. "Treatment of the initial value in Time-Memory-Data Tradeoff attacks on stream ciphers". In: *Information Processing Letters* 107.5 (Aug. 2008). Ed. by Andrzej Tarlecki, pp. 133–137 (cit. on p. 42).

[FN91] Amos Fiat and Moni Naor. *Rigorous Time/Space Tradeoffs for Inverting Functions*. In: *23rd ACM STOC*. New Orleans, LA, USA: ACM Press, May 1991, pp. 534–541 (cit. on p. 15).

[Gol97] Jovan Dj. Golic. *Cryptanalysis of Alleged A5 Stream Cipher*. In: ed. by Walter Fumy. Vol. 1233. LNCS. Konstanz, Germany: Springer, Heidelberg, Germany, May 1997, pp. 239–255 (cit. on p. 42).

[Hel80] Martin Hellman. "A Cryptanalytic Time-Memory Trade Off". In: *IEEE Transactions on Information Theory* IT-26.4 (July 1980), pp. 401–406 (cit. on pp. 24, 25, 29, 30, 42, 45, 55, 153, 154, 184).

[HJK+08] Jin Hong, Kyung Jeong, Eun Kwon, In-Sok Lee, and Daegun Ma. *Variants of the Distinguished Point Method for Cryptanalytic Time Memory Trade-Offs*. In: *Information Security Practice and Experience*. Ed. by Liqun Chen, Yi Mu, and Willy Susilo. Vol. 4991. Lecture Notes in Computer Science. Sydney, Australia: Springer, Apr. 2008, pp. 131–145 (cit. on p. 30).

[HKR83] Martin E. Hellman, Ehud D. Karnin, and Justin Reyneri. "On the Necessity of Cryptanalytic Exhaustive Search". In: *SIGACT News* 15.1 (Jan. 1983), pp. 40–44. ISSN: 0163-5700 (cit. on p. 18).

[HLM11] Jin Hong, Ga Won Lee, and Daegun Ma. *Analysis of the Parallel Distinguished Point Tradeoff*. In: *Progress in Cryptology - INDOCRYPT 2011*. Ed. by Daniel J. Bernstein and Sanjit Chatterjee. Vol. 7107. LNCS. Chennai, India: Springer, Heidelberg, Germany, Dec. 2011, pp. 161–180 (cit. on p. 30).

[HM13] Jin Hong and Sunghwan Moon. "A Comparison of Cryptanalytic Tradeoff Algorithms". In: *Journal of Cryptology* 26.4 (Oct. 2013), pp. 559–637 (cit. on pp. 16, 20, 30, 31, 33, 34, 44, 63, 157, 158, 160).

[Hon10] Jin Hong. "The cost of false alarms in Hellman and rainbow tradeoffs". In: *Designs, Codes and Cryptography* 57.3 (Dec. 2010), pp. 293–327 (cit. on pp. 27, 29, 40, 63–65, 67, 69–72, 166).

[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". In: *IEICE Transactions* 92-A.1 (2009), pp. 11–21 (cit. on p. 42).

[KH13] Byoung-Il Kim and Jin Hong. *Analysis of the Non-perfect Table Fuzzy Rainbow Tradeoff*. In: *ACISP 13*. Ed. by Colin Boyd and Leonie Simpson. Vol. 7959. LNCS. Brisbane, Australia: Springer, Heidelberg, Germany, July 2013, pp. 347–362 (cit. on pp. 41, 42).

[KH14] Byoung-Il Kim and Jin Hong. “Analysis of the Perfect Table Fuzzy Rainbow Tradeoff”. In: *Journal of Applied Mathematics* 2014 (2014), 765394:1–765394:19 (cit. on p. 41).

[KM96] Koji Kusuda and Tsutomu Matsumoto. “Optimization of Time-Memory Trade-Off Cryptanalysis and Its Application to DES, FEAL-32, and Skipjack”. In: *IEICE Transactions* E79-A.1 (Jan. 1996), pp. 35–48 (cit. on p. 26).

[LH16] Ga Won Lee and Jin Hong. “Comparison of Perfect Table Cryptanalytic Tradeoff Algorithms”. In: *Designs, Codes and Cryptography* 80.3 (Sept. 2016), pp. 473–523 (cit. on pp. 30, 158, 160, 162).

[Li16] Zhen Li. *Optimization of Rainbow Tables for Practically Cracking GSM A5/1 Based on Validated Success Rate Modeling*. In: *CT-RSA 2016*. Ed. by Kazue Sako. Vol. 9610. LNCS. San Francisco, CA, USA: Springer, Heidelberg, Germany, Feb. 2016, pp. 359–377 (cit. on p. 42).

[LLH15] Jiqiang Lu, Zhen Li, and Matt Henricksen. *Time-Memory Trade-Off Attack on the GSM A5/1 Stream Cipher Using Commodity GPGPU - (Extended Abstract)*. In: *ACNS 15*. Ed. by Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis. Vol. 9092. LNCS. New York, NY, USA: Springer, Heidelberg, Germany, June 2015, pp. 350–369 (cit. on pp. 5, 42, 84).

[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”. In: *IEEE Communications Letters* 11.12 (2007), pp. 988–990 (cit. on p. 42).

[MH08] Daegun Ma and Jin Hong. “Success probability of the Hellman trade-off”. In: *Information Processing Letters* 109.7 (Dec. 2008), pp. 347–351 (cit. on p. 26).

[MN98] Makoto Matsumoto and Takuji Nishimura. “Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator”. In: *ACM Trans. Model. Comput. Simul.* 8.1 (1998), pp. 3–30 (cit. on p. 46).

[Oec03] Philippe Oechslin. *Making a Faster Cryptanalytic Time-Memory Trade-Off*. In: *CRYPTO 2003*. Ed. by Dan Boneh. Vol. 2729. LNCS. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 2003, pp. 617–630 (cit. on pp. 5, 35, 36, 56, 107, 153, 156, 157).

[Oec04] Philippe Oechslin. *OPHCRACK (the time-memory-trade-off-cracker)*. Accessed: Jun 2014. 2004. Link. (Cit. on p. 35).

[Pag18] Pierluigi Paganini. *Chat app Knuddels fined €20k under GDPR regulation*. Accessed: Feb 2019. 2018. Link. (Cit. on p. 45).

[SRQL03] François-Xavier Standaert, Gaël Rouvroy, Jean-Jacques Quisquater, and Jean-Didier Legat. *A Time-Memory Tradeoff Using Distinguished Points: New Analysis & FPGA Results*. In: *CHES 2002*. Ed. by Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar. Vol. 2523. LNCS. Redwood Shores, CA, USA: Springer, Heidelberg, Germany, Aug. 2003, pp. 593–609 (cit. on pp. 30–32).

[TO05] Cedric Tissières and Philippe Oechslin. *Ophcrack*. Accessed: Apr 2017. 2005. Link. (Cit. on pp. 7, 35, 56, 93, 99, 162).

[Whi18] Zack Whittaker. *At Blind, a security lapse revealed private complaints from Silicon Valley employees*. Accessed: Feb 2019. 2018. Link. (Cit. on p. 45).

No matter how hard you try, you can't make a racehorse out of a pig. You can, however, make a faster pig.

A comment in the Emacs source code

Improvements on TMTOs 3

THE LITERATURE is prolific in adjustments for the TMTO technique. This chapter explores the different areas in which there is room for improvements. We review a collection of the most interesting works from the literature for these different areas, and recall some results that will be used in the subsequent chapters.

Contents

3.1 Improvements on TMTOs . . . **52**

3.2 Storage improvements . . . **53**

3.2.1 Table storage . . . 54

3.2.2 Indexing endpoints . . . 56

3.2.3 Compressed delta encoding . . . 58

3.2.4 Truncation . . . 62

3.3 Checkpoints . . . **63**

3.3.1 Generating checkpoints . . . 63

3.3.2 Online phase . . . 64

3.3.3 Performance of the checkpoint technique . . . 71

3.4 Fingerprints . . . **73**

3.4.1 Offline phase . . . 74

3.4.2 Online phase . . . 74

3.4.3 Optimal configuration . . . 77

3.5 Dealing with non-uniform problem spaces . . . **77**

3.5.1 Custom reduction function . . . 78

3.5.2 Interleaving . . . 78

3.6 Heterogeneous tables . . . **81**

3.6.1 Analysis . . . 82

3.6.2 Optimal tables parameters . . . 83

3.6.3 Performance . . . 83

3.7 Rainbow tables on GPU . . . **84**

3.7.1 GPU paradigm . . . 84

3.7.2 Rainbow method on GPU . . . 84

3.8 Conclusion . . . **85**

3.1 Improvements on TMTOs

We consider that a technique is an improvement if it increases the performance of the online phase, without incurring a significant overhead on the pre-computation phase. We present in the following sections different works from the literature that introduce such improvements, each tackling a different aspect of the TMTO technique to improve upon.

Let us specify the difference between what is called improvements in this thesis and the variants which were presented in Sections 2.2 to 2.5 of Chapter 2. Improvements can be thought of as an additional step after the pre-computation, *i.e.*, the computation of the TMTO matrix, which do not involve chains computation. For example, we cannot, from a DP matrix, obtain a rainbow or a Hellman matrix without redoing much of the computation, whereas improvements are more like alternative ways to process the matrix in order to obtain a table. Note that while it is convenient to think of improvements as a post process, it may not be optimal to only apply them after pre-computation. More precisely, one may benefit from keep in mind the improvements various beforehand when determining the TMTO parameters to, *e.g.*, compensate said gain with an increase of the corresponding parameter.

We aim at covering as much of the relevant literature as possible, but we focus on works which provide theoretical analysis. For various reasons, improvements proposed in literature are often directed towards one specific variant. The results are presented in the scope of this variant but most techniques can apply to the others. In particular, the rainbow trade-off is a frequent choice in the literature when it comes to new developments on improvements in TMTOs.

None of the presented works allow the TMTO to provide a better asymptotic performance. It does not come as a surprise, since even variants which involve strong modifications of the technique do not provide such a significant performance gain. Nevertheless, it should be noted that the evaluation of the performance of the trade-off considering only the asymptotic curves, which are in fact the same up to a factor among the different variants, is not sufficient. Reduction of the factor hidden in the asymptotic curve often yields a significant gain either in running time or by saving memory, which are noticeable in practice. This is the reason why all these improvements are considered in the first place.

Improvements on TMTOs translate in a reduction of the time or the memory of the online phase, or a reduction in either of these dimensions, at the cost of a negligible increase in the other, such that the online phase is more efficient overall. Note that in the event where a TMTO already offers sufficiently good online time with tables fitting in memory, such improvements can be used to construct an equivalent TMTO fitting in the same memory, providing the same online time, but covering a larger problem space. In most problems tackled by TMTOs, password cracking coming first in mind, one generally wants the problem space as large as possible given available resources. Thus, improvements that allow either of them to be reduced incurring only a negligible penalty with regard to the success probability of the TMTO are sought after.

The next sections present the different areas of TMTOs that were targeted for improvements in the literature, in such a way that the related work corresponding to a same area are presented together. The first consideration is the improvement of storage, presented in Section 3.2. It is an obvious starter and quickly yields interesting results, by exploiting techniques from the literature related to information theory, and adapting them to TMTOs. The next focus is on the amount of work which is induced by false alarms during the online phase, when verification chains are computed. The work related to false alarms can become a significant part of the total amount of computations of the one-way functions. The *checkpoints*, and the *fingerprint* techniques, presented in Sections 3.3 and 3.4, respectively, were found to reduce this work, at the expense of a relatively small memory overhead. The next two sections, 3.5

and 3.6, consider the tweaking of the online phase algorithm to work with non-uniform problem spaces with the *interleaving* technique, or with *heterogeneous tables* of different parameters (m, t) within a same TMTO. These two techniques can improve the average online time if we are ready to accept a worst time in case the answer is not found in the table. Finally, Section 3.7 presents an overview of the challenges of exploiting the availability of efficient hardware such as graphical processing units (GPU) in order to perform TMTOs. Indeed, due to their good performances in highly parallel workloads, GPUs are a continuing interest of the literature.

3.2 Storage improvements

TMTOs on problem spaces such as the ones used in practice require table computations from a particularly large number of chains, because these problem spaces are rather large themselves. When dealing with TMTOs implementations, one has to be careful not to exceed available memory by choosing adequate parameters. Since the early stages of TMTOs implementations, techniques were used to reduce the table size in memory which was scarce at the time. Even hard drive memory, which is cheap nowadays, was quite expensive. Many of these improvements were implemented without documentation of the process or the methodology that led to the chosen method.

We reuse the notations introduced in Chapter 2 to denote the parameters of the TMTOs. The one-way function is denoted with f. The problem space of size N is denoted with $\mathcal{N}$, the ℓ tables consist in m rows, extracted from chains of length t. The average online phase running time is denoted with T. In this section however, we give a more precise meaning of the memory M, as it now represents the amount in bits required to store the online phase procedure $\mathcal{A}^f_{\mathcal{N}}$.

Recall that $\mathcal{A}^f_{\mathcal{N}}$ consists in both an algorithm in the classic sense of the term, *i.e.*, a sequence of instructions on the one hand, and the information that was kept from the TMTO matrix on the other hand. Until now, we referred to this second part as "tables", and we shall keep the representation we gave of it in Chapter 2 when reasoning about both the offline and the online phase, but note that they can take various forms in their implementations, and therefore the array representation such as given in, *e.g.*, Figures 2.6 and 2.7, does not reflect the storage reality beyond the naive cases described below. As the description of the algorithmic part is clearly negligible in size compared to the TMTO matrix information in $\mathcal{A}^f_{\mathcal{N}}$, we will discard it in the computation of M, with no loss of accuracy.

Let us also clarify that what we mean by the memory M in the trade-off is actually the table, as it is stored in an offline fashion. Indeed, during the online time, $\mathcal{A}^f_{\mathcal{N}}$ may perform runtime operations on the stored information, in the way that it requires additional memory. For performance reasons, this additional memory complexity of the algorithm is generally kept negligible before its storage. It is therefore not accounted in M.

Given a fixed amount of memory, sparing any of it to store a table translates in the possibility to store more chains in a given table, for the same amount of memory. Recall that the trade-off in any of the main variants broadly follows the curve

$$T \approx \frac{N^2}{M^2},$$

which means that if we can fit, after improvements, the table (with the same m) in $M' < M$, and we still have the same online time, then we now have some unused space. So we can in fact consider tables that are $\frac{M}{M'}$ larger, and they will fit in the same amount as memory as the original tables, after improvements.

If we consider the online time T' of such tables, we have

$$T' = \frac{N^2}{(\frac{M}{M'}M)^2} = \left(\frac{M'}{M}\right)^2 T. \tag{3.1}$$

One can see that the reduction factor in Equation (3.1) is quadratic so there can be a significant time reduction as soon as the saving increases, *e.g.*, even a 5% memory improvement grants a 10% time reduction.

Among the several improvements that were applied to TMTOs, we distinguish between two types of techniques aiming at reducing the size of the endpoints in memory:

- *encoding and compression techniques*, which reduce the size without loss of information, and
- the *truncation technique*, where information is lost and must be somehow recovered during the online phase.

For the most common reduction techniques that can be found in the literature, some were applied and analysed in the specific context of TMTOs. These studies are gathered in this section where we will detail their principles. We begin by discussing some general considerations about the storage of the tables.

Note that since tables are generated with random functions, the storage values in bits given in this section are expected values. One could wonder if a particular random linking function $F = r \circ f$, where r is a reduction function, could allow the generation of tables in a way that the distribution of its elements can be stored in much less than the expected value. Since we always assume $m \gg 1$, the distribution of all storage values averaged over all the linking functions $F : \mathcal{N} \to \mathcal{N}$ will be grouped closely around the expected value, and an outlier situation of an especially well-structured table happens with negligible probability.

3.2.1 Table storage

In Chapter 2, we stated that if we consider a TMTO targeting a one-way function $f : \mathcal{N} \to \mathcal{H}$, the chains and the resulting matrix contain elements of $\mathcal{N}$. This is not strictly true as it is very much possible, with minor modification of the offline and online phase algorithms, to rely on elements of $\mathcal{H}$ instead. However, this is not a good choice in general for memory efficiency reason. Indeed, recall that for most one way functions we have $N \ll |\mathcal{H}|$, so the storage requirement of elements of $\mathcal{N}$ is much less than the one of elements of $\mathcal{H}$. Hence, we chose to always consider elements of $\mathcal{N}$ for the TMTO matrix.

The most basic storage of a table can be done as an array of rows which allows a storage equal to exactly m times the space required for a single row. More complex data structures, such as hash tables, would require additional space per row, and are usually not considered when the sorting of endpoints allows a fast enough search.

Non-optimised implementations tend to store points in integer types which are multiples of machine words, such as 32bits or 64bits integers. While the alignment in memory on a word boundary may give them a slight speed-up in certain cases, it is certainly negligible with regard to the loss of performance induced by the unused space. These implementations are therefore not considered, neither are the variants where the row itself (containing the endpoint and its corresponding starting points, both concatenated) is word-aligned.

We say the storage is made in a *bitpacked* manner, if each row consists in the bitwise concatenation of the starting point and the endpoint. The bit packing gets rid of the constraints regarding word alignment

in physical memory. It still requires both elements in the row to each be of fixed size, possibly padded by zero bits on their most significant side. Since all the elements of the TMTO matrix are in $\mathcal{N}$, they correspond to an integer strictly below N. The starting point and the endpoint can therefore each be encoded on at most $\lceil \log_2 N \rceil$ bits. This is what we will refer to as the *naive approach*. Therefore, the amount of bits required to store a single table using a naive approach is given by

$$M_{\text{naive}} = 2m\lceil \log_2 N \rceil.$$

There is a strong difference of representation in the literature between the improvements that deal with starting points storage and those that focus on the endpoints storage. Indeed, almost all improvements concern the endpoints, and very few works have been done regarding the starting points. This may have been induced by the fact that tables are usually sorted according to the endpoints, following the advice of Hellman in [Hel80], in order to ease the search needed to establish if a potential endpoint is in a given table, during the online phase. If the tables are sorted by the endpoints then the set of endpoints is structured, whereas the set of starting points stays seemingly uniformly distributed.

Starting points storage In the original presentation of TMTOs [Hel80], Hellman recommends to choose the m_0 starting points necessary to build a table at random in $\mathcal{N}$. An alternative method regarding the choice of starting points is to use a counter starting from 0. The usable information in the TMTO matrix, *i.e.*, the elements that can be inverted and that count in the coverage of $\mathcal{N}$, do not include the starting points, so their choice does not directly influence the success rate of the trade-off. Moreover, $F(\{SP_i\}_{i<m_0})$ is uniformly distributed, so in the case we use a counter, we actually get the same behaviour as with a random choice of starting points. However, this allows the starting points to be encoded in $\log_2 m_0$ bits, which is an improvement when $m_0 < N$. In fact, when tables that are not of maximal size are built, we have $m_0 < N$. In practice, we never make non-perfect tables of maximal size, since we would otherwise have $m_0 = m = N$, and we would get tables that are larger than dictionaries. Even in the maximal case, the counter technique does not perform worse than the random one, so this method should be preferred in most cases.

Due to their very different roles in the trade-off, starting points and endpoints are to be considered separately in their storage. In particular, their relation cannot be predicted since F is random, so their respective encodings are totally independent from each other. In the following, we will therefore separate the part of the table corresponding to the starting point, and the one relating to the endpoints, noting the one we refer to by using either SP or EP in superscript. For example, with the above counter method to store the starting points, the storage of the starting point part of a table is given, in bits, by

$$M^{SP}_{\text{counter}} = m \log_2 m_0.$$

In the non-perfect case, we have $m = m_0$.

Special case of DPs Note that in the case of the distinguished point trade-off, the structure imposed on the DP which is known and used by the online algorithm can be exploited to save space. With a distinguishing property of probability $\frac{1}{2^k}$, where the DP is of the form

$$b_1 b_2 \dots b_{n-k-1} b_{n-k} \underbrace{00\dots0}_{k},$$

it can be stored on $n-k$ bits at no cost and with no penalty during the online phase. This is an advantage in favour of the DP trade-off, as it can benefit in addition from the techniques presented below.

Lower bound Establishing a lower bound of the expected storage space may be useful in giving insight about the potential compression from which the tables could benefit. In general, the more structure can be established from the data, the more compressible the data is. The choices made in the construction of the table are therefore important, as they influence the compressibility.

In the case of the perfect rainbow trade-off, a lower bound is established by Avoine and Carpent in [AC14]. They only rely on the fact that F is random. Then, endpoints are expected to be uniformly distributed in $\mathcal{N}$. Since the table is perfect, the endpoints are also guaranteed to be distinct from one another. Therefore, for a given table of size m, linked with a problem size N, there are $\binom{N}{m}$ possible sets of endpoints for this table. This means that the minimum amount, in bits, to encode the part of a table corresponding to the endpoints is given by

$$M^{EP}_{\text{lower}} = m \log_2 \binom{N}{m}.$$

This is indeed a lower bound of average storage size in bits of the endpoints in a table, since any index referencing such a set would have to be encoded in as many bits.

A first idea to improve the storage of tables is to compress them, using universal data compression algorithms such as, *e.g.*, Dynamic Huffman coding [Knu85], or LZ77 algorithm [ZL77]. However, it implies that the information in the table must somehow be uncompressed into RAM to be in a usable form when it is processed by the online phase. One of the main issues with TMTOs is that tables require fast access to rows that are randomly determined during the online phase. Compressing the table as a whole defeats the random access requirement and most of the common compression techniques are therefore not applicable to TMTOs. Ideally, we would like to significantly compress tables while retaining a random access capability.

In general, we will want to minimise the access time to the table. A way to achieve this is to fit the whole table in the fastest memory available. As a consequence, a compression scheme that works with a very fine granularity, in that it can uncompress only a small amount of the table, is favorable to a sequential algorithm that requires all of the table, starting from the beginning, to be uncompressed beforehand. With a finer compression scheme, one can fit almost all the memory with the compressed table, allowing fast random access to sections of the table that can be uncompressed at runtime. Therefore, online phase algorithms that require negligible amount of additional memory to process the tables are sought, so that the memory can be used to store the tables themselves.

3.2.2 Indexing endpoints

One of the first techniques to be applied to the compression of endpoints is the *indexing technique* also called *prefix-suffix decomposition.* This technique is not exclusive to TMTOs, and was probably used in TMTO implementations before it appeared in literature. Biryukov, Shamir and Wagner describe the technique for the Hellman tables in [BSW00]. Oechslin, in [Oec03], mentions the usage of the technique for the implementation of rainbow tables in his experiment, as well as in his tool Ophcrack [TO05]. It is further discussed in the work of Avoine, Junod, and Oechslin [AJO08], as well as in [AC14]. The latter provides an analysis of the optimality of the method.

The indexing technique relies on the observation that in a group of elements that possess identical leading bits, these bits can be omitted for all but one of each element of the set, without losing any information on said set. In tables, endpoints are sorted, which implies that there is a high chance that for some size of prefix, the leading bits of close elements are indeed the same.

We can decompose each n-bit endpoint $EP_x \in \mathcal{N}$, where $1 \leq x \leq m$ is the index of the endpoint in the table, in a prefix P_x and a suffix S_x:

$$\forall p \geq 0, \quad \exists P_x, S_x, \quad \text{s.t.} \quad EP_x = P_x 2^p + S_x.$$

The prefix is p-bit long and the suffix is encoded on $s = \lceil \log_2 S_x \rceil$ bits. We separate the table into two sequences $P = (P_1, P_2, \dots, P_m)$ and $S = (S_1, \dots, S_m)$. Due to the sorting, as mentioned above, the first sequence is of the form

$$(P_x)_{1 \leq x \leq m} = (\underbrace{a_1, a_1, \dots, a_1}_{i_1}, \underbrace{a_2, \dots, a_2}_{i_2}, \dots, \underbrace{a_k, \dots, a_k}_{i_k}),$$

where $a_1, a_2, \dots, a_k$ are the k distinct prefixes. The smaller p is, the smaller k is, and so the more the sequence contains duplicates. The core idea is to discard the prefix sequence while keeping a table of pointers which associates each distinct prefix to the offset at which the corresponding first suffix is. In other words, given the notations above, we build an associative array I, which is of maximal size 2^p, such that

$$\forall p_x \in P, \quad \text{s.t.} \quad p_x = a_x, \quad I(p_x) = \iota_x$$

where $\iota_1 = 0$, and

$$\forall x > 1, \quad \iota_x = \sum_{j=1}^{x-1} i_j.$$

Note that the offset is in the range $\{0, \dots, m-1\}$ and can therefore be encoded on $\lceil \log_2 m \rceil$ bits. Figure 3.1 illustrates the decomposition for the following sequence of decimal integers:

$$(1, 32, 57, 91, 122, 180, 201, 262).$$

I		
000	→	0
001	→	4
002	→	6

offset	P	S
0	000	01
	000	32
	000	57
	000	91
4	001	22
	001	80
6	002	01
	002	62

Figure 3.1 – Prefix-suffix decomposition

The table of pointers is stored in only one occurrence, that is "outside" the rows, which implies we almost discard all prefixes in the table except for a negligible amount of data. The amount of bits per row therefore become s. The total amount of bits required to store such a table, including the table of pointers, is then given by

$$M_{\text{ps}}^{EP} = 2^p \lceil \log_2 m \rceil + ms. \tag{3.2}$$

The location p of the separation between prefix and suffix influences the efficiency of the technique. If p is too low, few bits are discarded from each row. If it is too high, the table of pointers is not negligible anymore with regard to the table. In [AC14], Avoine and Carpent show that this parameter can be optimally determined. This parameter is given in Proposition 3.1.

Proposition 3.1. *Given a set of m elements encoded with the prefix-suffix decomposition, using either the $\lfloor p^* \rfloor$ or $\lceil p^* \rceil$ parameter, where*

$$p^* = \log_2 \frac{m}{\lceil \log_2 m \rceil \log 2},$$

gives an optimal compression ratio of the set.

Proof. We consider the real function defined by the right-hand side of Equation (3.2).

$$\begin{aligned} \mu_{n,m} : \quad & \mathbb{R} \to \mathbb{R} \\ & p \mapsto 2^p \lceil \log_2 m \rceil + m(n-p). \end{aligned}$$

If we consider its second derivative on p, we have

$$\mu''_{n,m}(p) = 2^p \lceil \log_2 m \rceil \log^2 2 > 0,$$

so $\mu_{n,m}$ possesses a minimum. We search for p^* such that $\mu'_{n,m}(p^*) = 0$. Then

$$2^p \lceil \log_2 m \rceil \log 2 - m = 0,$$

and

$$2^{p^*} = \frac{m}{\lceil \log_2 m \rceil \log 2},$$

which is equivalent to the proposition's statement.

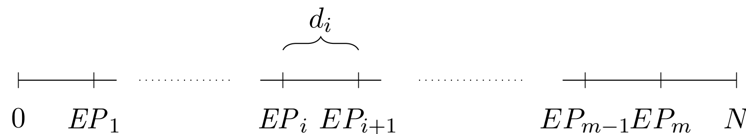
The actual best integer value (in bits) $\lfloor p^* \rfloor$ or $\lceil p^* \rceil$ has to be determined empirically by testing both values against the total amount of bits required to store the table, and picking the one which allows the minimum storage. □

3.2.3 Compressed delta encoding

Also in [AC14], another, more efficient, approach to table storage is introduced. The authors of [AC14] first determine a compression scheme suitable for sorted random integers, denoted with *compressed delta encoding*, and propose a technique which overcomes the aforementioned issues regarding the sequential nature of compression: with their method, the table does not have to be fully decompressed in order to access any part of it.

The work [AC14] targets the rainbow trade-off with the compressed delta encoding technique, but it is stated in [AC14] that it can be applied to the other variants as well. However, while this is not explicitly acknowledged in the paper, the developments in [AC14] make it clear that the formulas apply to the case where all the endpoints are distinct, *i.e.*, the technique is analysed in the context of the perfect version of the rainbow trade-off.

The key point of the compression is to deal with the differences (or deltas) between two consecutive endpoints, instead of the endpoints themselves. In fact, such an idea was hinted for rainbow tables in [BBS06]. Consider m endpoints $EP_1, EP_2, \ldots, EP_m$, and, for all $i \in \{1, \ldots, m-1\}$, their differences $d_i = EP_{i+1} - EP_i$. Since the endpoints are sorted, d_i corresponds to the distance between two consecutive endpoints, with $1 \leq d_i \leq N - m + 1$. The endpoints are uniformly distributed in $\mathcal{N}$ though, and in average are spaced equally:



We can see that in this case, for all $i \in \{1, \ldots, m-1\}$, the d_i will be in the order of $\frac{N}{m+1}$, which is much lower than N, since $m \gg 1$, and therefore they can be stored with much less bits than the whole endpoints.

In [AC14], Avoine and Carpent reason about $D_i = d_i - 1$, since all endpoints are distinct, and they establish the probability function and the expected value of the distribution of these differences, which are stated in Proposition 3.2.

Proposition 3.2. *Given m endpoints $EP_1, \ldots, EP_m$, such that $EP_1 < EP_2 < \ldots < EP_m$, we note, for all $i \in \{1, \ldots, m-1\}$, $D_i = EP_{i+1} - EP_i - 1$, and $\mathcal{D}$ the random variable taking the possible values of D_i.*

1. *The probability of $\mathcal{D}$ taking any value d in $\{0, \ldots, N-m\}$ is given by*

$$\Pr(\mathcal{D} = d) = \frac{\binom{N-d-1}{m-1}}{\binom{N}{m}},$$

   and $\Pr(\mathcal{D} = d) = 0$ otherwise.

2. *The expected value of $\mathcal{D}$ is*

$$\mathbb{E}[\mathcal{D}] = \frac{N-m}{m+1}.$$

Proof. Let d be an integer in $\{0, \ldots, N-m\}$. If we have $D_i = EP_{i+1} - EP_i = d$, and we fix EP_i, we can choose EP_{i+1} among the $m-1$ other endpoints, and it has $N-d-1$ possible values. There are $\binom{N-d-1}{m-1}$ different possibilities for such a choice. In total, there are $\binom{N}{m}$ possibilities to choose a random endpoint. Then, the probability of the difference to be exactly d for randomly chosen endpoints is given by

$$\frac{\binom{N-d-1}{m-1}}{\binom{N}{m}}.$$

Therefore, the expected value of the difference is given by

$$\sum_{d=0}^{N-m} d \Pr(\mathcal{D} = d) = d \sum_{d=0}^{N-m} \frac{\binom{N-d-1}{m-1}}{\binom{N}{m}}. \tag{3.3}$$

The second part of Equation (3.3) corresponds to the value studied in the second Problem of Section 5.2 of [GKP89], when replacing the variables m, k, and n in [GKP89], by N, d, and $N-m$ respectively. It is shown in [GKP89] that it is equal to the proposition's stated expected value. □

When observing the distribution of the differences between endpoints, the authors of [AC14] remark that it is in fact very similar to a geometric distribution. Let $\mathcal{D}'$ be a random variable following a geometric law such that $\mathbb{E}[\mathcal{D}'] = \mathbb{E}[\mathcal{D}]$. The claim is that $\mathcal{D}'$ is good enough to model the structure of the differences D_i. One can see in Figure 3.2, that, indeed, the geometric probability density function (pdf) of parameter $p = \frac{N-m}{N+1}$ closely follows a sample of $\mathcal{D}$ taken from an experiment of ours. Once the distribution of the differences is determined, it becomes possible to exploit this structure to optimally encode these deltas.

In [Gol66], Golomb derives a family of codes, indexed by a tunable parameter $\kappa \in \mathbb{N}^*$, that define a reversible prefix-free mapping between integers and binary variable-length codewords. A given $x \in \mathbb{N}$ is decomposed into q and r such that $qr = x$:

$$q = \left\lfloor \frac{x-1}{\kappa} \right\rfloor, \quad r = x - q\kappa - 1,$$

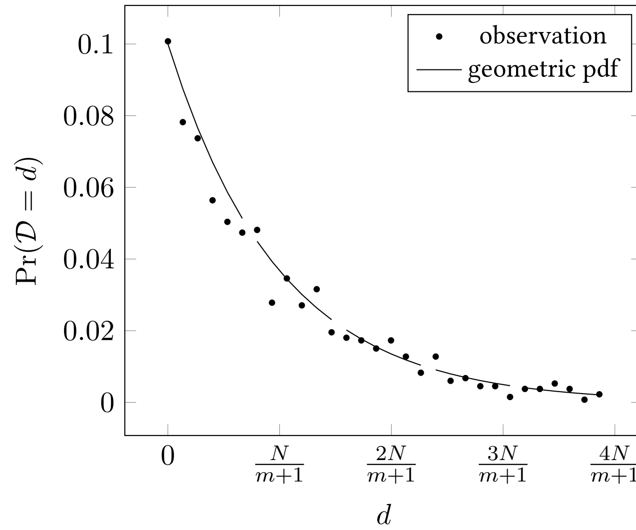


Figure 3.2 – Distribution of $\mathcal{D}$ with $N = 10^6$ and $m = 10^3$

and then q is encoded in unary coding, and r encoded with truncated binary coding. These codes are optimal to encode values that are geometrically distributed with probability $p = \frac{1}{2}$, *i.e.*, the expected length of these codewords is minimal. This property is extended in [GV75] where it is shown that there is an optimal Golomb code for all geometrically distributed values with all probabilities $p \in]0, 1[$. In [Ric79], Rice designs an entropy coding aiming at being efficient in computers which perform much better with binary arithmetic than with integers. He focuses on a subset of the Golomb codes in which $\kappa = 2^k$, for some $k \in \mathbb{N}$. The resulting *Rice codes* are slightly less optimal, but much simpler than generic Golomb codes. In particular, r can now be represented in simple binary coding, and q can be obtained with a binary shift, rather than with a division.

In [Kie04], Kiely focuses on Rice coding and establishes results on the code rate given by

$$R_\kappa = \sum_{j=0}^{\infty} \left(\left\lfloor \frac{j}{2^\kappa} + 1 + k \right\rfloor \right) \Pr(\delta = j), \tag{3.4}$$

which is the expected length of a codeword $\delta \in \mathbb{N}^*$, in the specific case of geometrically distributed values. Proposition 3.3, gives a simplification of Equation (3.4), that was stated in [AC14], in the case of geometrically distributed elements.

Proposition 3.3. *The expected number of bits per codeword given by a Rice coding of parameter κ of a collection of elements distributed following a geometric law of parameter p is given by*

$$R_\kappa = \kappa + \frac{1}{1 - p^{2^\kappa}}.$$

Proof. The code rate for a random variable δ is given by

$$R_\kappa = \kappa + 1 + \mathbb{E}\left[\frac{\delta}{2^\kappa}\right] = \kappa + 1 + \frac{1}{2^\kappa}(\mu - r_\kappa), \tag{3.5}$$

where $\mu = \mathbb{E}[\delta]$ and $r_\kappa = \mathbb{E}\left[\delta - 2^k \left\lfloor \frac{\delta}{2^\kappa} \right\rfloor\right]$. If δ follows a geometric law, for all $j > 0$, $\Pr(\delta = j) = p^j(1-p)$, and it is shown in [Kie04] that

$$r_\kappa = \sum_{j=0}^{\infty} \left(j - 2^\kappa \left\lfloor \frac{j}{2^\kappa} \right\rfloor\right)(1-p)p^j = \mu + 2^\kappa \left(1 - \frac{1}{1-p^{2^\kappa}}\right),$$

which, once injected into Equation (3.5), produces the claim. □

An optimal parameter is also determined in [Kie04], and given in Proposition 3.4.

Proposition 3.4. *The optimal Rice coding parameter to encode values distributed according to a geometric law of parameter p is given by*

$$\kappa_{opt} = 1 + \left\lfloor \log_2 \left(\frac{\log(\varphi - 1)}{\log p} \right) \right\rfloor,$$

where $\varphi = \frac{1+\sqrt{5}}{2}$ is the golden ratio.

Proof. The quantity $\left\lfloor \frac{j}{2^\kappa} \right\rfloor + 1 + \kappa$ is convex in k on $\mathbb{R}^+$, so R_κ from Equation (3.4) is clearly convex and possesses a minimum at κ_{opt} such that $R_{\kappa_{\text{opt}}} \leq R_{\kappa_{\text{opt}}-1}$, and $R_{\kappa_{\text{opt}}} \leq R_{\kappa_{\text{opt}}+1}$. The first of theses inequalities is true if and only if

$$p^{2^{\kappa_{\text{opt}}}} + p^{2^{\kappa_{\text{opt}}-1}} \geq 0,$$

which is satisfied only if

$$p^{2^{\kappa_{\text{opt}}}} \geq \frac{\sqrt{5}-1}{2}. \tag{3.6}$$

Then, the optimal parameter is the largest integer that satisfies Equation (3.6), or equivalently

$$\kappa_{\text{opt}} \leq 1 + \log_2 \left(\frac{\log \varphi - 1}{\log p} \right).$$

□

The straightforward application of Propositions 3.3 and 3.4 to the $\mathcal{D}$ distribution wrap-up into the following theorem given in [AC14]:

Theorem 3.5. *If we assume the differences of the endpoints (minus one) to be geometrically distributed, the optimal compression rate is given by*

$$R_\kappa = \kappa + \frac{1}{1 - \left(\frac{N-m}{N+1}\right)^{2^\kappa}},$$

where

$$\kappa_{opt} = 1 + \left\lfloor \log_2 \left(\frac{\log(\varphi - 1)}{\log\left(\frac{N-m}{N+1}\right)} \right) \right\rfloor.$$

Since the encoding works with deltas, and not with the endpoints themselves, additional work has to be done in order to exploit the compression. In particular, we have to switch between endpoint and delta representation in order to perform the online phase. The same dilemma as the one with general compression techniques on tables applies. If we only work with deltas, this means that we only have to store the first endpoint (or none at all if $EP_1 = 0$), and everything else is a delta which is encoded optimally. However, during the online phase, we then have to start the decoding starting from this first endpoint which means that we have to read half of the table in average for each access.

That is why the authors of [AC14] propose to split the table in blocks of size L, containing each a sequence of deltas relating to a local endpoint. We denote with $l = \lceil \frac{m}{L} \rceil$ the number of blocks, and $D_i^{EP_k}$, for $k \in \{1, \ldots, m\}$ and $i \in \{0, \ldots, L-1\}$, the ith delta corresponding to the EP_{k+i}, whose sequence starts from EP_k. A block is then defined by

$$\left\{ \left(SP_{kl+i}, D_i^{EP_{kl}} \right) \right\}_{1 \leq i < L} .$$

The idea is that for any search of a row k in the table, there is an endpoint at row $k' \leq k$ such that $k - k' < L$ is small and fewer data has to be decoded to reach the row k. However, at the extreme, if L is too small, we need very few data from the table at each search, but we have to store many endpoints in full, and the benefits from the compression are severely dampened.

An additional table is computed with pointers to the beginning of the blocks. With this technique, an encoded endpoint is described by both the position of the right block in the table, and the offset of the right endpoint to be decoded. During the online phase, once the right block is identified, the block is decoded up to the given offset, and the endpoint is recovered. This requires memory for a table of pointers to the beginning of L blocks, L endpoints and all the encoded deltas. The total amount of storage required for the endpoints in a table, in bits, is then given by

$$M_{\text{cde}}^{EP} = L(\lceil \log_2 mR_{\kappa_{\text{opt}}} \rceil + \lceil \log_2 m \rceil) + mR_{\kappa_{\text{opt}}} .$$

Regarding the choice of L, the authors of [AC14] specify that the number of indexed endpoints, *i.e.*, the number of blocks, should be small with regard to m, and claim that when this is the case the loss in compression rate compared to the degenerate case of $L = 1$, *i.e.*, when all the endpoints are encoded as delta, is negligible. The time overhead given by the average decoding of $\frac{m}{2L}$ endpoints before reaching the searched value should be negligible before the computation of the online chain. In [AC14], Avoine et al. choose L in such a way that the decoding takes the same amount of time as an application of F. In their experiment, with $N = 2^{40}$ and $m = 2^{24}$, they observe an increase of 0.006% of the online time, with a memory overhead of 0.6%. Moreover, their experiment confirms that the assumption on the distribution of the differences of endpoints allows a near optimal compression, as it shows that the compressed delta encoding method is very close to the theoretical lower bound (see Figure 4 of [AC14]).

3.2.4 Truncation

The truncation of the endpoint, as its name suggests, consists in omitting a part of the endpoint when storing it in the table. Contrarily to the previous methods, this is a lossy technique, and the loss of information has to be made up for during the online phase. Note that in this regard, this is different from the DP truncation technique which does not lose any information as the omitted part is guaranteed to be the same. It is not clear where the idea originates from. It is used by Biryukov, Shamir, and Wagner in [BSW00], where it is applied to the DP trade-off beyond the $\log_2 k$ bits usually removed from

a distinguishing property of $\frac{1}{k}$. The authors of [BSW00] remark that the truncation allows to save a considerable fraction of the space while still being able to disambiguate a given endpoint in most cases, so this come at a modest cost.

Note that this technique introduces a new kind of truncation-related false alarm, where the potential chain endpoint matches an endpoint which has the same truncation as another. Assume that we use a truncation of r bits, that is, we omit the r least significant bits of the endpoints in the table. During the online phase, a potential chain will lead to some potential endpoint PEP. The table does not contain full endpoints so, instead, the truncation of PEP by r bits is compared to the elements in the table. Then, either the element is not found, so we are sure that the endpoint was not in the table before truncation, or there is a partial endpoint which matches the partial potential endpoint, in which case we proceed as in the classical algorithm when an endpoint is found. In this second case, we find either an answer or a false alarm. The technique relies on the hope that the second event will not happen as frequently as the first one. The idea is that when $m \ll N$, endpoints are sufficiently distant from each other, so that there is a low probability that the stored values are too close to any random value.

In the case of the perfect trade-off, the truncation itself induces an overhead, as previously distinct endpoints can have the same partial value after the truncation. In [Hon10], Hong claims that this happens with negligible probability for typical parameters of the rainbow and Hellman trade-offs. An analysis of the endpoint truncation method is performed in [HM13]. In the example the authors consider, a truncation of 5 bits only gives a time overhead T of 2.5%. This means that, if the memory savings exceed $\sqrt{2.5} = 1.6\%$, which is easily obtained for typical problem sizes, the trade-off between truncation and memory gain is favourable.

3.3 Checkpoints

In [AJO05], Avoine, Junod, and Oechslin introduce a novel technique to reduce the online running time using *checkpoints*. Traditionally, only the starting point and endpoint are kept from the offline chain in the TMTO matrix. The idea of [AJO05] is to keep, in addition, information on the offline chain that can be exploited to avoid computational work made during the online phase, without incurring too much overhead on the memory. In fact, the checkpoint technique was initially thought of as a way to exploit unused bits in implementations of tables that were aligned in memory in such a way that rows were multiples of bytes.

Up until this technique, the verification chain had to be fully constructed to confirm the chain or discard it if it was a false alarm. When a bitpacked implementation is used, this idea goes in the opposite direction from the storage techniques, but the gain in time is worth the additional memory. It is indeed almost[1] always possible to compensate the overhead by increasing the length t of a table with checkpoints to a length t' (with $t' > t$), in a way that a table of length t without checkpoints would take the same running time. If the probability of success is kept identical, the size of the table with checkpoints will be smaller than the unmodified table.

3.3.1 Generating checkpoints

The checkpoint technique of [AJO05] requires a slight modification in the offline phase procedure. In particular, during the computation of each chain, additional information is collected. Let us define a set

[1] This does not apply if we are dealing with tables of maximal size, for which, by definition, each additional row will introduce redundancy in the table.

of k positions $(\alpha_i)_{i\leq k}$ in $\{1,\dots,t\}$. These will be the checkpoints of the pre-computation chain.

The authors of [AJO05] then introduce a *check function*

$$G:\mathcal{N}\to\{0,1\},$$

which extracts a bit of information from a given element of the matrix. An example of such a check function, the one used in [AJO05], is the parity function which outputs the lowest significant bit of an element.

During the computation of the TMTO matrix, for each chain, the function G is applied to elements at positions α_i, and the output bits of G are recorded. The resulting set of checkpoint bits

$$G(x_{\alpha_1})|G(x_{\alpha_2})|\dots|G(x_{\alpha_k}),$$

where | denotes the concatenation operator, is stored alongside the endpoint, effectively becoming its last bits. Note that when appending the bits to the endpoint, the row remains compatible with the sorting that could be applied to the table.

3.3.2 Online phase

We now describe the online phase modifications induced by the checkpoint technique. During the online phase, at position α_i, we calculate the value of $G(F^{\alpha_i}(SP_x))$, where SP_x is the current corresponding starting point of our potential endpoint. Either the value matches the one which is stored for this particular position in the table, in which case we continue, as the current chain must be the right one (but we are not sure of it, since we may have a collision on G with the correct value), or the value does not match, in which case we are sure the chain is incorrect, and we can stop rebuilding.

An analysis of the average online time is provided in [AJO05] as well as in its extended version [AJO08]. An independent computation of this same online time is performed by Hong in [Hon10], which also treats the case of the non-perfect rainbow trade-off, and the Hellman trade-off. A later work by Hong [Hon16] notes minor discrepancies between the two works [AJO08] and [Hon10], concerning the amount of work reduced by the checkpoint technique. When comparing the obtained formulas with the actual experimental online behaviour, Hong establishes that the formulas of [Hon10] follow the reality more closely. However, Hong acknowledges that the full online time given in [AJO08] is in fact close to the reality due to the fact that the inaccuracies of [AJO08] are the most significant for leftmost columns which account for a tiny proportion of the answers of the online phase. Indeed, recall that in a perfect trade-off, the elements of $\mathcal{N}$ tend to appear more than once in the TMTO matrix, which means that in average, the answer that is found corresponds to elements that are rather in the right part of the matrix. The formulas of [Hon10] are considered in the following developments.

We first introduce the expected gain in terms of computation of F when using a single checkpoint in a table. While, in theory, the checkpoint technique applies to all TMTO variants, the related work in the literature only deals with the Hellman and the rainbow trade-off. There is no known analysis of the checkpoint technique on either the DP trade-off, or its derivative, the fuzzy trade-off.

To compute the saved iterations, we must first determine the probability for a chain starting from a given column to merge before a given checkpoint. To do so, the following propositions give the size of the preimages of the elements of a given column in the TMTO matrix, given the amount of endpoints m. The case were the checkpoint is placed at position t is the classic false alarm probability, already computed for the different TMTO variants, and recalled in Proposition 3.6, in a form stating set sizes, as given in [Hon10].

Proposition 3.6. *We denote with $\mathcal{S}$ a set of m_0 random starting points, and with $\mathcal{E} = F(\mathcal{S})$ a set of m corresponding endpoints.*

Then, the preimage of $\mathcal{E}$ by F^k for $k < t$ is expected to be of size

$$|F^{-k}(\mathcal{E})| = m(1+k)\left(1 - \frac{mk}{4N}\right).$$

Proof. Let $D_0, D_1, \ldots, D_t$ designate the iterated images of F of respective sizes $m_0, m_1, \ldots, m_t$, such as defined in Proposition 2.1.

Consider a potential chain of length $k < t$. The probability of a random first element $x_0 \in \mathcal{N}$ of the potential chain not to be in D_{t-k} is $1 - \frac{m_{t-k}}{N}$. Since F is random, the probability of the next element $x_1 = F(x_0)$ not to be in D_{t-k+1} is independent and equal to $1 - \frac{m_{t-k+1}}{N}$. By induction, we conclude that the probability for an element not to be in any of the previous columns is given by

$$\Pr(x \text{ not in previous columns}) = \prod_{j=k}^{t} 1 - \frac{m_k}{N},$$

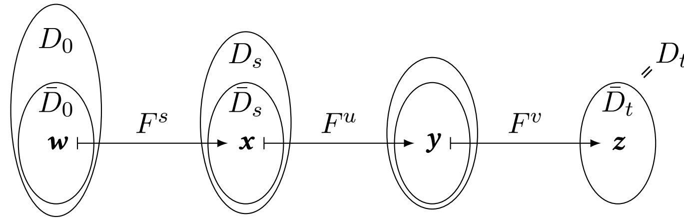
and then the probability of a merge is $1 - \Pr(x \text{ not in previous columns})$. When we apply Proposition 2.1 to approximate m_k, a series of cancellations within the product appears and gives

$$m_k(1+k)\left(1 - \frac{m_t k}{2(2N - m_t)}\right),$$

which can be approximated in the claim, when $m_t = m \ll N$. □

The case of the preimage of a column which is not the last is more complicated, and given in the following proposition, which is a slightly edited version of the corresponding proposition in [Hon10].

Proposition 3.7 (Merging set size between two columns)**.** *Let $F : \mathcal{N} \to \mathcal{N}$ be a random function. Let $s > 0, u > 0, v > 0$ and t be integers such that $s + u + v = t$. Let $D_0 \subset \mathcal{N}$ be a set of starting points, and $D_t = F^t(D_0)$ the set of the m corresponding endpoints. We denote $\bar{D_0} \subset D_0$ a set of m starting points such that $F^t(\bar{D_0}) = D_t$, that is, for each $e \in D_t$, we chose at random one element of D_0 in each preimage $F^{-t}(e)$, to constitute $\bar{D_0}$. We define D_s in a similar fashion as seen in the diagram below:*



Then,

$$|F^{-u}(F^{s+u}(\bar{D_0}))| = m(u+1) + \frac{u(u+2)}{v^2}\left(mv + 2N\ln\left(1 - \frac{mv}{2N}\right)\right).$$

Proof. The complete proof of the original version of this proposition can be found in [Hon10, pp. 5-16]. Due to its length and high technicality, we only present hereafter the key points of the proof, which we believe are important in order to highlight how the formula was constructed.

The proof is based on the concept of i-node from graph theory: $x \in \mathcal{N}$ is said to be an i-node for F if $|F^{-1}(x)| = i$. The concept extends to iterated functions, and we denote

$$\mathcal{R}_{k,i} = \{y \in \mathcal{N} \mid |F^{-k}(y)| = i\},$$

the set of all i-nodes for F^k. The probability of an element to be an i-node for F^k is $p_{k,i} = \frac{|\mathcal{R}_{k,i}|}{N}$.

Note that every element $x \in \mathcal{R}_{k,i}$ has i preimages, hence the probability

$$\Pr(F^k(x) \text{ is an } i\text{-node}) = ip_{k,i}. \tag{3.7}$$

Consider $D_u = F^u(D_0)$, and $D_{u+v} = F^v(D_u)$. If we consider a j-node $x \in D_u$ for F^v, we can say for its j preimages that

$$\forall k \in \{1, \ldots, j\}, \exists i_k, \quad \text{s.t.} \quad x \text{ is a } i_k\text{-node for } F^u,$$

so x is a $(\sum_k i_k)$-node for F^{u+v}. Therefore, the probability that x is an i-node for F^{u+v} is $p_{u,i_1}p_{u,i_2}\ldots p_{u,i_j}$, where $(i_k)_{1\le k\le j}$ is a partition of i, that is

$$\sum_{i_1+i_2+\ldots+i_j=i} p_{u,i_1}p_{u,i_2}\ldots p_{u,i_j} \tag{3.8}$$

If we denote, for $k > 0$, the power series

$$\mathcal{P}_k(x) = \sum_{i=0}^{\infty} p_{k,i}x^i,$$

it can be shown that $\mathcal{P}$ is convergent on $\mathbb{R}$ and that for all $k > 0$, P is injective, so $\mathcal{P}_k$ is defined for $k \in \mathbb{Z}$. Moreover, $\mathcal{P}$ satisfies the relation $\mathcal{P}_{u+v}(x) = \mathcal{P}_v(\mathcal{P}_u(x))$, which allows us to relate the iterated sizes of the distinct elements of a column $m_0, m_1, \ldots, m_t$,

$$\forall k \in \mathbb{Z}, \forall u > 0, \quad 1 - \frac{m_{k+u}}{N} = \mathcal{P}_u\left(1 - \frac{m_k}{N}\right), \tag{3.9}$$

with $\mathcal{P}_0(x) = x$. The relation (3.9) is somewhat reminiscent of Proposition 2.1 on image sizes of iterated random functions, and following its proof yields a similar approximation for $\mathcal{P}_k$:

$$\mathcal{P}_k(x) \approx 1 - \frac{2(1-x)}{2 + k(1-x)}. \tag{3.10}$$

In the following we denote with $\lceil \mathcal{P}(x) \rfloor_i$ the ith term of the power series $\mathcal{P}$. The power series $\mathcal{P}$ allows the manipulation of the $p_{k,i}$ across the iterations of F. In particular, notice that the right-hand side of Equation (3.8) is equal to $\lceil \mathcal{P}_k(x)^j \rfloor_i$.

Now consider x, y, and z such that $y = F^u(x)$ and $z = F^v(y)$. We have, similarly to how we obtain Equation (3.8),

$$\Pr(|F^{-(u+v)}(z)| = i) = \sum_{k+i_2+\ldots+i_j=i} p_{u,i_2}\ldots p_{u.i_j} = \lceil \mathcal{P}_u(x)^{j-1} \rfloor_{i-k}.$$

Also,

$$\Pr(F^{u+v} \text{ is an } i\text{-node}) = \sum_{k,j} \left(kp_{u,k}jp_{u,j} \Pr(|F^{-(u+v)}(z)| = i)\right), \tag{3.11}$$

and recall that according to Equation (3.7), the probability presented by Equation (3.11) also equals to $ip_{u+v,i}$, so by taking the partial sum on j of Equation (3.11) we can deduce

$$\Pr\left(y \text{ is a } k\text{-node for } F^u \middle| z \text{ is an } i\text{-node for } F^{u+v}\right) = \frac{kp_{u,k}}{up_{u+v,i}} \sum_{k+i_2+\ldots+i_j=i} p_{u,i_2}\ldots p_{u,i_j}\,.$$

For each group of i-node by F^{u+v}, the amount of preimages by F^u is

$$\sum_{k=0}^{i} k \Pr\left(|F^{-u}(y)| = k \;\middle|\; |F^{-(u+v)}(z)| = i\right).$$

The amount of i-nodes by F^{u+v} is given by

$$Np_{u+v,i}\left(1-\left(1-\frac{m_s}{N}\right)^i\right).$$

Then the amount of elements $x \in \bar{D}_0$ such that $F^u(x) \in \bar{D}_s$, where $\bar{D}_s$ is defined the same way from D_s as $\bar{D}_0$ is from D_0, is given by

$$\sum_i Np_{u+v,i}\left(1-\left(1-\frac{m_s}{N}\right)^i\right)\sum_k k\frac{kp_{u,k}}{ip_{u+v,i}}\sum_j jp_{u,j}\sum_{k+i_1+i_2+\ldots+i_j=i} p_{u,i_1}p_{u,i_2}\ldots p_{u,i_j}\,,$$

which, when substituting by terms of $\mathcal{P}$, becomes

$$N\sum_i \frac{1}{i}\left(1-\left(1-\frac{m_s}{N}\right)\right)\sum_k \lceil (x\mathcal{P}'_u(x))'\rfloor_{k-1}\cdot\lceil \mathcal{P}'_v(\mathcal{P}_u(x))\rfloor_{i-k}$$

and then, when approximating the sum by a definite integral:

$$\int_{\mathcal{P}_s\left(1-\frac{m_0}{N}\right)}^{1} (x\mathcal{P}'_u(x))'\mathcal{P}'_v(\mathcal{P}_u(x))\,dx\,.$$

By modifying the lower bound of the integral, we can obtain the amount of the preimages of $F^{s+u}(\bar{D}_0)$ by F^u:

$$\int_{\mathcal{P}\left(1-\frac{m_s}{N}\right)}^{1} (x\mathcal{P}'_u(x))'\mathcal{P}'_v(\mathcal{P}_u(x))\,dx\,.$$

By using the approximation of $\mathcal{P}$ in Equation (3.10), and computing the integral, we arrive at the claim. □

These two propositions, which quantify the expected amount of merges between any two columns in the rainbow matrix, will allow us to establish probabilities about these merges in the context of checkpoints. A result from [Hon10] is given in Theorem 3.8 which gives the amount of computations that can be avoided with a single checkpoint, in the Hellman TMTO.

Theorem 3.8 (Hellman online time with a checkpoint). *If we place a single checkpoint at column $t-c$ in a Hellman TMTO of parameters N, m, t, the number of applications of F that are avoided by the checkpoint is given by*

$$\frac{m}{N}\left(\frac{c(t-c)(t-c+1)}{4}\right).$$

Proof. We denote the set of elements of $\mathcal{N}$ in the kth column of the Hellman matrix by HM_k. The checkpoint can only avoid computations in the case of a false alarm. Consider that we get a false alarm at the kth iteration of the online phase.

If $k < c$, the potential chain starts at least at column $t - c + 1$ and therefore cannot be filtered by the checkpoint. If $k \geq c$, either the answer is found with probability $\frac{|HM_{t-k}|}{N}$, or the potential chain merges before the checkpoint, in which case the checkpoint is useless. This happens with probability

$$\frac{|F^{-(k-c)}(HM_{t-c})|}{N}. \tag{3.12}$$

In the case the merge happens after the checkpoint, which filters out the potential chain with probability $\frac{1}{2}$, the probability of the chain to go through the checkpoint undetected is

$$\frac{1}{2}\frac{|F^{-k}(HM_t)| - |F^{-(k-c)}(HM_{t-c})|}{N}. \tag{3.13}$$

We know from Proposition 3.6 that

$$|F^{-k}(HM_t)| = m(1+k)\left(1 - \frac{mk}{4N}\right).$$

With the common assumption on the matrix stopping rule $\frac{mt^2}{N} = 1$, and if we assume $t \gg 1$, we have

$$\frac{mk}{4N} = \frac{mt^2}{N}\frac{k}{4t^2} = \frac{k}{4t^2} < \frac{1}{t} \approx 0,$$

so

$$|F^{-k}(HM_t)| = m(1+k).$$

From Proposition 3.7, we have that

$$|F^{-u}(HM_{t-c})| = m(u+1) + \frac{u(u+2)}{c^2}\left(mc + 2N\ln\left(1 - \frac{mc}{2N}\right)\right).$$

With the same assumptions, we can see that $mc < mt = \frac{N}{t}$, so $\ln(1 - \frac{mc}{2N}) \approx -\frac{mc}{2N}$, and the proposition simplifies into

$$|F^{-k}(HM_{t-c})| = m(k+1).$$

Therefore, the probability that a false alarm happens without the possibility for the checkpoint to filter it out is

$$|F^{-k}(HM_{t-c})| - \frac{m}{N} = \frac{mk}{N},$$

and when passing through the checkpoint, it avoids detection with probability

$$\frac{1}{2}\frac{m(k+1) - m(k-c+1)}{N} = \frac{mc}{2N}.$$

The total number of false alarms is given by

$$\sum_{0<k\leq t}(t-k+1)\frac{mk}{N} - \sum_{c<k\leq t}(t-k+1)\frac{mc}{2N}. \tag{3.14}$$

The second term of (3.14) represents the number of false alarms avoided by the checkpoint and simplifies in the stated claim. □

No general formula is given in [Hon10], and a closed formula does not seem in reach for the case of an arbitrary number of checkpoints. However, it is hinted in [Hon10] how to extend the result to several checkpoints, starting from Theorem 3.8. The probability is computed for each section between the starting column of the potential chain and the next checkpoint to which are added the compound probability that the potential chain escapes the filtering process. The example of the case of two checkpoints at position $t - c_1$ and $t - c_2$ is given. The amount of computations of the one-way function due to the false alarms in a Hellman table is then given by

$$\begin{aligned}\sum_{0<k\leq c_1} (t-k+1)\frac{mk}{N} + \sum_{c_1<k\leq c_2} (t-k+1)\left((k-c_1)+\frac{c_1}{2}\right)\frac{m}{N} \\ + \sum_{c_1<k\leq c_2} (t-k+1)\left((k-c_2)+\frac{c_2-c_1}{2}+\frac{c_1}{4}\right)\frac{m}{N}.\end{aligned} \tag{3.15}$$

While complicated for an arbitrary number of checkpoints, it can be seen how to compute the online time when few checkpoints are involved.

In [Hon10], an evaluation of the rainbow trade-off with a single checkpoint, both in the perfect and the non-perfect version, is given. The result it yields is the expected number of saved applications of F.

Theorem 3.9. *If a single checkpoint is placed at position $t - c$ in the table, the amount of computation of the one-way functions removed by that single checkpoint for the rainbow trade-off is*

$$\sum_{k=c+1}^{t} (t-k+1)\prod_{j=1}^{k-1}\left(1-\frac{m_{t-j}}{N}\right)\frac{mc}{2N}$$

for the non-perfect version, where the m_i represent the number of distinct elements in the column i, and

$$\sum_{k=c+1}^{t} (t-k+1)\left(1-\frac{m}{N}\right)^{k-1}\left\{\frac{mc}{2N}-\left(\frac{mc}{2N}+\ln\left(1-\frac{mc}{2N}\right)\right)\frac{1}{c^2}(k-c)(k-c+2)-\frac{m^2}{8N^2}k(k+1)\right\}$$

for the perfect version.

Proof. The first two factors in the sums account for the amount $t - k + 1$ of computations done to rule out a false alarm and the probability to reach this step in the trade-off. In the non-perfect trade-off, the probability of a merge between columns $t - c$ and t is the same as with the Hellman trade-off: $\frac{mc}{2N}$. The probability to reach the step k was given in Proposition 2.7:

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

We denote with RM_t the set of elements of the kth column of the rainbow matrix. In the perfect case, we reach the kth step if all the previous steps failed which corresponds to the probability

$$\left(1-\frac{m}{N}\right)^{k-1},$$

since all the columns have the same number of distinct elements m. The probability that a false alarm is stopped by a checkpoint at position $t - c$ is given by

$$\frac{1}{2}\left(\frac{|F^{-k}(RM_t)| - |F^{-k-c}(RM_{t-c})|}{N}\right). \tag{3.16}$$

The first term of (3.16) is computed in Proposition 3.6, and the second term is given in Proposition 3.7. When the values are replaced in (3.16) and simplified, we obtain the claimed result. □

The generalisation to an arbitrary number of checkpoints for the rainbow trade-off is performed by Wang and Lin in [WL13]. However, Hong in [Hon16] brings the attention to the fact that their formula reduced to the single checkpoint case does not match the formula of [Hon10], corresponding to Theorem 3.8, from which they are derived. A later work of Kim, Seo, Hong, Park and Kim [KSH+12] provides a coherent generalisation in the special case of the perfect rainbow trade-off, which is given in Theorem 3.10.

Theorem 3.10 (Perfect rainbow table false alarms with multiple checkpoints)**.** *The expected number of applications of F induced by false alarms in a perfect rainbow table using n checkpoints at positions $t-c_1, t-c_2, \ldots, t-c_n$, is*

$$\frac{1}{N}\sum_{j=1}^{n}\left\{\sum_{c_j<k\leq c_{j+1}}\left(1-\frac{m}{N}\right)^{k-1}(t-k+1)\left((z_{c_j,k}-m)+\sum_{u=0}^{j-1}2^{u-j}(z_{c_u,k}-z_{c_{u+1},k})\right)\right\},$$

where $c_{n+1}=t$, and

$$\forall k<t, \forall i\geq 0, \quad z_{i,k}=\begin{cases} m(1+k-i)\left(1-\frac{mi}{4N}\right), & \text{if } i=0, \text{and} \\ m(k+1-i)+\frac{(k-i)(k-i+2)}{i^2}\left(mi+2N\ln\left(1-\frac{mi}{2N}\right)\right) & \text{otherwise.}\end{cases}$$

Proof. We denote the set of elements belonging to the kth column of the rainbow matrix by RM_k. We note for $i\geq 0$, and $k\leq t$

$$z_{i,k}=|F^{-k-i}(RM_{t-i})|.$$

The case $i>0$ corresponds exactly to Proposition 3.7, and the case $i=0$ corresponds to the amount of false alarms at column k, given by Proposition 3.6.

Consider a potential chain of length k, and $j<n$ such that $c_j<k\leq c_{j+1}$, which merges with a chain of the rainbow matrix. The merge happens before $t-c_j$ with probability

$$\frac{1}{N}|F^{-(k-c_j)}(RM_{t-c_j})\backslash RM_{t-k}|=\frac{1}{N}(z_{c_j,k}-m), \tag{3.17}$$

in which case the endpoints are useless. The probability of a false alarm due to the merge happening between $t-c_u$ and $t-c_{u+1}$ for $u<j$ is given by

$$\Pr(\text{merge between } t-c_u \text{ and } t-c_{u+1})=\frac{1}{N}(z_{c_u,k}-z_{c_{u+1},k}). \tag{3.18}$$

Such a false alarm is avoided with probability $\frac{1}{2^{j-u}}$. Then the probability accounting for all the checkpoints before $t-c_j$ is given by

$$\sum_{u<j}\Pr(\text{merge between } t-c_u \text{ and } t-c_{u+1}). \tag{3.19}$$

The probability of the last case, where the merge happens after $t-c_1$, is

$$\frac{1}{N}(z_{0,k}-z_{c_1,t}), \tag{3.20}$$

and the remaining checkpoints after c_{j+1} are avoided with probability $\frac{1}{2^j}$. Note that (3.20) is the degenerate case of (3.18) when $u = 0$, so the sum of the expression of (3.19) and the expression of (3.18) corresponds to (3.18) when starting the summation with $u = 0$.

Then the probability of a false alarm at column k is given by

$$\frac{1}{N}\left\{(z_{c_j,k} - m) + \sum_{u=0}^{j-1} 2^{u-j}(z_{c_u,k} - z_{c_{u+1},k})\right\}.$$

The number of applications of F induced by a false alarm that arises at column k is $t-k+1$. We reach this column in the trade-off with probability $\left(1 - \frac{m}{N}\right)^{k-1}$. A summation on all the columns between all the checkpoints leads to the result. □

3.3.3 Performance of the checkpoint technique

Naturally, the choice of the positions of the checkpoints influences the online time, and therefore the efficiency of the method. The later the checkpoints are on a chain, the greater the probability that they will be crossed by the verification chain. However, if a checkpoint is too far towards the end of the table, the probability that the verification chain merges before that checkpoint is greater. Inversely, if a checkpoint is too close to the starting point, it will not be used very often and a lot of calculations from the verification of all the columns after the one which contains that checkpoint will have to be done.

Another trade-off to consider is the number of checkpoints to use. Using several checkpoints ensures a repartition in the matrix that is guaranteed to be applied on a maximum of potential chains. However, each added checkpoint adds 1 bit of information per row, which can become significant, especially when said bits do not come “for free”, such as in a bitpacked implementation.

In [Hon10], Hong provides the memory cost of the checkpoints, *i.e.*, the cost induced by having tables entries weighting a few bit more in memory. The idea is to compute the equivalent online time of a table whose size is the same as the size of the table with the checkpoints, bu whose additional space is made of additional chains of the table. An example of such a gain is given in Table 3.1. This allows us to assess whether the checkpoints are indeed worth their impact on the memory. In order to be worthwhile, the reduction of online time by using the checkpoint has to exceed this cost gain.

Table 3.1 – Gain in online time by adding the equivalent of 1 bit per row of memory to store additional chains

size of row	gain
25 → 26	7.54%
30 → 31	6.35%
40 → 41	4.82%
50 → 51	3.88%
60 → 61	3.25%

The optimal position of checkpoints of both the Hellman trade-off and the rainbow trade-off are discussed in [Hon10]. We first mention the Hellman case. If we consider a checkpoint at position $t - c$, the value which maximises Theorem 3.8 is $\frac{c}{3}$. This means that for the approximately $\frac{1}{6}t$ that are caused by

the false alarms, about $\frac{1}{27}t$ iterations can be removed by the checkpoint. Using the observation leading to formulas in the like of (3.15), optimal positions when using a few checkpoints in the Hellman case are provided in [Hon10], and reproduced in Table 3.2. It is stated that a greater number of checkpoints

Table 3.2 – Optimal ratios $\frac{c}{t}$ for a single Hellman table using checkpoints at position $t - c$

1 checkpoint	0.33			
2 checkpoints	0.25	0.41		
3 checkpoints	0.21	0.33	0.48	
4 checkpoints	0.18	0.28	0.39	0.53

is unlikely to be beneficial to the Hellman trade-off. With the matrix stopping rule $mt^2 = N$, the respective non cumulative gains of using a 1st, 2nd, 3rd, and 4th checkpoint are of 3.17%, 2.03%, 1.43% and 1.06% respectively. Therefore, according to Table 3.1, the checkpoint technique is indeed not useful at this success rate, for the Hellman case. In [Hon10], the author observes that a benefit can be obtained at higher rates for the Hellman case, and that, more generally, the checkpoint technique is more suitable to high success rate trade-offs, and is especially powerful on variants which suffer more from false alarms, *i.e.*, for which the verification of false alarm requires a greater number of computations of F, such as the rainbow trade-off.

This lead us to consider the case of the rainbow trade-off. The optimal position of a single checkpoint c in a single rainbow table is also given in [Hon10], for various matrix stopping rules. Both the perfect and non-perfect trade-offs are considered, and the proportion of saved iterations of F on the total online phase is given. The results are gathered in Table 3.3. It can be seen that the rainbow trade-off does benefit a lot more from the checkpoints than the Hellman trade-off: even for matrix stopping rules corresponding to moderate success rates, the gain on the online time easily exceeds the gain obtained by using the checkpoint bit to store additional chains seen in Table 3.1, as long as the rows are not too small.

Table 3.3 – Impact of a single optimal checkpoint on a rainbow table for various success rates

perfect	mt/N	1	1.5	1.8	1.9	2
	c/t	0.74	0.8	0.83	0.84	0.86
	filtered	5.10%	6.55%	7.42%	7.52%	7.62%
non-perfect	mt/N	1	2	5	10	100
	c/t	0.70	0.77	0.885	0.89	0.91
	filtered	5.91%	9.57%	14.6%	17.3%	20.1%

For the perfect rainbow case, a set of optimal positions for the checkpoint is computed in [AJO08] in the more realistic context of $\ell = 4$ tables corresponding to a problem size $N = 8.06 \cdot 10^6$, and with parameters $t = 10^4$ and $m = 15.5 \cdot 10^6$. The authors of [AJO08] take the opposite approach when accounting for the memory penalty of the technique with regard to what is done in [Hon10]. Instead of computing the time penalty induced by the additional checkpoints in the table, they compute the gain in time that would be obtained with a new table of the size of the given table, including checkpoints. Note that, as opposed to the comparison of [Hon10], this approach does not work when considering the

case of maximum tables. The example in [AJO08] uses a ratio $\frac{mt}{N} \approx 1.91 < 2$, so their tables are not maximal.

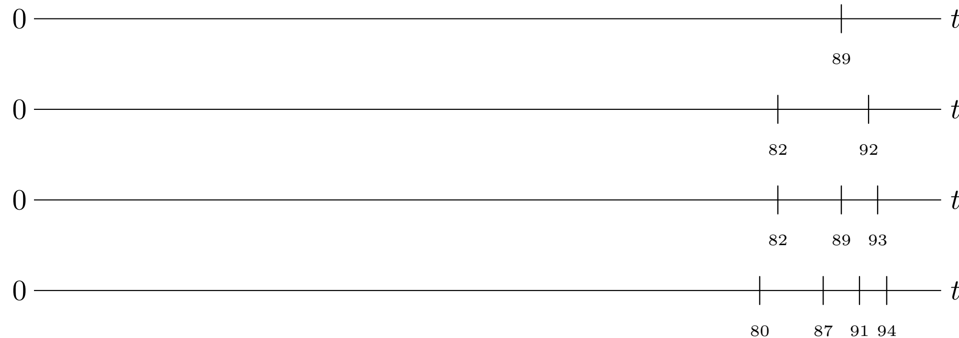


Figure 3.3 – Optimal checkpoint positions $\frac{c}{t}$ for a typical perfect rainbow trade-off (in %)

The optimal positions are computed by evaluating the online time for each position of the checkpoints and keeping the positions which give the minimum online time. Due to the large search space of the minimisation problem, not all values were tried and the results are to be considered with an error of $\pm 5 \cdot 10^{-4}$. We disregard the discrepancies noted by [Hon16] in this case, because the position of checkpoints is determined with the full online time which was stated fairly accurate in [Hon16], so we shall consider their results close to the reality. The results are given in $\frac{1}{t}$ units. These optimal positions are represented in Figure 3.3. One can observe that the checkpoints are all grouped at the rightmost part of the table, which is the average position of the answer in the case when $\ell = 4$. The gain obtained from the use of these checkpoints goes from 1.76% using 1 checkpoint, to 32% when 6 bits of checkpoints are used, which is a significant improvement.

In conclusion, the checkpoint technique has the potential to provide a reduction of the online time by avoiding a lot of work induced by false alarms. The technique itself requires additional steps during the pre-computation phase, and additional memory to store the checkpoints. This gain outweighs the cost due to the additional memory when the cost of a false alarm is high. This happens when the success rate is high, and false alarms are frequent. When this is the case, the use of several checkpoints provides a substantial improvement. In particular, the rainbow trade-off benefits greatly from this technique, even more so in its perfect version.

3.4 Fingerprints

The fingerprint method, presented by Avoine, Bourgeois and Carpent in [ABC15], takes the checkpoint approach a step further. It comes from a consideration on the nature of the endpoint in a TMTO. In a table, the endpoint serves two roles during the online phase:

- it allows at each column, *i.e.*, for each possible length of the potential chain, to determine if the potential chain is likely to be in the TMTO matrix by checking for the presence of the potential endpoint in the table, and
- it associates the above endpoint with its corresponding starting point, *i.e.*, that is it designates the right chain in the TMTO matrix.

The endpoint itself is therefore only the mean of a characterisation of the correct online chain. The technique of the truncated endpoint described in Section 3.2 shows that storing the whole endpoint is not necessarily the best approach, as using those truncated bits to store more chains is more advantageous. The checkpoint technique shows that, for certain settings, storing bits of information about the offline chain included in the TMTO matrix is more useful than using the same amount of memory to store other chains. In [ABC15], these two techniques are combined and extended into a framework which revolves around a new structure: the *fingerprint* of a chain, which allows an unified view of the aforementioned techniques.

The fingerprint technique is presented in [ABC15] in the context of the rainbow trade-off. Therefore, the subsequent descriptions of the modification of the online and offline phases are to be thought of as modifications on the rainbow trade-off phases, but, in theory, nothing prevents the technique from being applied to the Hellman trade-off.

3.4.1 Offline phase

Let us denote with $(\sigma_i)_{i\leq t}$ a set of positions in the matrix, as it was done for the checkpoints. We define a family of functions $(\Phi_i)_{i\leq t}$, analogous to the check functions of checkpoints, one for each column of the TMTO matrix, by

$$\Phi_i : \mathcal{N} \mapsto \begin{cases} \{0,1\}^{\sigma_i} & \textit{if } \sigma_i > 0 \\ \varepsilon & \textit{otherwise} \end{cases},$$

where ε represents no information.

Note that contrarily to the checkpoint method, the functions $\Phi_1, \ldots, \Phi_t$ can output an arbitrary amount of bits, which is of paramount importance since the endpoints are not stored anymore. Indeed, 1 bit per column may be insufficient to gather all the information that was previously provided by a full endpoint.

The offline phase algorithm of fingerprints reuses the same additional steps introduced by the checkpoint technique. At each column k of the offline chain creation, Φ_k is applied to the obtained element and stored. Note that in the case where it outputs ε, the application is essentially a non-operation. No special treatment is made on the endpoint, which is only considered as the element of the last column.

Fingerprints are then the concatenation of the outputs of the Φ_k on every element of the chains:

$$\forall i \in \{1, \ldots, m\}, \quad F_i = \Phi_1(x_{i,1})|\Phi_2(x_{i,2})|\cdots|\Phi_t(x_{i,t}),$$

where $|$ denotes the concatenation of elements, and (x_{ij}) is the TMTO matrix of the given table. The functions used in [ABC15], in the continuity of the G functions of [AJO08] are functions which output one or more of the least significant bits of a given element.

Note that in the case of the perfect trade-off, the full endpoints have to be stored in memory temporarily. The rows are then sorted according to the endpoints and those for which there is already an equivalent row with the same endpoint are discarded. This is due to the fact that sorted fingerprints do not allow such an easy pruning of duplicates.

3.4.2 Online phase

During the online phase, potential chains $(x_i)_{i\geq k}$ for $k \leq t$ are computed. In a chain of length k, for i in {0, …, k-1}, $\Phi_{t-k+i}(x_i)$ is computed. This gives a partial fingerprint of length k of the chain, missing $\Phi_1(x_1)|\ldots|\Phi_{k-1}(x_{k-1})$. As with the checkpoints, if $k < \sigma_1$, the missing part of the fingerprint contains

no information and the partial fingerprint is equivalent to the full version. This partial fingerprint is compared to the corresponding part of each fingerprint stored in the table. If there is a match, a verification chain is computed, otherwise the next potential chain is tried.

Since the fingerprints include checkpoints information, they reduce the amount of false alarm that have to be verified. However, they introduce an event which does not happen with classical TMTOs. If the partial fingerprint matches the corresponding part of a fingerprint in table, there is a possibility that the corresponding full fingerprint of the potential chain does not match the fingerprint in the table. This happens because only the full endpoint storage would be able to disambiguate such a situation. Afterwards, a verification has to be computed. The authors of [ABC15] call this event a *Type-II* false alarm, with the *Type-I* referring to the classical false alarms induced by the potential chains merging with chains in the table.

The authors give in [ABC15] an analysis of the average online time of a rainbow table using fingerprints, which is generic for a varying number of bits per checkpoints. The computation of the potential chains is identical to the one of the classic rainbow trade-off, We give the amount of work due to false alarms, extracted from the more complex formula of [ABC15], in Theorem 3.11. The full online time with fingerprint is obtained by including the work done for computing the potential chains W_k, such as defined in the general formula of the online phase of the rainbow trade-off in Section 2.4.

Theorem 3.11. *We consider a rainbow trade-off of parameters N, t, where m_i denotes the distinct elements at column i of the rainbow matrix. With the fingerprint method defined with bit positions $\sigma_1, \sigma_2, \dots, \sigma_t$, the amount of computations which are related to false alarms is given by*

$$Q_{fp} = \sum_{j=1}^{t\ell} \frac{m}{N}\left(1-\frac{m}{N}\right)^{j-1} \sum_{i=1}^{j} (t-c_i) \left[\frac{1}{N} \sum_{k=1}^{c_i} (z_{j-1,c_i} - z_{j,c_i}) \prod_{j=t-c_i+1}^{t-k} \phi_k + m\left(1 - \prod_{i=t-c+1}^{t}\left(1-\frac{m_i}{N}\right)\right) \prod_{k=t-c_i+1}^{t} \phi_k \right],$$

where $c_i = \left\lceil \frac{i-1}{\ell} \right\rceil$,

$$\phi_c = \begin{cases} \frac{1}{2^{\sigma_i}} & \text{if } \sigma_i > 0, \text{ or} \\ 0 & \text{otherwise,} \end{cases}$$

and

$$z_{i,c} = \begin{cases} \frac{(c-i)(c-i+2)}{i^2}\left[mi + 2N\log\left(1-\frac{mi}{2N}\right)\right] + m(1+c+i) & \text{if } i > 0, \text{ and} \\ m(1+c)\left(1-\frac{mc}{4N}\right) & \text{if } i = 0. \end{cases}$$

Proof. The average amount of computation is calculated with the summation of the expected amount per columns on the ℓt steps of the online phase algorithm. The quantity $c_i = \frac{\lceil i-1 \rceil}{\ell}$ associates the column of the table corresponding to the ith step of the algorithm. The rainbow trade-off probability of finding the answer at a given column follows a geometric law, so we have

$$Q_{fp} = \sum_{j=1}^{t\ell} \frac{m}{N}\left(1-\frac{m}{N}\right)^{j-1} (\text{cumulated cost of false alarms until step } k).$$

The length of a verification chain at step j, and therefore the cost of a false alarm, is $t - c_i$. Therefore,

we get the cumulated cost by

$$\sum_{i=1}^{j}(t-c_i)\Pr(\text{false alarm at step } j).$$

Let us then focus on the bracket expression in the theorem equation, which represents the probability of a false alarm at a given step i. It consists in the probability of false alarm of Type-I plus the probability of a Type-II false alarm minus the probability of a right answer. This last probability is neglected, *i.e.*, the true alarm is counted in the false ones.

The probability of a Type-I false alarm with fingerprints is obtained by tweaking the probability that was computed for Theorem 3.10 in Section 3.3, to accommodate for the fact that there can be multiple bits per checkpoint. The function z which account for the preimage size of a given column is reused here, with $\forall i \leq t, \forall c > i$, $z_{i,c}$ being the amount of chains which contain an element from the m_{t-c} at column $t-c$ that merge in column $t-i$. The probability of a merge between $t-c$ and $t-i$ is then

$$\Pr(\text{merge between } t-c \text{ and } t-i) = \frac{(z_{i-1,c}-z_{i,c})}{N}. \tag{3.21}$$

To have a false alarm, the partial fingerprint must also match the potential chain. This is given by the joint probability that all the ϕ_j are equal to the ones in the table, for all column until i:

$$\Pr(\text{partial fingerprint match between } t-c \text{ and } t-i) = \prod_{j=t-c}^{t-i}\phi_j. \tag{3.22}$$

Hence, the probability of Type-I false alarm is given by adding the probability

$$\Pr(\text{partial fingerprint match between } t-c \text{ and } t-i) \times \Pr(\text{merge between } t-c \text{ and } t-i)$$

to merge at each column between the start of the potential chain and the given column c. This is the first term within the brackets.

The probability of a merge from the current column until the last column is given by the false alarm formula already computed in Section 2.4:

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

Then there are $m-q_c$ non-merged chains at the last column. Also, the partial fingerprint has to match for all the columns until the endpoint, so the probability of a Type-II false alarm is given by

$$(m-q_c)\prod_{k=t-c_i+1}^{t}\phi_k.$$

This is the second term in the bracket. □

3.4.3 Optimal configuration

As with the checkpoint technique, the performance of the fingerprint technique is highly dependent on the choice of the $(\Phi_k)_{k\leq t}$, and in particular on the values of the σ_i. The authors of [ABC15] discuss the cost of the brute force approach, which would require even for a single checkpoint to evaluate $\log_2^t N$ possibilities, evaluate T for each of them, and keep the one that minimises T. This approach clearly becomes infeasible when $t \gg 1$.

The chosen approach is to tackle the minimisation problem with numerical analysis. To do so, the authors of [AJO08] chose to use a Hill climbing algorithm. Hill climbing designates a class of optimisation algorithms which start with a potential solution and try to generate perturbed states that are closer to the goal, *i.e.*, higher or lower, depending on whether this is a maximisation or a minimisation problem. When one is found, the solution is updated and the process is reiterated. The limitation of such algorithms is that they tend to find a local optimum, rather than a global one.

A few assumptions are made in order to use the Hill climbing method on the fingerprint optimisation. First, it is assumed that the functions Φ_k output exactly 1 bit for $k \in \{1, \ldots, t-1\}$, and Φ_t outputs several bits. The argument of the authors of [ABC15] in favor of this assumption is that they observed experimentally that good configurations tended to have the bit positions scattered, except for the last column. Moreover, given that $t \gg 1$, they state that having single bits in adjacent columns is nearly equivalent to output several bits in one of these columns. The observation of the authors also seems to indicate that with the previous assumption, the distribution of the online times T with regard to the bit positions is unimodal, *i.e.*, that if a minimum is found, it is global. A final observation is that even if the minimum found is not optimal, the optimal solution would yield even better results than their findings.

An experimental validation is performed in [ABC15] using an implementation of the perfect rainbow trade-off with maximum tables. In the context of maximum tables, the parameters ℓ, t, and m can be optimally derived from each other, and N, as shown in [AJO08], can be derived by exploiting the maximal probability of success given in Theorem 2.6. They present the gain of their technique compared to regular perfect rainbow tables without any improvement. For an example of fixed memory of $M = m\ell \log N =$ 8GB, and problem spaces from $N = 2^{40}$ to $N = 2^{46}$, they find that 9 positions $(\sigma_i)_{i\leq 9}$ are optimal. The gain in online time from using the fingerprints then ranges from 32.76% to 41.58%. In another comparison with a fixed problem size $N = 2^{48}$, they find that $M =$ 2GB, 4GB, and 8GB yield improvements of 46%, 45% and 44%, respectively.

The fingerprint technique, combining the concept of endpoint truncation, and checkpoints, is therefore clearly interesting, at least in the case of the perfect trade-off with maximum tables. In this specific case, the reduction in online time can be shown to be from a third to nearly half of the effort in terms of computations of F, compared to an unoptimised variant.

3.5 Dealing with non-uniform problem spaces

Every previously described TMTO deals with its problem space in a uniform manner, *i.e.*, every element of the space $\mathcal{N}$ is considered equally probable, and therefore has an equal chance of being found, given a specific amount of time. This makes sense when the problem space is a key space, as keys are supposed to be generated randomly. However, there are cases, *e.g.*, when dealing with password hashes, where elements in a problem spaces have different probability of being the answer we are looking for.

The goal when processing non-uniform problem spaces is to let the TMTO be aware of priorities in a way that it will perform more quickly on some subset of higher importance than with other elements.

Two approaches have been proposed in the literature in order to solve this problem. One of them consists in editing the reduction function to influence the path taken by the chains, and the other one is a technique called *interleaving* which splits the problem in sub-TMTOs of equal density.

3.5.1 Custom reduction function

The idea of modifying the reduction function is proposed by Hoch in [Hoc09]. By tweaking the reduction function, the elements of $\mathcal{H}$ are more frequently reduced to elements with higher priorities. However, this technique comes with the drawback of rendering the reduction function complex to the point that it does not perform anymore in a negligible time with regard to the one-way function. The slow down then affects both the offline and the online phases.

Another critical point regarding the reduction function is its need in memory. When the reduction function is not cache-friendly, *i.e.*, if it requires so much memory to operate that it does not fit into the CPU cache, the construction of chains during the online phase involves a memory access at each column, which would otherwise not happen with a simpler reduction function such as a permutation. These additional memory lookups are the main reason why an arbitrary dictionary is difficult to process with a TMTO. One can clearly see that, in the extreme case, the reduction function would have to access a data structure in the order of $O(N)$ to know to which element to reduce to, which takes the expensive memory resource needed by the table itself.

A variant of the idea using Markov chains is studied by Tissière, Oechslin and Lestringant in [TOL13] and by Narayanan and Shmatikov in [NS05]. Both works remark that the penalty performance of such a method is not negligible, to the point that the flexibility of the method does not make up to the performance penalty.

3.5.2 Interleaving

The second approach to influence the order of the online phase is the interleaving technique, introduced by Avoine, Carpent and Lauradoux in [ACL15]. The key idea is to partition $\mathcal{N}$ in equiprobable subsets, in the sense that the answer on the inversion problem is likely to be found in any of them at random with the same probability. The authors of [ACL15] use the example of passwords with and without special characters.

If $\mathcal{N}$ is the set of all possible passwords of a size up to, say, 8 characters, let $\mathcal{N}_1$ be the subset containing only the password with alphanumeric characters, and $\mathcal{N}_2$ containing the password with the special characters, *i.e.*, $\mathcal{N} = \mathcal{N}_1 \cup \mathcal{N}_2$. There are about 33 special characters, which means that $\mathcal{N}$ is overwhelmingly constituted of $\mathcal{N}_2$. Indeed, $|\mathcal{N}_2| \approx 95^8$, $|\mathcal{N}_1| \approx 62^8$, and $\frac{|\mathcal{N}_1|}{|\mathcal{N}|} \approx 3\%$. Nevertheless, according to online password database leaks, such as Rock You [Cub09], the amount of people that use special characters is a minority. We can see intuitively that a TMTO which deals uniformly with such a space will spend a lot of time on a space way larger than necessary for a lot of problems. The interleaving technique does apply specifically to this context where the problem space can be divided in several subsets of very different densities.

The problem space $\mathcal{N}$ (of size N) is divided in n subsets $\mathcal{N}_1, \mathcal{N}_2, \dots, \mathcal{N}_n$ of respective sizes $N_1, N_2, \dots, N_n$. For the interleaving to work, we have to determine an integer partition

$$\rho_1, \rho_2, \dots, \rho_n \quad \text{s.t.} \quad \sum_{i=1}^{n} \rho_i = 1,$$

with which we can weigh the subsets.

Assuming the memory of the initial TMTO is M, the idea is to construct n smaller TMTOs (sub-TMTOs) with both the same probability of success P and number ℓ of tables as the initial TMTO, using a memory $m_i = \rho_k M$, on a problem space $\mathcal{N}_k$, for $k \in \{1, \ldots, n\}$. Note that in this case, the length of the tables t_k for each sub-TMTO is determined by the other constraints that are M, N and P.

The authors of [ACL15] present an application of the technique applied on the perfect rainbow trade-off. While the global method is independent of the trade-off, the formulas presented below are specific to the perfect rainbow trade-off.

A first remark is that the probability of success of the set of sub-TMTO is the same as the probability of success of the original TMTO. This result from [ABC15] is gathered in Proposition 3.12.

3

Proposition 3.12. *The success rate of an interleaved perfect rainbow trade-off of maximal size is the same as the one of the set of its sub-TMTOs. It is given by*

$$P^* = 1 - e^{-2\ell}.$$

Proof. For each of the n sub-TMTO, we denote with P_i^* the corresponding success rate, for $i \in \{1, \ldots, n\}$. In the case of the perfect rainbow table with maximum tables, if $t \gg 1$, the probability of success is only dependent on ℓ:

$$\forall i < n, \quad P_i^* = 1 - e^{-2\ell}.$$

Since we have $\sum_{b=1}^{n} \rho_i = 1$, the result follows. □

The same applies to the pre-computation effort which is linear in M. That is to say that the interleaving technique covers the same amount of elements on the total surface of its sub-TMTOs pre-computation matrices.

For each sub-TMTO b where $1 \leq b \leq n$, the average number of evaluations of the one-way function f at the kth column of any table, starting at the end of the table, is denoted by $[C_k]_b$. For a perfect table of chain length t, we have

$$C_k = k + (t - k + 1) q_{t-k+1}, \tag{3.23}$$

with

$$q_i = 1 - \frac{i(i-1)}{t(t+1)}$$

being the probability of a false alarm at column i.

Order of visit If the sub-TMTOs were to be processed sequentially, it would defeat the purpose of giving an equal chance for each sub-TMTO to yield the answer. Since the sub-TMTOs are of different size, the order of visit of the different sub-TMTOs has to be adapted. In [ACL15], the authors discussed the problem and introduced a function which outputs the best next candidate at each step.

The order of visit aims at minimising the online phase running time, by dynamically determining the optimal path to follow. The overhead of the choice itself must be negligible with regard to the search time. The authors of [ACL15] suggest using a metric based on the probability to find the solution in the next step on a given table, and the work needed in said step. The proposed metric is given by

$$\eta(k, b) = \frac{\Pr(\text{preimage found at step } k \text{ in sub-TMTO } l)}{\mathbb{E}[\text{work for the step } k \text{ of sub-TMTO } l]}.$$

At each step k, the table l which maximises $\eta(k,l)$ should be chosen for the step $k+1$.

In fact, in the case of the perfect rainbow trade-off, the probability to find an answer at each column is $\frac{m}{N}$ and is therefore independent of the column. As a result, the most efficient way to browse each sub-TMTO taken in isolation is the classical way introduced in Section 2.4. The order of visit therefore interleaves the classical way of browsing tables with several TMTOs, hence the name of the technique. An explicit formula is given in [ACL15] for the metric:

$$\eta(k,b) = \frac{\rho_b \left(1 - \left(1 - \frac{m_b}{N_b}\right)^{\ell}\right) \left(1 - \frac{m_b}{N_b}\right)^{(k-1)\ell}}{\ell \left[C_k\right]_b}.$$

Online time The authors of [ACL15] also provide an analytical formula of the average number of calls to the one-way function f during the online phase of the perfect rainbow trade-off with the interleaving technique which is recalled in Theorem 3.13.

Theorem 3.13. *Let the order of visit be denoted with*

$$V = (V_1 V_2 \dots V_{t^*}), \quad \textit{with} \quad t^* = \sum_{i=1}^{n} t_i.$$

The average online time of a successful search for the interleaved rainbow trade-off with maximum tables is given by

$$T_{int} = \sum_{k=1}^{t^*} \rho_{V_k} \left[1 - \left(1 - \frac{m_{V_k}}{N_{V_k}}\right)^{\ell}\right] \left(1 - \frac{m_{V_k}}{N_{V_k}}\right)^{(S_k-1)\ell} \sum_{i=1}^{k} \ell[C_{S_i}]_{V_i},$$

where

$$S_k = |\left\{V_i \in \{V_1 \dots V_k\} \,|V_i = V_k\right\}|$$

is the number of steps already performed in the sub-TMTO chosen at step k. The expected online time if the search is unsuccessful is given by

$$\left[\sum_{b=1}^{n} \rho_b \left(1 - \frac{m_b}{N_b}\right)^{\ell t_b}\right] \sum_{b=1}^{n} \sum_{s=1}^{t_b} \ell[C_s]_b.$$

Proof. We begin with the successful search. The average time is decomposed in the probability to stop the search at step k coupled with the amount of work until this step k. Let us consider a step k. The sub-TMTO chosen at this step is V_k. The step of search of this particular sub-TMTO taken in isolation is $S_k - 1$. The answer is in this particular sub-TMTO with probability ρ_{V_k}, and in one of the ℓ columns visited at this step with probability

$$1 - \left(1 - \frac{m_{V_k}}{N_{V_k}}\right)^{\ell} \tag{3.24}$$

The answer must also not have been found in any of the previous visits of this TMTO, which corresponds to the probability

$$\left(1 - \frac{m_{V_k}}{N_{V_k}}\right)^{(S_k-1)\ell}. \tag{3.25}$$

The probability to stop at search k is then the product of ρ_{V_k}, (3.24), and (3.25).

The cost up to step k is the sum of the costs at all the steps $i < k$. At step i, we are at the step S_i of the sub-TMTO V_i, the cost of which is given in (3.23) by C_{S_i} for each table. The cumulated cost is given by

$$\sum_{i=1}^{k} \ell[C_{S_i}]_{V_i}.$$

We now deal with the unsuccessful search. The probability that a given element is not in a given sub-TMTO is the weighted probability (by the ρ_i) that it is not in any column of any table in said sub-TMTO which is

$$\rho_b \left(1 - \frac{m_b}{N_b}\right)^{\ell t_b}.$$

We sum on all the sub-TMTOs to get the total probability.

The cost of a failure is the sum on all TMTOs of the cost of each sub-TMTO which was given in the successful search case, except that it is independent from the order of visit, so we simply sum on $b \in \{1, \ldots, n\}$. □

Memory allocation Given the online runtime formula, the optimal partition of M can be determined with regard to the proportion ρ_b for each sub-TMTO b. It is done by solving a minimisation problem. The sequence of $\rho_1, \ldots, \rho_n$ which minimises the online time T, given by the online time formula, is selected and considered for the pre-computation phases. The authors of [ACL15] state that the technique works best when densities $\frac{\rho_i}{N_i}$ are very different from each other. Nevertheless, for each additional subset, an overhead incurs, which indicates that a division in a few subsets is ideal. Finally, the authors remark that with a distribution that is very skewed, one could even decrease the running time cost below theoretical bounds of the classic perfect rainbow trade-off.

3.6 Heterogeneous tables

We now describe a technique which shares some concepts from the interleaving technique which was mentioned above, applied to uniform spaces. Recall that, in the perfect rainbow trade-off, a single table cannot reach a full coverage by itself. That is why several tables are computed. In the different trade-offs variants, the tables are similar, with the same t (or $\hat{t}$) and m parameters, the difference between them residing in the fact that they use a different set of reduction functions. In [AC17], Avoine and Carpent explore the consequences of using *heterogeneous* tables of different chain lengths, instead of *homogeneous* tables with a fixed chain length, as in the classical scheme. Note that, in this technique, each table retains a fixed chain length, unlike DP tables. The idea of the resulting technique is to design what is called an *order of visit* of the tables in [AC17], which allows for a better average time.

The authors of [AC17] present and analyse the heterogeneous technique on the perfect rainbow table scheme. They argue that while the technique may be applicable to other TMTO variants, it might have less impact on, *e.g.*, the Hellman trade-off, since it relies on the difference between a successful and an unsuccessful search. Indeed, in the Hellman trade-off and in the non-perfect trade-off, the answer is not actually more likely to be at the end of the table, so changing the order of visit is actually not more likely to be useful.

Offline phase There is an additional step in the offline phase process to determine the parameters of the individual tables. Like in the interleaving technique, the idea is to build an equivalent TMTO in terms of success rate. This is done by partitioning the cumulative covered set of $\ell t \times \ell m$ elements of $\mathcal{N}$ into several tables covering the same part of $\mathcal{N}$, but with different sizes. For each computed table i, where $1 \leq i \leq \ell$, the amount of rows and the length of the chains are denoted hereafter by $[m]_k$ and $[t]_k$ respectively.

3.6.1 Analysis

The average number of evaluations of the one-way function f at the column k of the lth table, starting at the end of the tables, is denoted by $[C_k]_l$. For a perfect table l with chains of length t,

$$[C_k]_l = k + (t - k + 1)q_{t-k+1},$$

with q_i denoting the probability of a false alarm at column i.

The authors of [AC17] reuse the concept of order of visit introduced in [ACL15]. This time it is used to define the order in which the tables, instead of the sub-TMTOs, are browsed. Therefore, similar notations are used including the total length of the concatenated tables $t^* = \sum_{i=1}^{\ell} [t]_i$, and the sequence of visit $(V_k)_{0<k\leq t^*}$.

The order of visit is determined in the same way as in [AC17], by using a metric which computes the ratio of the probability to find the preimage in a given column $\frac{[m]_l}{N}$ (which is independent from the column number in the case of the perfect rainbow trade-off), over the work $[C_i]_k$ needed to verify the answer on said column. It is then determined by maximising the metric for the given column. We denote the metric for the lth table at the kth column with

$$\eta(k, l) = \frac{[m]_l}{N[C_k]l}.$$

During the online phase, after each column k, the value $\eta(k+1, i)$ is computed for $1 \leq i \leq \ell$, the highest of which determines the next table. This gives, in [AC17], the expected online time of the technique, which is recalled in Theorem 3.14.

Theorem 3.14. *The online time expected in number of computations of the one-way function f for the heterogeneous rainbow trade-off is given by*

$$T_{het} = \sum_{k=1}^{t^*} \left[\frac{[m]_{V_k}}{N} \prod_{j=1}^{k-1} \left(1 - \frac{[m]_{V_j}}{N}\right) \sum_{j=1}^{k} [C_{S_j}]_{V_j} \right] + e^{-2\ell} \sum_{i=1}^{\ell} \sum_{s=1}^{[t]_i} [C_s]_i,$$

with

$$S_k = \#\{i \leq k | V_i = V_k\},$$

being the number of steps done specifically in the table V_k after the kth step overall.

Proof. The expression is a generalisation of the classical online running time, when we consider a single table with columns of arbitrary size. The probability to stop at step k is the probability that an element is in the current column $\frac{m_{V_k}}{N}$, and the probability that it was not in any of the preceding columns

$$\prod_{j=1}^{k-1} \left(1 - \frac{[m]_{V_j}}{N}\right).$$

The work done in all the columns until the current one is given by

$$\sum_{j=1}^{k}[C_{S_j}]_{V_j}. \tag{3.26}$$

The probability of failure is $1 - P^* = e^{-2\ell}$, and the total work in this case is given by extending the expression in (3.26) to all the columns, on all the tables. It is given by

$$\sum_{k=1}^{t^*}\frac{[m]_{V_k}}{N}\prod_{j=1}^{k-1}\left(1-\frac{[m]_{V_j}}{N}\right)\sum_{j=1}^{k}[C_{S_j}]_{V_j},$$

for the case an answer is found. When the table fails to return an answer, all the $[t]_i$ columns of the ℓ tables have to be browsed, which corresponds to

$$\sum_{i=1}^{\ell}\sum_{s=1}^{[t]_i}[C_s]_i$$

computations. □

Now that the online time formula of the technique is available, one can deduce optimal parameters for the size of the tables.

3.6.2 Optimal tables parameters

The selection of the right partition when dividing the resources of the TMTO is done with the help of the running time formula. Since the problem size N is fixed, it suffices to find either of the sets $\{[t]_1, \ldots, [t]_\ell\}$ or $\{[m]_1, \ldots, [m]_\ell\}$ that is optimal. In [AC17], the problem of optimisation is presented with regard to the $[t]_i$. The problem is to find which combination of $[t]_i$ both minimises T, and is such that

$$\sum_{i=1}^{\ell}[m]_i < \ell m.$$

The technique is experimented in [AC17] on a problem of size $N = 2^{40}$, with $t = 10^4$ and a commonly used number of table $\ell = 4$. The tables used are nearly of maximal size. The optimisation gives a set of tables of relative size of approximately $\frac{1}{2}t$, t, $\frac{3}{2}t$ and $2t$, with corresponding inverse heights such that the surface $[t]_i[m]_i$ of each table i remains the same as the surface mt of its homogeneous counterparts.

3.6.3 Performance

The same set of parameters is used with a varying number of tables and t. Results are that the more tables are used, the better the technique performs, with a relatively stable gain with regard to t. The results from [AC17] are summarised in Table 3.4.

Note that the technique works better for a greater number of tables, *i.e.*, for a probability of success very close to 100%. While the average time performance is significant, the technique suffers from the drawback that the worst case takes more time than in the same trade-off with homogeneous tables. In the example of $\ell = 4$ tables, it takes more than twice as much time as the worst case of the regular rainbow trade-off.

Table 3.4 – Performance of the heterogeneous tables compared to the regular rainbow scheme

N	t	# of tables	T gain (%)
2^{40}	10^4	3	25%
		4	40%
		5	55%
		6	70%

3.7 Rainbow tables on GPU

We now present an overview of a topic that recently got some attention in various parts of computer science. The usage of GPUs to perform the rainbow trade-off in particular has been studied in the literature, in works such as [KPPM12; KSH+12], or [LLH15]. The online phase of the rainbow method consists in computing a lot of independent chains, which is a task that can benefit greatly from parallelisation. GPUs nowadays permit the use of their highly parallel architecture to perform parallel tasks very efficiently. Due to the fact that GPU architecture is different from classical CPU, some modifications of the trade-off are needed. In this section we discuss the use of a GPU to improve the online time of the rainbow trade-off.

3.7.1 GPU paradigm

While GPUs provide a high number of cores, they are much simpler than classical CPU core. The memory associated with each core is also very limited. A GPU typically possesses thousands of streaming processors, which are able to perform computations in parallel. This high parallelisation follows a hierarchical structure. Recall that these processors are historically designed to perform operations on a fraction of an image. Typically, an operation has to be applied on every pixel of an image. The image is divided in subgroups of pixels and each processor applies the same transformation on each of its group of pixels. One may think of such a transformation as a function of the position and the value of a given pixel.

A GPU is divided in *blocks*, which are themselves divided in *threads*. While the parallelisation is performed at the thread level, in practice, the problem can only be divided at the block level, meaning operations within a block have to be of a similar nature. Similar concerns apply with regard to the scheduling, as the smallest unit of synchronisation is a set of 32 threads called a *warp*.

An important aspect which differs from a classical CPU is the memory access. While all threads share a common memory, if one is to maximise the throughput of a GPU computation, each core can only rely on its assigned memory. Furthermore, the access to the main RAM memory from the GPU is relatively expensive. This implies that there is a large subclass of problems which perform poorly on a GPU, such as random access on large sections of memory.

3.7.2 Rainbow method on GPU

In order to give an overview of the challenges involved, we present hereafter an implementation from Kim, Seo, Hong, Park and Kim in [KSH+12], which applies to the rainbow trade-off on GPU, and on a hybrid GPU+CPU model. In their work, Kim et al. include the checkpoints technique from Section 3.3 into the rainbow trade-off, and discuss the architectural adjustments that are needed because of it. In fact, the implementation only partially relies on the GPU, as the tables are still stored in RAM and

the CPU still performs the scheduling, as well as a varying part of the computations. This model is sometimes called *hybrid*.

A first idea to implement the parallel version of the online phase is to make the ith thread compute the chain at column $t - i$, *i.e.*, the chain of length i. The same thread is tasked with verifying whether the endpoint exists in the table, and then, would it be the case, to compute the verification chain. There are fewer chains to compute during the verification than during the initial computation of the potential endpoints. Nevertheless, the verification takes more time due to the granularity of the scheduling. Indeed, within a warp, only some of the threads are tasked with a verification, and the other cannot be put to use before these finish, thus wasting computing time.

In order to solve the problem of the unused threads, another method is to split the work between the lookup procedure, in which the chains are extended until a potential endpoint, and the verification. In this improved method, the work is split between the GPU which handles the extension, and the CPU which does the verification. In practice, each thread does the extension itself, then checks whether the potential endpoint is in the table. Then if the endpoint does belong to the table, the chain is copied into RAM (*i.e.*, CPU memory), where the CPU can build the chain from the corresponding starting point to verify whether the solution is found. This hybrid implementation does eliminate the waste incurred by the warp serialisation, as the computations on the GPU are deterministic. Nevertheless, while the GPU computations are efficient, the verification done by the CPU does take a lot longer than the GPU part, and becomes the major part of the time spent during the online phase.

In order to further reduce the time taken by the algorithm, the authors of [KSH+12] apply the checkpoint technique to the online phase. The improved algorithm introduced above has to be slightly modified. The checkpoints are loaded into the general memory of the GPU, *i.e.*, the memory shared by the threads of the GPU. An experiment with 22 checkpoints is performed in [KSH+12], on a table of parameters $N = 2^{41.7}$ and $m = 80530636$. The checkpoints in the experiment, allow for a reduction of the amount of calculation due to false alarms by 84%. The result is that the total time of the algorithm is reduced to roughly the time it takes to build only the chains. The key point of this technique is that the verification part has been reduced, with the help of the checkpoints, to the point that it takes less time than the construction of the chains. Note that the number of checkpoints needed to achieve equality in time by both the part done on CPU, and the one done on GPU, seems quite high with regard to the fact that it needs an additional 22 bits to be stored alongside the chains. The checkpoints therefore amount for a sizeable part of the resulting storage, and it may be argued that, for this reason, this algorithm is probably not the best approach.

The question remains to determine whether the hybrid approach is relevant, since the load on the CPU is too heavy without resorting to the use of a lot of checkpoints. Figuring a way to perform some of the verification without stalling the threads in a warp may be the key to achieve such an improvement.

3.8 Conclusion

We reviewed in this chapter the most significant improvements on cryptanalytic TMTOs that were introduced in the literature. We saw that the objective of such improvements was to exploit the resources available to the implementer of a given TMTO, in such a way that it allowed either to reduce the online running time of the algorithm, or to store the table with less memory. These improvements gave us a basis to choose from when seeking the best performance from the TMTO technique, which is the work tackled in Chapter 6.

We discussed that while the prefix-suffix technique, which was commonly used in implementations,

allowed a significant gain in space, it was not optimal. Moreover, it was possible to bring the size of a given table close to the theoretical lower bound for assumptions that are known today. By exploiting the apparent distribution of the endpoints and by encoding the differences between the endpoints, it was shown that the size of the table was near optimal, without sacrificing its usability table by the online algorithm. It was also observed that truncating a few bits of the endpoints was in fact beneficial to the online time if the saved space was used for additional chains.

The checkpoint technique, extended the concept that the classic structure of the row in a table was not optimal in the sense that keeping information on other part of the chain than the endpoint allowed to reduce the work induced by false alarm in high success rate TMTOs. The fingerprint technique, pushing the idea to the extreme, proved that the truncation and checkpoints techniques combined nicely in an even more performant trade-off. The idea of an endpoint-less table presented in this chapter can serve as a comparison point to the alternative technique we will introduce in Chapter 5, which does not need to store an endpoint either.

The traditional assumption that the problem space is uniform was challenged. It was seen that when this is not the case, it was possible to lower the average online running time, by modifying the order of visit of several sub-TMTO of equal probability, but different problem spaces. This reduction came at the expense of a worse worst case running time. The seemingly independent idea of using differently shaped tables within a same TMTO turned out to also benefit from a similar method of order of visit rearrangement, although with the same drawback.

Finally, to complete our overview on the TMTO improvements, a mention was made of the GPU case and the challenges it involves when one wishes to harness the power of a parallel workload to improve the online phase of a rainbow trade-off. In particular, we highlighted that the costly link between host and GPU memory tends to bottleneck the online procedure. A methodology involving checkpoints and a repartition of the workload between GPU and CPU in a hybrid approach was presented in order to overcome this limitation.

References

[ABC15] Gildas Avoine, Adrien Bourgeois, and Xavier Carpent. *Analysis of Rainbow Tables with Fingerprints*. In: *ACISP 15*. Ed. by Ernest Foo and Douglas Stebila. Vol. 9144. LNCS. Brisbane, QLD, Australia: Springer, Heidelberg, Germany, June 2015, pp. 356–374 (cit. on pp. 73–75, 77, 79, 162, 163).

[AC14] Gildas Avoine and Xavier Carpent. *Optimal Storage for Rainbow Tables*. In: *ICISC 13*. Ed. by Hyang-Sook Lee and Dong-Guk Han. Vol. 8565. LNCS. Seoul, Korea: Springer, Heidelberg, Germany, Nov. 2014, pp. 144–157 (cit. on pp. 56–62, 161).

[AC17] Gildas Avoine and Xavier Carpent. *Heterogeneous Rainbow Table Widths Provide Faster Cryptanalyses*. In: *ASIACCS 17*. Ed. by Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi. Abu Dhabi, United Arab Emirates: ACM Press, Apr. 2017, pp. 815–822 (cit. on pp. 20, 81–83).

[ACL15] Gildas Avoine, Xavier Carpent, and Cédric Lauradoux. *Interleaving Cryptanalytic Time-Memory Trade-Offs on Non-uniform Distributions*. In: *ESORICS 2015 , Part I*. Vol. 9326. LNCS. Vienna, Austria: Springer, Heidelberg, Germany, Sept. 2015, pp. 165–184 (cit. on pp. 20, 78–82).

[AJO05] Gildas Avoine, Pascal Junod, and Philippe Oechslin. *Time-Memory Trade-Offs: False Alarm Detection Using Checkpoints*. In: *Progress in Cryptology - INDOCRYPT 2005*. Ed. by Subhamoy Maitra, C. E. Veni Madhavan, and Ramarathnam Venkatesan. Vol. 3797. LNCS. Bangalore, India: Springer, Heidelberg, Germany, Dec. 2005, pp. 183–196 (cit. on pp. 63, 64, 154).

[AJO08] Gildas Avoine, Pascal Junod, and Philippe Oechslin. "Characterization and Improvement of Time-Memory Trade-Off Based on Perfect Tables". In: *ACM Transactions on Information and System Security* 11.4 (4 July 2008), 17:1–17:22. ISSN: 1094-9224 (cit. on pp. 16, 27, 30, 37, 56, 64, 72–74, 77, 154, 156, 157, 160).

[BBS06] Elad Barkan, Eli Biham, and Adi Shamir. *Rigorous Bounds on Cryptanalytic Time/Memory Tradeoffs*. In: *CRYPTO 2006*. Ed. by Cynthia Dwork. Vol. 4117. LNCS. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 2006, pp. 1–21 (cit. on pp. 35, 40, 58, 153, 155–157).

[BSW00] Alex Biryukov, Adi Shamir, and David Wagner. *Real Time Cryptanalysis of A5/1 on a PC*. In: *Fast Software Encryption – FSE'00*. Ed. by Bruce Schneier. Vol. 1978. Lecture Notes in Computer Science. New York, USA: Springer, Apr. 2000, pp. 1–18 (cit. on pp. 5, 56, 62, 63).

[Cub09] Nik Cubrilovic. *RockYou Hack: From Bad To Worse*. Accessed: Apr 2017. 2009. Link. (Cit. on p. 78).

[GKP89] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete mathematics: a foundation for computer science*. In: vol. 3. 5. AIP, 1989. Chap. 5, pp. 175–178 (cit. on p. 59).

[Gol66] Solomon Golomb. "Run-length encodings (Corresp.)" In: *IEEE Transactions on Information Theory* 12.3 (1966), pp. 399–401 (cit. on p. 59).

[GV75] Robert G. Gallager and David C. van Voorhis. "Optimal source codes for geometrically distributed integer alphabets (Corresp.)" In: *IEEE Transactions on Information Theory* 21.2 (Mar. 1975), pp. 228–230. ISSN: 0018-9448 (cit. on p. 60).

[Hel80] Martin Hellman. "A Cryptanalytic Time-Memory Trade Off". In: *IEEE Transactions on Information Theory* IT-26.4 (July 1980), pp. 401–406 (cit. on pp. 24, 25, 29, 30, 42, 45, 55, 153, 154, 184).

[HM13] Jin Hong and Sunghwan Moon. "A Comparison of Cryptanalytic Tradeoff Algorithms". In: *Journal of Cryptology* 26.4 (Oct. 2013), pp. 559–637 (cit. on pp. 16, 20, 30, 31, 33, 34, 44, 63, 157, 158, 160).

[Hoc09] Yaacov Zvi Hoch. *Security Analysis of Generic Iterated Hash Functions*. PhD thesis. Rehovot, Israel: Weizmann Institute of Science, Aug. 2009 (cit. on p. 78).

[Hon10] Jin Hong. "The cost of false alarms in Hellman and rainbow tradeoffs". In: *Designs, Codes and Cryptography* 57.3 (Dec. 2010), pp. 293–327 (cit. on pp. 27, 29, 40, 63–65, 67, 69–72, 166).

[Hon16] Jin Hong. "Perfect Rainbow Tradeoff with Checkpoints Revisited". In: *PLOS ONE* 11.11 (Nov. 2016), pp. 1–18 (cit. on pp. 64, 70, 73).

[Kie04] Aaron Kiely. *Selecting the Golomb Parameter in Rice Coding*. Tech. rep. 42-159. Jet Propulsion Laboratory, Pasadena, California, JPL Publication: The Interplanetary Network Progress Report, Nov. 2004, p. 18 (cit. on pp. 60, 61).

3

[Knu85] Donald E. Knuth. “Dynamic huffman coding”. In: *Journal of Algorithms* 6.2 (1985), pp. 163–180. ISSN: 0196-6774 (cit. on p. 56).

[KPPM12] M. Kalenderi, D. Pnevmatikatos, I. Papaefstathiou, and C. Manifavas. *Breaking the GSM A5/1 cryptography algorithm with rainbow tables and high-end FPGAS*. In: *22nd International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Aug. 2012, pp. 747–753 (cit. on p. 84).

[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: *Progress in Cryptology - INDOCRYPT 2012*. Ed. by Steven D. Galbraith and Mridul Nandi. Vol. 7668. LNCS. Kolkata, India: Springer, Heidelberg, Germany, Dec. 2012, pp. 303–316 (cit. on pp. 70, 84, 85, 187).

[LLH15] Jiqiang Lu, Zhen Li, and Matt Henricksen. *Time-Memory Trade-Off Attack on the GSM A5/1 Stream Cipher Using Commodity GPGPU - (Extended Abstract)*. In: *ACNS 15*. Ed. by Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis. Vol. 9092. LNCS. New York, NY, USA: Springer, Heidelberg, Germany, June 2015, pp. 350–369 (cit. on pp. 5, 42, 84).

[NS05] Arvind Narayanan and Vitaly Shmatikov. *Fast Dictionary Attacks on Passwords Using Time-Space Tradeoff*. In: *ACM CCS 05*. Ed. by Vijayalakshmi Atluri, Catherine Meadows, and Ari Juels. Alexandria, Virginia, USA: ACM Press, Nov. 2005, pp. 364–372 (cit. on p. 78).

[Oec03] Philippe Oechslin. *Making a Faster Cryptanalytic Time-Memory Trade-Off*. In: *CRYPTO 2003*. Ed. by Dan Boneh. Vol. 2729. LNCS. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 2003, pp. 617–630 (cit. on pp. 5, 35, 36, 56, 107, 153, 156, 157).

[Ric79] Robert F. Rice. *Some practical universal noiseless coding techniques*. Tech. rep. NASA-CR-158515. Jet Propulsion Laboratory, Pasadena, California, JPL Publication, Mar. 1979, p. 132 (cit. on p. 60).

[TO05] Cedric Tissières and Philippe Oechslin. *Ophcrack*. Accessed: Apr 2017. 2005. Link. (Cit. on pp. 7, 35, 56, 93, 99, 162).

[TOL13] Cedric Tissieres, Philippe Oechslin, and Pierre Lestringant. *Limites des tables Rainbow et comment les dépasser en utilisant des méthodes probabilistes optimisées*. In: *Actes du 11ème symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*. 2013 (cit. on p. 78).

[WL13] Wenhao Wang and Dongdai Lin. *Analysis of Multiple Checkpoints in Non-perfect and Perfect Rainbow Tradeoff Revisited*. In: *ICICS 13*. Ed. by Sihan Qing, Jianying Zhou, and Dongmei Liu. Vol. 8233. LNCS. Beijing, China: Springer, Heidelberg, Germany, Nov. 2013, pp. 288–301 (cit. on p. 70).

[ZL77] Jacob Ziv and Abraham Lempel. “A universal algorithm for sequential data compression”. In: *IEEE Transactions on Information Theory* 23.3 (1977), pp. 337–343 (cit. on p. 56).

The difference in time between modern CPU and disk technologies is analogous to the difference in speed in sharpening a pencil using a sharpener on one's desk or by taking an airplane to the other side of the world and using a sharpener on someone else's desk.

Douglas Comer

4 Rainbow tables with external memory

What happens when we perform a TMTO with tables stored on disk instead of RAM? We provide, in this chapter, an analysis of TMTOs in the external memory model, with an emphasis on the algorithmic variants that are used in practice by the developers of TMTOs.

Contents

4.1 Motivation . . . **92**
4.1.1 Context . . . 92
4.1.2 Towards larger TMTOs . . . 93
4.2 Models . . . **94**
4.2.1 RAM model . . . 94
4.2.2 External memory model . . . 95
4.3 Related work on TMTOs in external memory . . . **97**
4.3.1 Overview of existing studies . . . 97
4.3.2 Algorithms studied . . . 98
4.4 Performance of the algorithms . . . **100**
4.4.1 Context and notations . . . 100
4.4.2 Direct lookup algorithm . . . 102
4.4.3 Analysis of the algorithm used in practice . . . 103
4.4.4 Optimal configuration . . . 105
4.5 Establishing the constants . . . **106**
4.5.1 Experimental setup . . . 106
4.5.2 Computation time . . . 107
4.5.3 Single block read time . . . 107
4.5.4 Sequential block read time . . . 108
4.6 Analysis . . . **109**
4.6.1 Problem parameters . . . 109
4.6.2 Comparing the two algorithms . . . 110
4.6.3 Best algorithm . . . 113
4.6.4 HDD and SSD . . . 114
4.6.5 Discussion . . . 115
4.7 Experimental results . . . **117**
4.7.1 Parameters and methodology . . . 117
4.7.2 Paging and caching mechanisms . . . 118
4.7.3 Reducing the caching impact . . . 119
4.7.4 Results . . . 119
4.8 Conclusion . . . **121**

4.1 Motivation

A common assumption in the literature, which we made in Chapter 2, and that all the techniques reviewed in Chapter 3 share, is that the tables fit in a memory that is fast enough that the table lookups can be neglected. We will refer to this fact as *the fast memory assumption* in the following sections. This is justified in practice by the fact that the tables reside in the main memory, which often corresponds to the Random Access Memory (RAM) of the computer.

One of the objectives of this thesis is to study the effect of using a slower additional memory, such as hard drive disks, possibly mingled with the classical internal memory, for the online phase of the perfect rainbow trade-off. We upgrade the current Random Access Machine (RAM) model to the external memory model, which allows us to perform an analysis of the classical algorithm when using tables on disk. Furthermore, we consider the impact of modern technologies such as Solid State Drive (SSD), as well as recently introduced non-volatile memory (NVM) technologies with which we establish results that will stay pertinent in the forthcoming years. In the following, we first present some context, and then introduce some terminology by describing the two models, before presenting the state of the related work. Then, we perform an analysis of online time algorithms in the context of external memory, and conclude with an experimental validation.

4.1.1 Context

Let us first discuss the consequences on the online phase algorithm of assuming that the tables fit in RAM. In all of the trade-offs variants, during the online phase, potential chains are computed and their endpoints are compared to the ones in the table. In a table of size m, this comparison implies a search in the table which involves about $\log_2 m$ random lookups in the table. The cost of these lookups is neglected when we assume that the memory is fast enough. Then, every search for an endpoint, and the fetching of the corresponding starting point are not accounted for during the online phase. What remains is the computation of the potential chain (or chains in the case of the rainbow trade-off), and the computation of verification chains in the event of false alarms.

The fast memory assumption allows us to reduce the complexity of the analysis. Moreover, we can often assume that the computation of the one-way function f is nearly constant time. This is at least true for the applications presented in Section 2.7 of Chapter 2. Indeed, if f is a cryptographic hash function, its running time is dependent on the number of processed blocks and it can be assumed that for most purposes involving TMTOs we are unlikely to require the hashing of more than one block[1]. If f is constant time, so is the linking function $F = f \circ r_*$, where r_* denotes one of the reduction functions. Indeed, a reduction function is an atomic operation such as a XOR or a modulo, which is constant time. This means that the online running time can be given both in actual time or in number of applications of F without loss of meaning, since the conversion between the two being easily made by multiplying or dividing with the time taken by a single computation of F.

This brings up an important point about the convenience of evaluating the online running time in applications of F. Beyond the simplification of the analysis, it also allows to completely abstract away the underlying hardware, and give online time formulas that can be used by implementers in a generic way. A main use for these formulas is to provide a way to determine the set of parameters of a given TMTO before the pre-computation phase. The abstraction therefore allows the analyst to avoid the

[1] Cryptographic primitives almost always have block size ≥ 512 bits, which implies that anything shorter than 16 bytes is going to be contained in a single block. In the case of passwords, that largely covers what can be done with TMTOs nowadays in most cases.

examination and the choice of a particular technology which would put constraints on the analysis, and to leave it to the implementers. This work allowed us to observe the practicality of TMTO parameters with regard to current technologies. In particular, we present our results in ranges of parameters that make sense in practice.

Note that the fast memory assumption restricts the analysis to only one computer. The RAM on a single computer is limited in practice. There are setups, such as supercomputers, that provide a unified logical interface for huge amount of RAM, and therefore could claim to provide arbitrary amount of memory. However, they actually consist of clusters of interconnected computers under the hood, which are more like a classic, albeit fast, network of somewhat regular computers [AAA+02; YLL+11; TSG11; YSU+11]. The current model prevents us from considering the parallelisation of the offline phase of the trade-off without reconsidering most of the current analysis' formulas. We could consider a parallelisation of the online phase that reduces intercommunication between nodes to a minimum, so that the RAM model is applicable, but, to our knowledge, no analysis has been done in this context. Indeed, even if the access time to the tables themselves is negligible, any splitting of the search across several machines will introduce some synchronisation latency, which cannot be neglected.

Moreover, the restriction of the size of the tables to the size of the available RAM does not make sense in the context of the ever-increasing size of the problem spaces, which are due to an increasing computation capacity that can be allocated to the offline phase. We now give evidence that the study of TMTO should be performed with problem sizes that exceed the size of available RAM.

4.1.2 Towards larger TMTOs

In practice, TMTOs, and especially rainbow trade-offs, that do not fit in internal memory have indeed been implemented for some time. These implementations make use of disks that offer higher storage capacity by at least an order of magnitude with what we can find with RAM. This is the case with some already computed tables available to download. These allow individuals to perform attacks without the need for huge computing power. We already discussed some of these in Section 1.2: the project RainbowCrack [Shu09] makes use of crowd-computing by distributing the pre-computation to participants, while companies such as the one behind the Ophcrack tool [TO05], Objectif Sécurité [Oec17], provides both the online phase tool and several tables for free, while selling tables covering a larger problem space. Computing those paid tables would require substantially more computing power than any individual could reasonably afford.

While the storage of tables on disk is obvious in practice, an online phase using tables on disk with a limited RAM requires additional considerations. The claim that lookups are negligible comes from asymptotic observations. In the rainbow trade-off, on the one hand, the cumulated cost of computing the chains increases quadratically with the length t of the table. On the other hand, the number of lookups is at most linear in t. Therefore, it is said that for large values of t, the cost of a lookup becomes negligible. This may be true when the lookup cost is initially small, as it is in RAM, but when a single search is costly, it becomes predominant in the early stages of the online phase, when the potential chains are short. Search is indeed expensive with most cold storage, such as hard drives disks. Most established analysis methods of TMTOs fail to take this aspect into consideration, while the actual modern usage of TMTOs in practice is mainly done with such large tables.

4.2 Models

We now give more details on the context of our study. We present the two models considered for the analysis of the online phase:

- the *RAM model* used in the majority of the literature is convenient and allows the synthesis of machine-independent formulas, but requires the memory's fetch access time to be negligible compared to the computation of the one-way function, while
- the *external memory model* includes the side effects related to the use of an additional, slower memory.

We describe these two models and establish the limitations of the RAM model. We then explain why the external memory model is chosen for our refined analysis of TMTOs. In particular, we aim at coming closer to what is done in practice.

In the following, the CPU is to be thought of as an abstract computing unit able to process the online phase algorithm, and the memory corresponds to the support of the table. What is meant by disk is a memory with an additional set of constraints regarding the data access. Note that the CPU cache, which is a very fast and a very small memory space available to the CPU, is not mentioned here. The study of the impact of the CPU cache is outside the scope of our study, since the effect of loading data in the CPU directly from RAM instead of the CPU cache is not going to be noticeable in the total lookup time, since it is faster by at least an order of magnitude than RAM access.

4.2.1 RAM model

The *Random Access Machine* model (not to be confused with the physical RAM which constitutes the main memory[2]) is a very early representation of a computer making use of the memory, evolving from the simpler register machine model. It is already used by Elgot and Robinson in [ER64] and formally introduced by Cook and Reckhow in [CR72]. Its simplicity (see Figure 4.1) makes it a common choice for algorithm design in general.

The use of the model with TMTOs consists of three main points:

- each computation of f takes a small amount of $c > 1$ cycles,
- each access anywhere in the memory takes 1 cycle, and
- the memory is unbounded.

As its name suggests, it acknowledges the relation between the CPU and the memory more accurately than the more classic Turing machine with its incremental access to memory. This model adds indirect addressing to the memory, which basically allows random access to data stored anywhere in the memory. In particular, each access to memory, that is, the action of fetching an arbitrary (dependent on the registers size) amount of data from the memory into the register to be processed by a given algorithm, is possible in $O(1)$, regardless of its location.

Limitation of the model. The limitations of the model are twofold. First, as stated above, the infinite quantity of memory is a problem when a single physical system easily bottlenecks the memory required for a TMTO. Also, the constant time access is only a good approximation when the memory is small, and accessed randomly. Modern CPUs tend to access unsolicited adjacent addresses in memory in

[2]The use of the RAM acronym, outside of the few exceptions of referencing the model, here, in this section, will always designate the computer memory.

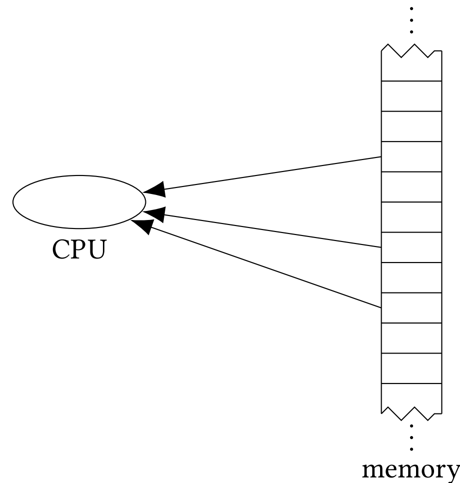


Figure 4.1 – Memory access with the Random Access Machine model

a mechanism called *cache prefetching*. Sequential accesses of memory following a linear pattern are therefore pipelined and take less time to perform starting from the second access. Note that in the case of TMTOs, lookups are generally determined by the potential endpoints, and are therefore random.

RAM access time also varies depending on the address' relative location between fetches. Two distant fetches take more time than two (non-consecutive) close access operations. In particular, characteristics such as the location of the chipset on the motherboard, and which of the Dual Inline Memory Module (DIMM) the data is on, influence the access time. Nevertheless, with a setup equipped with 32GB of memory, we never encountered any RAM access time that was noticeable compared to the computation time of the one-way function, thus we assume that the variance of fetch access times is small enough that the worst access time is still negligible. Furthermore, for the overwhelming majority of one-way functions that could be considered for TMTOs, the worst RAM access time is in fact negligible compared to the computation of the one-way function f.

The model does therefore accurately capture TMTO behaviour when the tables are fully loaded in RAM. An increase by an order of magnitude of the system memory is unlikely to change these results. Increasing the memory further is not practical nowadays, with a maximum of memory available per CPU which revolves around a terabyte of RAM. In the future, the introduction of specialized CPU instructions for hash functions such as SHA1 [Cor13] could reduce the time taken by an application of the one-way function to the point that it is below the time taken by a search in the table. Moreover, table sizes that are in the order of dozens of terabytes of RAM, for which a binary search would take more time than with the current table sizes, would have the same effect. These facts indicate that the evolution of the technology's performance might eventually require a reconsideration of the RAM model.

4.2.2 External memory model

We now present the external memory model, altogether with some architectural concepts of importance regarding data access on disk. These will allow us to refine our analysis in the next sections, by choosing pertinent parameters which reflect the reality better.

The *external memory model*, also called disk access model or I/O model (for Input/Output), takes the

hierarchy of memory, which is present on most computers, into account. In most systems, a distinction is made between the long-term storage, usually on disk, and the main memory, which is reset upon reboots and therefore serves as temporary storage for the need of different algorithms. The model is introduced in [AV88], which focuses on evaluating the cost of common algorithms, when dealing with data on disk.

Indirect access. The key point in this model is that while the processor, as in the RAM model, can directly access the RAM, every access to the disk is indirect, as the data must first be fetched into the main memory before being accessible to the CPU. The great difference in latency between the CPU and the RAM on the one hand, and between the RAM and the disk on the other hand, raises a number of questions about whether it is more efficient to access a small quantity of data or to preload a large quantity of data at once. For some time, systems have been optimized for the latter, with techniques such as Direct Memory Access (DMA) which allows the storage device to transfer large amounts of data directly into the RAM, without involving CPU operations byte per byte, as it was done in the past.

Disks in general do not totally behave like a random access memory, due to physical limitations. In particular, mechanical disks have, for a long time, been divided in sectors of usually 512 bytes, which are the base unit of read and write operations with the disk. Modern disks work with a logical addressing and the concept of sector is now obsolete. Nonetheless, the fact remains that it is not possible to address data on a byte basis within a disk. Indeed they are now divided in *blocks*, whose size can be parameterised by the file system which contains the data. Each time a byte is requested, the whole block is retrieved in the main memory, and can be accessed in a smaller granularity of bytes by the CPU from there, as illustrated in Figure 4.2.

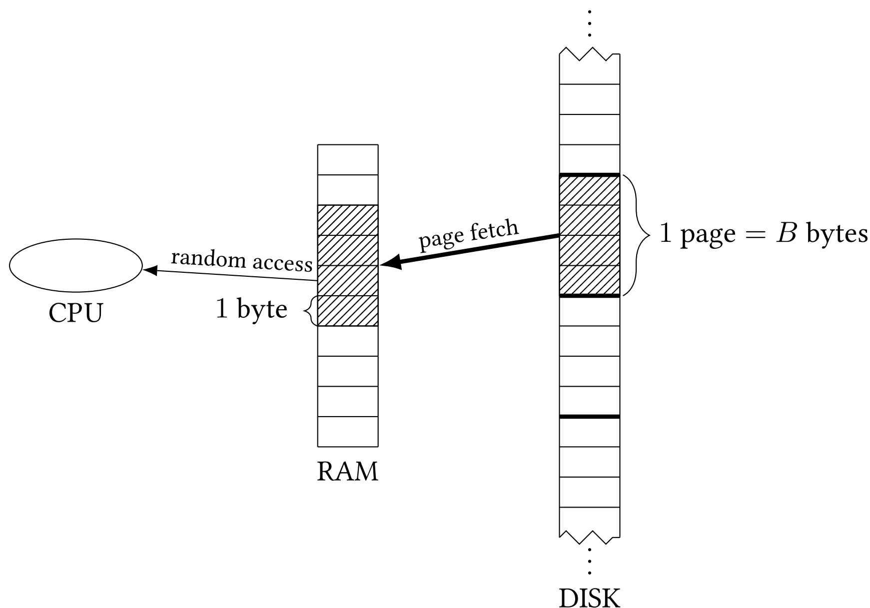


Figure 4.2 – Indirect access from the disk to the CPU

The read operation from the disk is therefore defined in the external memory model as an $O(1)$ operation which retrieves B elements of data, where B is the amount of elements processed by an

algorithm (*e.g.*, a set of bytes, or integers) that fits in a single block. This implies that algorithms which benefit from the fact that they use several data from the same block, that is, which exploit what is called the *locality* of the data, can perform better by reducing the number of operations.

There also exists a minimal amount of disk data that can be accessed in a single fetch operation by the operating system, regardless of the I/O device considered. This amount is called a *page* and has a size of 4KB in an overwhelming majority of situations [WW09a]. While it is possible that it does not match the block size in some consideration, as the block size can be a multiple of a page size, this is usually done in most systems and the special case is not considered. This will be discussed further in Section 4.7, notably with the introduction of the concept of paging, as the operating system influences the behaviour of TMTOs algorithms to the point that the model has to be adjusted. The performance of the algorithm is evaluated in the context where both the data input and output are expected to be on disk.

4.3 Related work on TMTOs in external memory

The problem of TMTOs in the specific case of the external memory has not been addressed by many scientific publications. To the best of our knowledge, only Spitz in [Spi07], and Kim, Hong, and Park in [KHP13] consider the specific impact of disk latency in their running time analysis of TMTOs. The evaluation of disk lookup latencies requires building tables of non-trivial size, and the pre-computation cost involved is a serious impediment to most experiments. This may explain why the study of this phenomenon on TMTOs is scarce in the scientific literature.

Both of the aforementioned works only consider mechanical hard drives as disk storage. The overlook of newer technologies may be explained by either the poor adoption rate at the time of writing, the high price of the first generations of drives, or the limited capacity available which may have precluded interesting size of tables to be considered when compared to what was possible with hard drives.

We begin with an overview of the two works, and position our work with regard to them.

4.3.1 Overview of existing studies

The work of Spitz [Spi07] contains the first mention in the scientific literature of an application of the TMTO technique that accounts for an external memory. It focuses on studying the practicality of TMTOs when using the COPACOBANA FPGA framework [KPP+06]. The main target of [KPP+06] is the cryptanalysis of the DES encryption primitive, with a problem space of size $N = 2^{56}$. They use an array of 4 SATA hard-drives of 500GB each, with a worst case time access of 8ms, and a minimum access time of 0.8ms from which they deduce an average value of 4ms. Using this average value, and taking the constraints imposed by the COPACOBANA architecture into account, they perform a brief comparison of the different TMTO variants. Spitz then settles on the rainbow trade-off because it allows the lowest pre-computation cost for their use case. He use 2TB tables, and aim for a 80% success rate. With this setup, the author indicates that the online phase requires 69h in the worst case, and 35h in average. While the t parameter and the number of tables used are missing, the work shows that such a usage of large tables offloaded on a hard-drive disk is in fact practical.

In [KHP13], Kim, Hong, and Park review some of the implementations of the rainbow trade-off that are publicly available. They observe that the algorithms that are used in practice are different from the ones that are usually dealt with in the literature on TMTOs. In particular, these implementations are often designed to deal with tables that outweigh the available RAM. In [KHP13] the authors present

a formal analysis of one of these algorithms, and compute its optimal TMTO parameters. They also provide experimental results by measuring the time taken by an implementation of this algorithm, and thus confirm the correctness of their analysis. The work of [KHP13] therefore overlaps with the analysis performed in this thesis, in the sense that they also perform their analysis using the external memory model.

Nevertheless, the authors of [KHP13] limit their analysis to one approach taken in implementations. They only consider the strategy where the tables are loaded in RAM, either partially or as a whole, and the online procedure is then performed in the RAM model. Regarding their choice, they state the following:

> "[...] the highly non-localized memory access behavior of the original rainbow trade-off makes its straightforward implementation on a modern computer quite impractical for use, except in the less interesting case of small search spaces."

This point of view seems to be preferred across most TMTO implementers. In [KHP13], Kim et al. demonstrate that their algorithm does perform better with an HDD, for the problems they consider. In our analysis, although we find results in agreement with the work presented in [KHP13], we precise the limits of the conclusion of Kim et al. by showing that, when we extend the scope of the analysis, there are practical situations were the strategy they focus on is not optimal. Furthermore, as stated above, their study lacks the consideration of the evolution of storage technologies, which we include in our analysis.

4.3.2 Algorithms studied

We now describe the different algorithms of interest for our study. We first recall more in depth the classical rainbow algorithm as presented in Section 2.4 of Chapter 2, which, usually in the RAM model, fetches the tables, in the search of an endpoint, at each step of the online phase. This algorithm will be denoted with $\mathsf{Algo}_{\mathsf{DLU}}$ from now on, with DLU standing for Direct Look-Up.

In the RAM model, since there is no lookup cost involved, only the computation of the potential chains is considered. Therefore, we begin with the rightmost part of the table, where the potential chains to be computed are of small length. Considering this, it makes more sense to search for a potential endpoint in the last columns of all of the ℓ tables before moving to an earlier column. This behaviour of $\mathsf{Algo}_{\mathsf{DLU}}$ is illustrated in Figure 4.3. The detailed algorithm of the online phase of the rainbow trade-off is given in Algorithm 4.3. Note that an important step is somewhat hidden in the algorithm: at each step, to determine whether there exists a given $i \in \{1, \dots, m\}$ such that $EP_i = PEP$ where PEP is the potential endpoint at this step, one has to search the table, which may require several fetches in the case of a binary search.

We now discuss the algorithm on which [KHP13] focuses. In the rest of this chapter, we will refer to the algorithm they study with $\mathsf{Algo}_{\mathsf{STL}}$, where the acronym designates the characteristic of the algorithm: the Sub-Table Loading. The algorithm presented in [KHP13] is based on how the two tools rcracki [WNS14] and its improved variant, proposed on [Aut06], perform their online phase. These are two of the most used TMTOs of perfect rainbow table implementations. The algorithm is akin to the idea exposed in [Spi07], and makes several important modifications to the original rainbow algorithm. It is described in details hereafter. Note that the algorithm is applied to perfect tables without further optimisation, and on a single CPU core.

We now describe the $\mathsf{Algo}_{\mathsf{STL}}$ algorithm itself. Its procedure is given by Algorithm 4.4. The input of the algorithm is a value y in $\mathcal{H}$ whose preimage is looked for. The algorithm starts with computing all

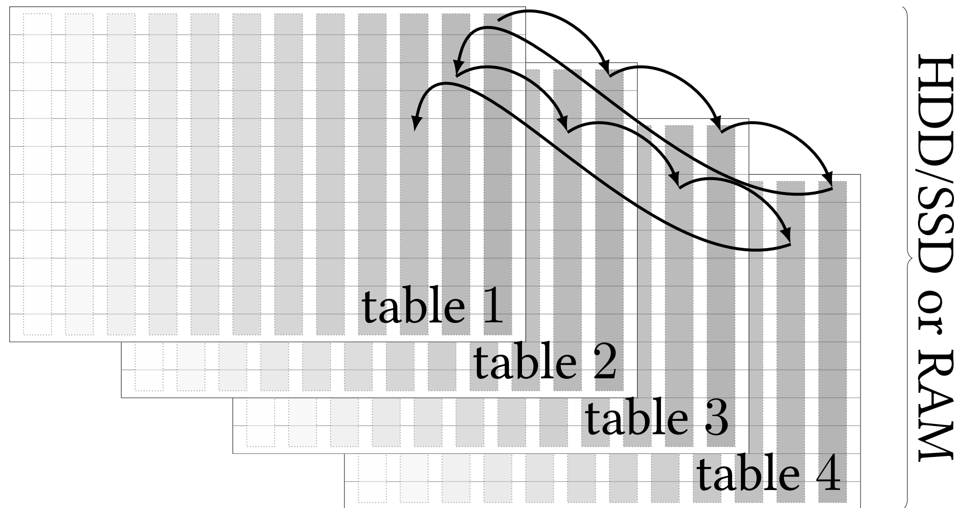


Figure 4.3 – Table lookup path for $\mathsf{Algo}_{\mathsf{DLU}}$ with $\ell = 4$ tables

the t potential chains in an array $(e_j)_{1 \leq j \leq t}$. This differs greatly from the classical variant and is the root of the principal differences in performance that are to be expected between the two algorithms, as this way of processing the table was discarded for performance reasons in the original algorithm.

The kth table table$_k$ ($1 \leq k \leq \ell$), which contains m ordered pairs (starting point, ending point), is split in s *sub-tables* that contain m/s ordered pairs each. Tables are stored in full in an external memory (SSD or HDD) and each of them is virtually partitioned into s non-overlapping sub-tables of a given size, as shown on Figure 4.4. Once every potential endpoint is computed, each of the s sub-tables is loaded into RAM, one at a time. Then, the same online procedure as with a single classical rainbow table in RAM, that is $\mathsf{Algo}_{\mathsf{DLU}}$, is performed on these sub-tables, as indicated by the arrows, for the first columns, on Figure 4.4. For each sub-table loaded, the t potential endpoints $e_1 \ldots e_t$ are used to discover matching endpoints, similarly to what is done in the classical algorithm. When there is a potential match, the classical way of processing it is performed, that is, a verification chain is computed and either the solution is found, or a false alarm is encountered and it must be discarded.

Note that, since the computation of the online chains is done once at the beginning of $\mathsf{Algo}_{\mathsf{STL}}$, the whole set of the t potential endpoints is kept in memory during the whole procedure. When t is large, the memory requirement for these potential endpoints could become non-negligible. However, for practical values of the problem size N, and when the matrix stopping rule $mt^2 \approx N$ holds, we should have t in the order of $\sqrt{m}$. Thus, as long as s is not too large (and we will see in Section 4.4 that s is not too large in practice), the memory required to store these potential endpoints can be neglected compared to the $\frac{m}{s}$ elements in the sub-table.

Overall, the objective of the approach of [KHP13] is to minimise the transfer of tables between the disk and the memory. Computing every endpoint at once is costly, but it allows the algorithm to process the table sequentially afterwards. This implies that every sub-table is only loaded once during the online phase. The order of loading is not important since the cost of each sub-table is the same, and the solution is equally probably in any of the sub-tables.

We finish by mentioning the special case of Ophcrack [TO05], which is a popular tool for rainbow

Algorithm 4.3 Online phase in Direct Look-Up (DLU) algorithm

```
for k = 0 to t do
  x ← r_k(y_0)
  for i = 1 to ℓ do
    for j = k + 1 to t do
      x ← r_j ∘ f(x)
    end for
    if ∃i  s.t.  x = EP_i then
      x ← SP_i
      for j = 0 to k − 1 do
        x ← r_j ∘ f(x)
      end for
      if f(x) = y_0 then return x
      end if
    end if
  end for
end for
```

tables. It is also covered by [KHP13], where Kim et al. describe it as using a hybrid approach. The tool tries to load all of the tables in RAM. If all the tables fit, the classical algorithm $\mathsf{Algo}_{\mathsf{DLU}}$ is used. If only a single table fits in RAM, the tables are loaded one by one, and the search is performed sequentially, in the spirit of $\mathsf{Algo}_{\mathsf{STL}}$ when we consider a single sub-table per table. In the case where a single table outweighs the RAM, another, more complex approach is taken, by using an index file to access the tables. In [KHP13], Kim et al. claim that the algorithm used by Ophcrack is likely to be inefficient because it forces the CPU to be idle most of the time when it performs the lookups, and thus no analysis of this algorithm is performed.

4.4 Performance of the algorithms

We now introduce the scope of our analysis. In addition to the introduction of the classical perfect rainbow algorithm, that we present in the context of external memory, we include a thorough description of the analysis performed in [KHP13], as it will be a point of comparison for our work in the following section.

4.4.1 Context and notations

In order to provide a faithful comparison with [KHP13], we will restrict our discussion to the context presented in [KHP13]. In particular, we focus on the perfect rainbow trade-off. The analysis context is as follows. The inversion (excluding the pre-computation phase) is performed on a single computer with access to RAM and external memory (SSD or HDD).

We consider a trade-off using ℓ tables of size m, computed with chains of length t. In the case of $\mathsf{Algo}_{\mathsf{STL}}$, the number of sub-tables is denoted with s. Pages, referred to as *blocks* in [KHP13], are assumed to contain B rows of a table, in such a way that each table consists of $\frac{m}{B}$ pages.

We reuse the notations of [KHP13] for time constants and denote with τ_F the time (seconds) taken by the CPU to compute an iteration of the one-way function F. Similarly, we need a constant to indicate

Algorithm 4.4 Online phase in Sub-Table Loading (STL) algorithm

```
for i = 1 to ℓ do
  for k = 1 to t do
    e_k ← y
    for n = t − k to t do
      e_k = F^n(e_k)
    end for
  end for
  for j = 1 to s do
    load jth sub-table of ith table in RAM
    for k = 1 to t do
      x ← e_k
      if ∃r  s.t.  x = EP_r in sub-table then
        x_0 = SP_r
        for n = t − k to t do
          x_0 = F^n(x_0)
        end for
        if f(x_0) = y_0 then return x_0
        end if
      end if
    end for
  end for
end for
```


the access time in the external memory. Due to the system of pages which was presented in Section 4.2, the access time when consecutive data is requested, and the one when the access operations are random are unlikely to be the same. We actually introduce two parameters to capture this effect more precisely – τ_S and τ_L. The sequential read time τ_L (seconds), used in the analysis of [KHP13], is the expected time needed to read a value among many consecutive other values. In addition, we introduce the seek time τ_S (seconds), which corresponds to the time taken to read a single random value. In fact, it corresponds to the time needed to fetch a page from the external memory into the RAM. The constant τ_L is likely to be lower than τ_S, even if we take the page mechanism only into account. Indeed, in the event of, *e.g.*, the read of $2B$ values, a sequential read will perform at most 3 page fetches (if the data is not aligned to a page, 2 otherwise). Then, we can expect the sequential constant to be given by an approximation such as $\tau_L \approx \frac{3}{2B}$, whereas the random access τ_S will usually be independent from the amount of data retrieved, as long as $B \ll m$.

When the tables fit in RAM, we place ourselves in the RAM model. We consider that the costs of a memory access and that of an h-operation are of the same order of magnitude, *i.e.*, a few hundred of CPU cycles. Then the assumption that the linear growth of the number of memory accesses is eclipsed by the quadratic growth of the number of computations of the one-way function holds. Therefore, since the table is assumed to be wide enough, we will neglect the memory access time (*i.e.*, $\tau_S = \tau_L = 0$). In that case, the attack time is equal to the amount T of applications of h multiplied by the cost τ_F of a single h operation.

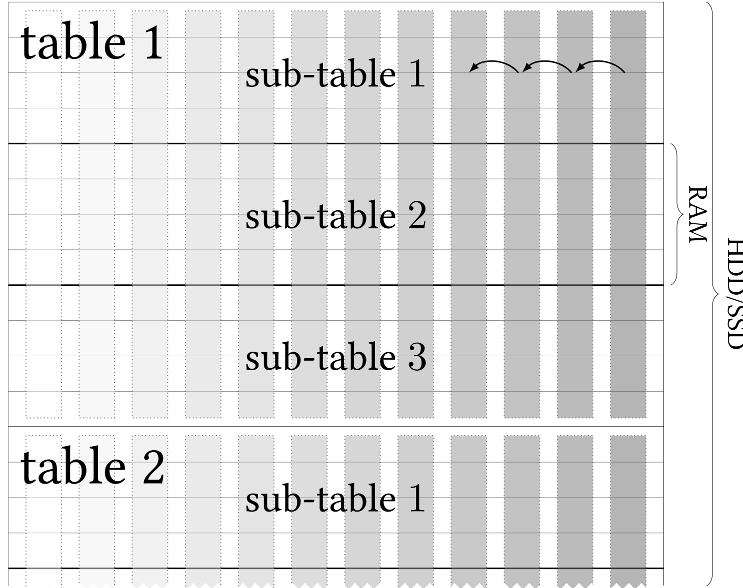


Figure 4.4 – Table division with $\mathsf{Algo}_{\mathsf{STL}}$

4.4.2 Direct lookup algorithm

We now consider what we refer to as the $\mathsf{Algo}_{\mathsf{DLU}}$ algorithm, used to performed the online phase of the rainbow trade-off in the literature, and adapt it to the external memory model. The analysis of $\mathsf{Algo}_{\mathsf{DLU}}$ is actually straightforward, as it suffices to account for the lookup at each column during the online phase. The expected wall-time clock is given in Theorem 4.1.

Theorem 4.1. *Algorithm* $\mathsf{Algo}_{\mathsf{DLU}}$*'s average wall-clock time is*

$$T_{\mathsf{DLU}} = \gamma \frac{N^2}{M^2} \tau_F + \frac{N}{m} (\log_2 m) \tau_S,$$

where γ is a constant dependent on the matrix stopping constant $\bar{c} = \frac{mt}{N}$, and $M = km$, for $k \in \mathbb{Z}$, represents the storage (in bits) of the table.

Proof. We saw in Theorem 2.10, that the online running time of the perfect rainbow trade-off can be approximated by a polynomial $\gamma' t^2$, where γ' is a constant which depends on the matrix stopping constant $\bar{c}$. We have $t^2 = \frac{\bar{c}N}{m} = \frac{\bar{c}Nk}{M}$. If we set $\gamma = \bar{c}k\gamma'$, we obtain the first term of T_{DLU}.

Now we deal with the second term. Recall, from Lemma 2.7 of Section 2.4 in Chapter 2, that the perfect rainbow trade-off follows a geometric law of parameter $\frac{m}{N}$. The expected number of steps before finding the answer is therefore given by $\frac{N}{m}$. This corresponds to the average number of lookups before we find a preimage. A binary search is used to seek the potential endpoint in the table, which means that a lookup requires to seek $\log_2 m$ times in the memory.

Finally, the time taken for a random fetch in the memory was defined to be τ_S. □

The online wall clock time considered in Theorem 4.1 tells us that, while there is an additional term in m, the denominator will dominate the logarithm when m is large. This means that $\mathsf{Algo}_{\mathsf{DLU}}$ will still benefit from an increased memory, while slightly less so than the classical algorithm in the RAM model. We will notice in the following that $\mathsf{Algo}_{\mathsf{STL}}$ is not so fortunate.

4.4.3 Analysis of the algorithm used in practice

We now present an analysis of the algorithm presented by Kim et al. in [KHP13], which is more complex than the one $\mathsf{Algo}_{\mathsf{DLU}}$, because it involves computing the different trade-off characteristics again, to account for the splitting in sub-tables. The overall idea of the analysis is to compute the online time in the context of a large number of sub-tables. In the following, the expected number of sub-table loadings in RAM is given for one part, and the number of computations of the one way function for the other part. These quantities are then simplified using the assumption that $s \gg 1$, to get values independent from s.

Lemma 4.2. *The probability of failure of for the first $i-1$ tables, is given by $e^{-(i-1)\bar{c}}$, where $\bar{c} = \frac{mt}{N}$.*

Proof. Finding an answer in a table fails with probability

$$\left(1 - \frac{m}{N}\right)^t \approx e^{-\frac{\bar{c}}{s}}.$$

Then, $i-1$ tables do not contain the answer with probability $(e^{-\bar{c}})^{(i-1)} = e^{-(i-1)\bar{c}}$. □

Within the table that contains the answer, the probability of failure can also be estimated.

Lemma 4.3. *The probability of failure for the first $j-1$ sub-tables is given by $e^{-\frac{j-1}{s}\bar{c}}$, where $\bar{c} = \frac{mt}{N}$.*

Proof. The failure of the first $j-1$ sub-tables is given by

$$\left(1 - \frac{m(j-1)}{sN}\right)^t.$$

The approximation of this probability by an exponential, like in Lemma 4.2, gives the result. □

From these probabilities, the expected amount of pages that will need to be processed is computed in Proposition 4.4.

Proposition 4.4. *During the online phase of the perfect rainbow trade-off, the expected number of pages that will be processed by the $\mathsf{Algo}_{\mathsf{STL}}$ algorithm is approximately given, when the number s of sub-tables is large, by*

$$L = \frac{1 - e^{-\bar{c}\ell}}{\bar{c}} \frac{m}{B}.$$

Proof. The probabilities given in Lemma 4.2 and Lemma 4.3 give the probability to reach a given table. Such a table contains $\frac{m}{sB}$ pages. The summation for all sub-tables gives

$$\sum_{i=1}^{\ell} \sum_{j=1}^{s} e^{-(i-1)} e^{-\frac{j-1}{s}\bar{c}} \frac{m}{Bs},$$

which is equal to

$$\frac{1 - e^{-\bar{c}\ell}}{1 - e^{-\frac{\bar{c}}{s}}} \frac{m}{Bs}.$$

The claim is an approximation of this result when $s \gg 1$, using the power series expansion of $\frac{1}{1-x}$. □

4

Now that we know how many tables are loaded, it remains to account for the amount of computations of the one-way function. Proposition 4.5 gives $\mathsf{Algo}_{\mathsf{STL}}$'s expected work due to computations of the one-way function.

Proposition 4.5. *The expected computations of the one way function during the online phase of* $\mathsf{Algo}_{\mathsf{STL}}$ *is approximately given by*

$$\mathcal{F} = (1 - e^{-\bar{c}\ell}) \left(\frac{1}{2(1 - e^{-\bar{c}})} + \frac{1}{6} - \frac{\bar{c}}{48} \right) t^2.$$

Proof. The amount $\mathcal{F}$ is the sum of the computations due to the potential chains, and the ones due to false alarms, which are dealt with separately below.

We begin with the computations of the potential chains. The expected amount of tables that are processed during the online phase can be inferred from Proposition 4.4 by setting $s = 1$, and focusing on the number $L\frac{B}{m}$ of tables instead of the number of pages. For each table, $\sum_k^t k \approx \frac{t^2}{2}$ computations are performed. This gives an expected number of computations of

$$\frac{1 - e^{-\bar{c}\ell}}{1 - e^{-\bar{c}}} \frac{t^2}{2}.$$

The amount of computations of the one-way function due to the false alarms performed in [KHP13] is a reevaluation of the formula that was presented in Section 2.4 of Chapter 2, and especially of the approximation given in Theorem 2.10. It is done by replacing the probability of success by the probability of reaching the kth column of the jth sub-table of the ith table

$$p_{i,j,k} = e^{-(i-1)\bar{c}} e^{-\frac{k-1}{t}\frac{\bar{c}}{s}} e^{-\frac{j-1}{s}\bar{c}}.$$

The probability p_k can be computed the same way as in Lemma 4.3, by including the probability

$$\left(1 - \frac{(j-1)m}{sN}\right)^{(t-k+1)}$$

of reaching the kth leftmost column of the given sub-table. The probability of a false alarm at column k was given by Proposition 3.6 of Chapter 3:

$$q_k = \frac{m(k+1)}{N}\left(1 - \frac{mk}{4N}\right).$$

The amount of computations due to false alarms is then

$$\sum_{i=1}^{\ell}\sum_{j=1}^{s}\sum_{k=1}^{t} p_{i,j,k}(t-k+1)\frac{1}{s}q_k,$$

which is approximated by a definite integral and computed in Lemma 3 of [KHP13]. The result, when $s \gg 1$ can be approximated by $(1 - e^{-\bar{c}\ell})\left(\frac{1}{6} - \frac{\bar{c}}{48}\right)t^2$. □

The combination of Proposition 4.4 and Proposition 4.5, associated with the time constants, allows us to obtain a wall-clock time of the expected duration of the online phase of the perfect rainbow trade-off, when using $\mathsf{Algo}_{\mathsf{STL}}$. It is given in Theorem 4.6.

Theorem 4.6. Algo$_{\mathsf{STL}}$*'s average wall-clock is given by*

$$T_{\mathsf{STL}} = L \cdot \tau_L + \mathcal{F} \cdot \tau_F,$$

where

$$L = \frac{mP}{\bar{c}B}, \quad \mathcal{F} = \frac{P^3}{B^2}\left(\frac{1}{2(1-e^{-c})} + \frac{1}{6}l - \frac{c}{48}\right)\frac{N^2}{L^2},$$

$\bar{c} = \frac{mt}{N} < 2$*, B is the number of table rows (starting point – endpoint) per page, and $P = 1 - e^{-c\ell}$ is the total probability of success.*

Note that the online time given in Theorem 4.6 assumes that the loading operations and the computation operations happen sequentially. While it would be possible to design an algorithm which performs these task in parallel, such techniques are outside the scope of [KHP13], whose authors remark that no parallelism is employed by the implementations [WNS14] and [Aut06], on which Algo$_{\mathsf{STL}}$ is based.

4.4.4 Optimal configuration

Algo$_{\mathsf{STL}}$ has an optimal amount of memory at which it operates. This is because in the external memory model, we have $T = O\left(m + \frac{1}{m^2}\right)$, instead of $T = O\left(\frac{1}{M^2}\right)$ in the RAM model. Beyond a certain threshold, the decrease of the $\mathcal{F}$ factor fails to compensate for the increase of the L factor. This behavior is further commented on in Section 4.6.

In order to determine the optimum, the authors of [KHP13] first set the matrix stopping rule such that it is optimal with regard to ℓ: $\bar{c} = -\frac{\ln(1-P)}{\ell} < 2$. Then a coefficient δ which characterises the new trade-off curve of Algo$_{\mathsf{STL}}$ is given by:

$$L^2\mathcal{F} = \delta N^2.$$

The approximation of this coefficient when $s \gg 1$ is given in Theorem 4.7, alongside the wall clock time of Algo$_{\mathsf{STL}}$ in the optimal setup. The minimal time is obtained by replacing $\mathcal{F} = \frac{\delta N^2}{L^2}$ in Theorem 4.6 to obtain the online time as a function of L and finding the minimum for that function.

Theorem 4.7. Algo$_{\mathsf{STL}}$*'s minimal wall-clock time is given by*

$$T^*_{\mathsf{STL}} = \frac{3}{2^{\frac{2}{3}}}\tau_L^{\frac{2}{3}}\tau_F^{\frac{1}{3}}\delta^{\frac{1}{3}}N^{\frac{2}{3}},$$

with

$$L = \left(\frac{2\tau_F}{\tau_L}\right)^{\frac{1}{3}}\delta^{\frac{1}{3}}N^{\frac{2}{3}} \quad \textit{and} \quad \mathcal{F} = \left(\frac{\tau_L}{2\tau_F}\right)^{\frac{2}{3}}\delta^{\frac{1}{3}}N^{\frac{1}{3}},$$

where

$$\delta = \frac{P^3}{B^2}\left(\frac{1}{2(1-e^{-c})} + \frac{1}{6}l - \frac{c}{48}\right).$$

The memory associated to the optimal case corresponding to Theorem 4.7 is given by

$$m^* = \left(\beta\frac{\tau_F}{\tau_L}\right)^{\frac{1}{3}}\left(\frac{1}{1-e^{-c}} + \frac{1}{3} - \frac{c}{24}\right)^{\frac{1}{3}} cN^{\frac{2}{3}}.$$

We finish the presentation of Algo$_{\mathsf{STL}}$ by discussing the s parameter.

Optimal number of sub-tables. The value s (number of sub-tables per table) is thoroughly discussed in [KHP13]. If s is too small, sub-tables are very large, and when the search ends, it is likely that significant time was wasted loading the last sub-table. If s is too big, read operations are done on a small amount of memory, where the TMTO search is sub-optimal. As stated in [KHP13] however, the value of s has relatively little impact on the efficiency of $\mathsf{Algo}_{\mathsf{STL}}$, provided it is "reasonable" (at least 45 in the examples discussed in [KHP13]). In what follows, we assume such a reasonable s is used.

4.5 Establishing the constants

In Section 4.4 we introduced the τ_S, τ_L and τ_F parameters in order to compute the expected online time of both $\mathsf{Algo}_{\mathsf{STL}}$ and $\mathsf{Algo}_{\mathsf{DLU}}$ algorithms. In this chapter, we aim at presenting our analysis in a practical context. Therefore, before we begin the actual comparison in Section 4.6. In this section, we establish values for these parameters by performing hardware measurements. We also introduce here our experimental setup which will be used to perform, in Section 4.7, an experimental validation of the results of Section 4.6.

4.5.1 Experimental setup

The measurements that will be performed, both in this section and in Section 4.7, have all been done on the same machine, whose characteristics are detailed below.

CPU. The CPU is an Intel `E5-1603 v3` CPU clocked at 2.8 Ghz. This processor does not have dynamic over-clocking, such as, *e.g.*, *Turbo Boost* capabilities, which means that it performs exactly 2.8 billion cycles per second.

In fact, this class of processors is featured with an *invariant time stamp counter (TSC)*, about which the manual from the CPU vendor [Cor11] states:

> "The invariant TSC will run at a constant rate in all ACPI P-, C-., and T-states. [...] On processors with invariant TSC support, the OS may use the TSC for wall clock timer services (instead of ACPI or HPET timers)."

This guarantees that the computations are expected to perform the same way, even in the event that the processor goes in sleeping mode, or any power-saving state. Therefore, we can use the processor's internal time stamp counter, via the RDTSC instruction, to make measurements. This instruction is also synchronized across all cores on this CPU model, which prevents any race condition during the experiments. This allows for accurate measurements up to a nanosecond precision.

RAM. The machine is equipped with a total amount of 32GB of RAM memory, consisting of 4 memory sticks of 8GB each. These are running at 1833Mhz, which means that a memory cycle takes approximately 0.5ns. This gives an idea of the order of magnitude that we can expect from a memory fetch from the RAM to the CPU. In particular, we will see that all the time constants computed in this section are well above this value. This validates the assumption that RAM operations can be neglected.

External memory. The SSD disk in use is an Intel SSD `DC-3700` with a capacity of 400GB. We will simply denote it with SSD in the rest of this chapter. Non-Volatile Memory Express technology, shortened as NVMe [CH12], is a standard interface introduced in 2011 designed for businesses in the

storage industry with solutions based on PCI-Express SSDs. Its deployment went beyond the business world, and its current adoption in commodity computers suggests that it will become more and more prevalent in forthcoming years. The NVMe interface provides smaller latencies between the disk and the CPU than the previous Serial-ATA (SATA) interface, by connecting the SSD directly via PCI-Express to the processor and memory, instead of going through an intermediate controller, as it is the case with traditional hard disk drives. The machine also possesses a more classical hard drive, which is a Seagate ST4000DM000-1F21 of 4TB capacity, with a speed of 5400 rpm, and connected via a Serial AT Attachment (SATA) controller. It is denoted with HDD in the following. The operating system from which the experiments are performed is not located on either of these two disks, in order to avoid interferences in the measurements.

4.5.2 Computation time

We first determine the time of a computation of the one-way function. In order to get a time value in seconds we have to fix a particular one-way function. We chose to use the MD5 hash function for this purpose. This is also the function that will be used in our experimental validation in Section 4.7. It is, together with SHA1, one of the most used functions in rainbow tables, according to the offering of tables that can be found on [Aut06]. Note also that it is close in design and in speed, to the NTLM password function, which is one of the most well known application of rainbow tables since the work of Oechslin [Oec03]. We assume that, during the execution of the TMTO, the CPU is warmed-up, that is, it processes the one-way function at the best of the possibilities of its pipeline. In particular, no other process is interfering with its instruction cache.

In this case, the time taken by successive applications of f is constant. We then estimate the time τ_F taken by a single application of f by averaging over the measurement of 10^6 applications of f. We obtain $\tau_F = 1.786 \cdot 10^{-7}$s, which is about 180ns. It is indeed way above the time needed to fetch data in RAM. Note that MD5 is among the fastest cryptographic hashes, and globally among the fastest functions that qualify to be used in TMTO, which means that this value of τ_S should be considered the lower end of the practical range of values it can take.

4.5.3 Single block read time

Calling the random access time a constant is slightly misleading, as this value depends on several factors, similar to those which affect RAM access time that were enumerated during the description of the RAM model in Section 4.2. Preliminary observations showed that the access time depends on the location of the data on the disk, and in particular on which chipset it was stored, and from the behavior of the firmware of the disk. Unfortunately, the schematics of the disks often constitute trade secrets and are unlikely to be disclosed, along with the source code of said firmwares. Moreover, such a model would have been tied to a particular vendor and therefore not that useful when extending these results to other disk brands. Initiatives such as LightNVM [BGB17] exist so that the disk internals would be standardised in such a way that the operating system would manage the disk more directly. However, those have yet to be adopted by the industry to be considered in works such as ours.

Therefore, for this experiment, we rely on an average of the random access time to determine τ_S, and treat it as a constant. We measured the time taken by successive single-page reads of values at random positions on the block device. Each read is measured separately in the sense that no cache is kept between measurements. Moreover, the first measurement is discarded as a warm-up round.

The value obtained is an average over 500 measurements, and it is given in Table 4.1, alongside the maximum, minimum, and standard deviation.

Table 4.1 – Average random read of a single page from the SSD and the HDD (500 runs)

disk	min	average	max	deviation
SSD	108.6 μs	164.5 μs	455 μs	66.2 μs
HDD	4.06 ms	15.7 ms	26.2 ms	4.21 ms

As expected, the latency of the SSD is much lower than the one of the classical hard drive, by a factor 100, which is easily explained by the mechanical nature of the hard drive which incurs a penalty when the read mechanism has to wait for the physical disk to rotate, and by the intermediate controller which adds an extra layer to the communication with the CPU compared to the hard drive.

4.5.4 Sequential block read time

In the context of $\text{Algo}_{\mathsf{STL}}$, the constant τ_L refers to the time taken to read a page on disk during a sequential read of many blocks. The sub-tables are typically chosen to reach the maximal read throughput of the disk, with sizes in the order of the dozens or hundreds of megabytes. We note that, since disk usually have much better performance in sequential reads than in random access, we should have $\tau_L \ll \tau_S$. We measure the average time to load random data ranging from 1KB to 1GB, and average the read of a single page from the time taken by loading all the data into the RAM. The results are given for an average over 100 measurements for each size, in Figure 4.5.

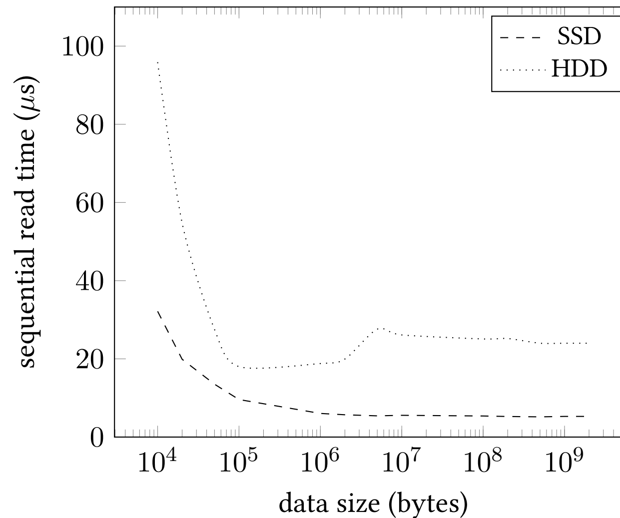


Figure 4.5 – Evolution of sequential read time with the size of the data

We remark that the sequential read time is high for data of low size, which would correspond to the case were the sequential read is akin to a random access, and decreases afterwards when it becomes amortised by the volume of data. Notice that for small chunks of data, between 10^5 and 10^6 bytes, a caching effect can be observed on the HDD, that is completely absent on the SSD. Such an effect cannot be relied upon directly in the context of a TMTO search though, since in a real search, consecutive

access would happen much more frequently than in our measurement. Therefore, we have to ignore this caching effect when choosing τ_L. Moreover, sub-tables of such size are likely to be very inefficient, as the table lookup would not take advantage of the fact that it is in a fast memory, since too few values would have to be searched. We take the stabilised values, for size > 10MB of data, from Figure 4.5 as τ_L. For the HDD, we obtain $\tau_L = 24 \cdot 10^{-6}$s with a standard deviation $\sigma = 1.02 \cdot 10^{-6}$s. For the SSD, we have $\tau_L = 5.28 \cdot 10^{-6}$s with $\sigma = 6 \cdot 10^{-8}$s.

The x-value of where the sequential read time stabilises seems to indicate that the optimal number of sub-tables for Algo_{STL}, for these specific disks, is consistent with the observations of [KHP13]: with $s > 45$, sub-tables with sizes in the tenth of MB would correspond to tables starting from 500MB which seems reasonable as a lower bound on the size of the tables. Recall that the formula given in Theorem 4.6 of Section 4.4 is independent from s as long as the latter is not too small, which is the case here, so a slight variation of s will not affect the results. About the values obtained, we remark that, again, the obtained values for τ_L show that the SSD performs better than the HDD, but note that the factor between the two values is only of 4. The difference is much lower in the sequential case than with the random access time, which means that Algo_{STL} will not improve greatly by using a lower latency disk as Algo_{DLU} would.

4.6 Analysis

This section evaluates the algorithms Algo_{STL} and Algo_{DLU} which were both described in Section 4.3. The analysis is performed on different memory types and aims to characterize which of the two algorithms performs better depending on various parameters. For clarity, let us introduce some notation to designate the algorithms in their context of application. We denote with $\text{Algo}_{\text{DLU/RAM}}$ the Algo_{DLU} algorithm performed in RAM. When the algorithm is used in external memory, we distinguish between whether the external memory is the HDD or the SSD by using the $\text{Algo}_{\text{DLU/HDD}}$ and $\text{Algo}_{\text{DLU/SSD}}$ notations, respectively. Similarly, we denote with $\text{Algo}_{\text{STL/HDD}}$ and $\text{Algo}_{\text{STL/SSD}}$ the algorithm Algo_{STL} when it is performed on disk. Note that there is no analog of $\text{Algo}_{\text{DLU/RAM}}$ for Algo_{STL}, since it corresponds to the case where both the external memory and the fast memory designate the RAM. This algorithm would imply to treat the tables sequentially in RAM, which does not make sense compared to $\text{Algo}_{\text{DLU/RAM}}$.

We look at the case where the memory that is used is RAM. In such a case, the $\text{Algo}_{\text{DLU/RAM}}$ algorithm is based on the assumption that the tables entirely reside in RAM, and no external memory is used. It takes advantage of fast RAM access operations given that we assume the RAM access time is negligible. In this case, formula of Theorem 4.1 is simplified and $\text{Algo}_{\text{DLU/RAM}}$'s average wall-clock time becomes

$$T_{\text{RAM}} = \gamma \frac{N^2}{M^2} \tau_F.$$

4.6.1 Problem parameters

We now specify the parameter of the TMTO considered in our analysis.

A comparison between $\text{Algo}_{\text{STL/HDD}}$ and $\text{Algo}_{\text{DLU/RAM}}$ was done in [KHP13]. However, this comparison is limited in several ways. Most importantly, it only accounts for Algo_{STL} in optimal configuration, that is, with a fixed memory size. Furthermore, it only considers two data points of memory for $\text{Algo}_{\text{DLU/RAM}}$ and one for $\text{Algo}_{\text{STL/HDD}}$, and does not study $\text{Algo}_{\text{DLU/HDD}}$. The conclusion drawn

in [KHP13] is that $\mathsf{Algo}_{\mathsf{STL}}$ is better for large problems, but the comparison is inconclusive for smaller problems.

We base our comparison on the "Small Search Space Example" given in [KHP13], on which $\mathsf{Algo}_{\mathsf{DLU/RAM}}$ and $\mathsf{Algo}_{\mathsf{STL/HDD}}$ have been compared. The problem space of this example corresponds to passwords of length 7 over a 52-character alphabet (standard keyboard), which gives $N = 52^7 = 2^{39.903}$. For this TMTO, a number of $\ell = 4$ tables is considered, with a matrix stopping constant $\bar{c} = \frac{mt}{N} = 1.7269$ for each table. The TMTO success rate is therefore of $P = 0.999$. It is assumed in [KHP13] that the elements of the problem space are stored in 8 bytes, without any special storage technique used. While page size is theoretically dependent on the system architecture, it can be noted that a quasi universal constant of 4KB has been used in most computers since the 70s [WW09b]. This gives that with 16 bytes per row in the table, we obtain a value $B = 256$ rows per page. With these TMTO parameters, we deduce numerical values for analytical formulas presented in Section 4.4. The trade-off coefficient used in Theorem 4.7 is $\delta = 0.73662/\beta^2$, and the γ factor of Theorem 4.1 evaluates to $\gamma = 8.3915$.

For τ_F, τ_L, τ_S, we use values obtained experimentally in Section 4.5, Note that a value for τ_L was computed in [KHP13]. However, this constant emanated from a different machine, using a different HDD, which would not have allowed us to validate our findings experimentally. Moreover, their value would not be consistent with the value of τ_L we computed for the SSD. The same remark applies to the random seek time τ_S which was not considered at all in [KHP13].

4.6.2 Comparing the two algorithms

We begin with a comparison of the online running time of the two algorithms in external memory. Figure. 4.6 presents the average wall-clock time for the two algorithms $\mathsf{Algo}_{\mathsf{DLU/SSD}}$ and $\mathsf{Algo}_{\mathsf{STL/SSD}}$, for a varying amount of memory. We also include $\mathsf{Algo}_{\mathsf{DLU/RAM}}$, at a somewhat unfair advantage since it uses RAM instead of external memory, and is obviously expected to perform better in any case. It is therefore only presented for reference.

We notice that the cost of $\mathsf{Algo}_{\mathsf{STL}}$, represented in Figure 4.6 with a green solid line, stops decreasing beyond a certain amount of memory available. This is due to the fact that the time taken to load an increasing amount of chains in RAM, given by the $L\tau_L$ term in Theorem 4.6, becomes predominant in the total online running time. Since it is an increasing function of m, there exists a threshold beyond which it is no longer made up for by the decrease in computation provided by the fact that the algorithm runs in RAM.

The minimum online time of $\mathsf{Algo}_{\mathsf{STL}}$, denoted by "opt" in Figure 4.6, corresponds to the optimal amount of memory which gives the optimal case mentioned in Section 4.4, and whose online time was given in Theorem 4.7. Any increase of memory beyond this point is useless for this algorithm. This means that $\mathsf{Algo}_{\mathsf{STL}}$ has an inherent minimal average search time which can never be improved upon regardless of the memory available. It is assumed that the optimal amount of external memory is used when possible, even when more disk space is available. This is the green dot-dashed line which is denoted with "post-opt", for post-optimal, in Figure 4.6.

We will now focus on $\mathsf{Algo}_{\mathsf{DLU}}$. Unlike $\mathsf{Algo}_{\mathsf{STL}}$, it has no threshold and continues to decrease throughout the whole range of external amount of memory considered. We can notice a slight inflexion in the curve of $\mathsf{Algo}_{\mathsf{DLU}}$ due to the lookup term $\log_2 m\tau_S$ of Theorem 4.1, but it is obviously very moderately increasing with m. It is small enough for small amount of memory that the online time of $\mathsf{Algo}_{\mathsf{DLU}}$ is indistinguishable from whether it is performed in RAM or on disk. In fact, the amount of

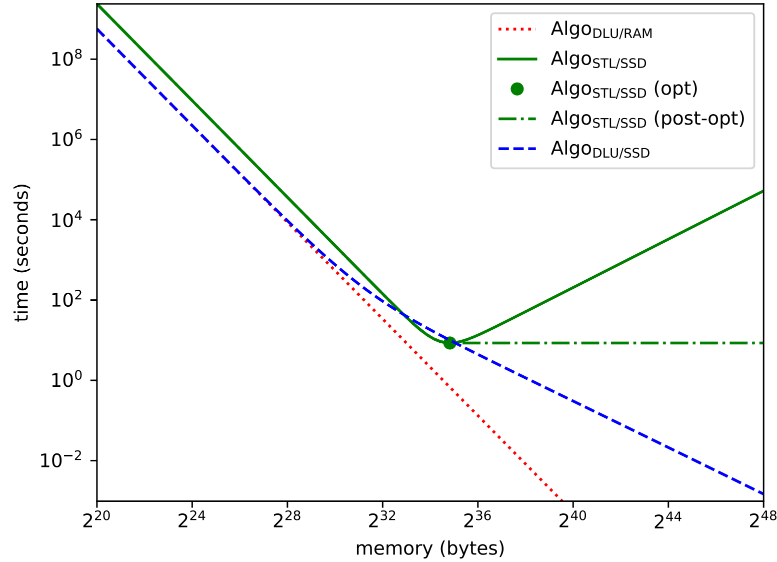


Figure 4.6 – Average wall-clock time of the algorithms on SSD

time spent in lookup can be clearly seen as the difference between the red dotted line of $\text{Algo}_{\text{DLU/RAM}}$ and the blue dashed line of Algo_{DLU}.

The area (in terms of the external amount of memory) where Algo_{STL} is more efficient than Algo_{DLU} happens to be quite small. Nevertheless, it must be noted that the range of memory for which Algo_{STL} thrives is the most likely to happen in practice as this range corresponds to the amount of memory commonly found on commodity computers.

We now consider the online running time on HDD of the algorithms. In that case, the constant τ_L is moderately increased, and the constant τ_S is increased more substantially. The running time is presented the same way as in Figure 4.7.

The behaviour of the two algorithms is somewhat similar to that of Figure 4.6. However, we can see that, expectedly, $\text{Algo}_{\text{DLU/HDD}}$ suffers from longer seek time, and differs from $\text{Algo}_{\text{DLU/RAM}}$ starting from more modest amount of memory. This implies that $\text{Algo}_{\text{STL/HDD}}$, whose online time is almost unchanged, dominates on a much larger area. This concurs with the observations of [KHP13]. However, notice that for very large tables, Algo_{DLU} is able to lower the online time arbitrarily, allowing, *e.g.*, to perform an inversion in less than a second in average, which is impossible for Algo_{STL}. We also notice that Algo_{STL} is superior to Algo_{DLU} in a significant area where it is not in its optimal state. Moreover, this area corresponds to the range of commonly available amount of memory.

We now consider changing the other parameters, one at a time, and observe the resulting changes in the algorithms' behaviour. The results are gathered in Figure 4.8.

Our first change is to use a much slower one-way function. The `md5crypt` function is a password derivation function which is used to hash UNIX passwords. It consists in hashing its input several times with the MD5 function, and is thus much slower. We determine a new constant, by taking an average over 10^3 computations, in a similar way to what we did for `MD5`. We observe that it is 955 times slower than `MD5`, with $\tau_F = 0.171\text{ms}$. The result when using this new τ_F appears in Figure 4.8a.

Then, we consider a TMTO with a much larger problem space size of $N = 2^{56}$, corresponding to the key space of the DES cryptographic function. The other parameters, such as the matrix stopping

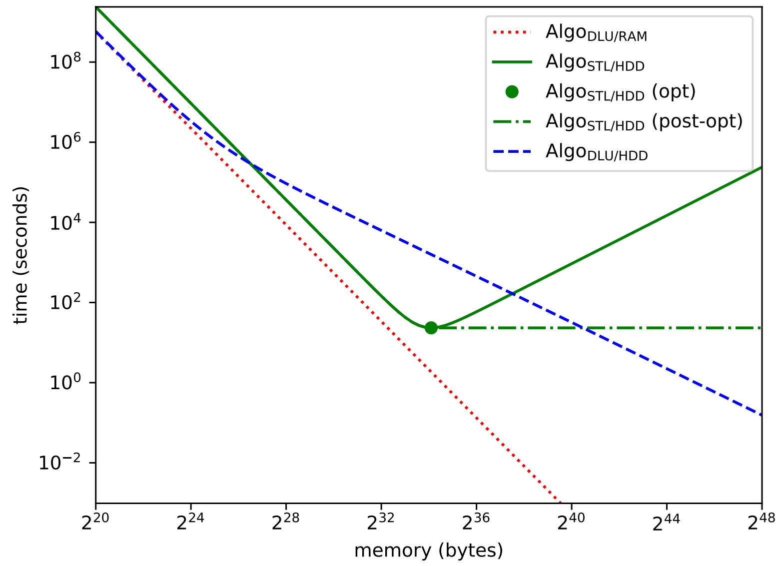
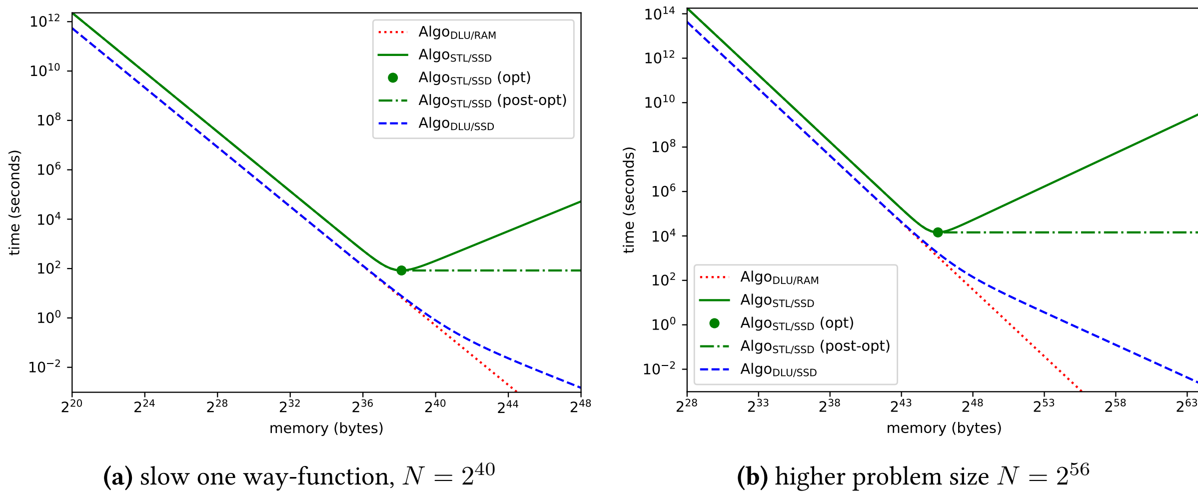


Figure 4.7 – Average wall-clock time of the algorithms on HDD

constant $\bar{c}$, the number of tables ℓ, and therefore the success rate, are unchanged. The results for this larger TMTO are presented in Figure 4.8b.



(a) slow one way-function, $N = 2^{40}$

(b) higher problem size $N = 2^{56}$

Figure 4.8 – Average wall-clock time for different parameters

We can observe a similar behaviour happening in the two cases. The Algo$_{\text{DLU/SSD}}$ algorithm becomes superior throughout, following Algo$_{\text{DLU/RAM}}$ more closely. The fact that these two changes have the same result is not surprising, since they both imply an increased amount of time spent computing the one-way function. This is trivially explained when τ_F is high. With a higher problem space, for the same memory, the chains are longer, and therefore the amount of time spent in each sub-table in Algo$_{\text{STL}}$ becomes predominant. Therefore, Algo$_{\text{DLU}}$ is more adapted for such large problems, as the constraint of the minimum of Algo$_{\text{STL}}$ is going to be a problem when one wants to further reduce the time of such

an inversion.

4.6.3 Best algorithm

We now consider which algorithm is best according to a given setup. By setup, we mean a machine including both RAM and external memory up to a certain amount. In particular, for each setup, we evaluate, using formulas given in Section 4.4, which of Algo_{STL} or Algo_{DLU} performs better.

At the extreme, TMTOs are not the most efficient algorithm to consider. When the memory is too small to store a table of sufficient size, it does not make sense to compute long chains to find the answer. Similarly, if the memory can hold the whole problem space, the TMTO online phase can be beaten by better search algorithms. Therefore, we also consider the time that would be taken by an exhaustive search of the entire problem space, *i.e.*, a so-called "brute-force", and the use of a dictionary. With a brute-force, the answer is expected to be found after testing half of the problem space, so its cost T_{bf} is given by $T_{\text{bf}} = \frac{N}{2}\tau_F$ operations. If we denote by $\mathcal{N}$ the problem space, we will consider as a dictionary an array consisting of $f(x)$, for all $x \in \mathcal{N}$, that is sorted such that the ith element of the array contains $f(i)$. Then, using a binary search, the expected time T_{dict} needed to perform an inversion is given by

$$T_{\text{dict}} = \begin{cases} (\log_2 N)\tau_S & \text{in external memory, or} \\ 0 & \text{in RAM.} \end{cases}$$

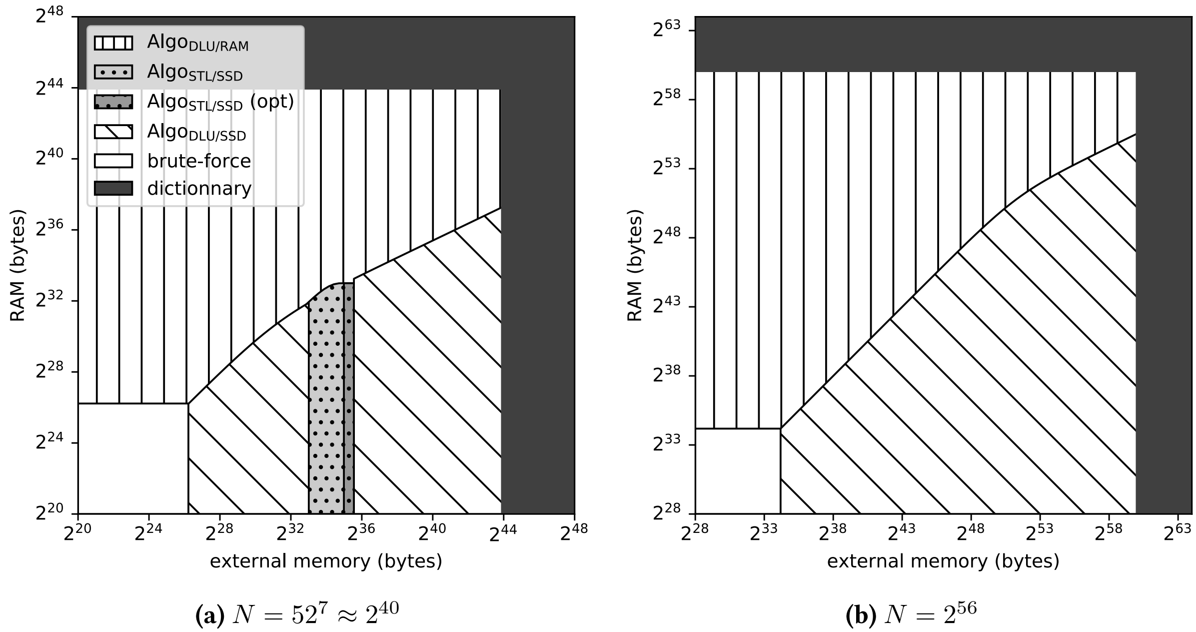
Since, for any TMTO, we have $m \gg 1$ and $t \ll 1$ (or else the pre-computation is not worth it), the quantity T_{dict} is smaller than any TMTO on the same problem space. We have that each $f(x)$, as a MD5 hash, is stored in 16 bytes, so it is assumed that if the memory exceeds $16N$, the dictionary method is better than both Algo_{STL} and Algo_{DLU}.

We first consider the case where the external memory is the SSD. Figure. 4.9 presents our findings in regions where, in terms of RAM and external memory available, which of $\text{Algo}_{\text{STL/SSD}}$, $\text{Algo}_{\text{DLU/SSD}}$, brute-force or the dictionary method is the most efficient. In addition to the problem space considered in this section, whose results are given in Figure 4.9a, we present in Figure 4.9b the results for a second problem space of $N = 2^{56}$, as it was done for Figure 4.8b.

Each point of the two figures corresponds to a given setup. The y-axis represents the amount of RAM available for the online phase, and the x-axis gives the amount of external memory available to store the TMTO tables. Both axes are delimited by the black regions, where the dictionary method is preferred. In both figures, the roughly diagonal separation gives, for each table storage, the amount of RAM beyond which one should not consider exploiting the SSD at all. In Figure 4.9a, when there is not enough RAM, the lower part of the graphics shows where each algorithm is more efficient on external memory. We see that for a limited range of memory $\text{Algo}_{\text{STL/SSD}}$ is the best choice. It is clearer in this graphics that the part where $\text{Algo}_{\text{STL/SSD}}$ performs better in its optimal state is very small, compared to the non-optimised version of the algorithm.

In Figure 4.9b, where the problem space is increased to $N = 2^{56}$, we can immediately see that there is no setup where $\text{Algo}_{\text{STL/SSD}}$ is the best algorithm, either in its classic or optimised state. In this case, only the amount of RAM has to be considered in a given setup to see if $\text{Algo}_{\text{DLU/RAM}}$ or $\text{Algo}_{\text{DLU/SSD}}$ has to be used, *i.e.*, whether the tables should be computed in a way that they fit in the setup for a given amount of RAM.

We now reproduce the same procedure, that is, we determine the best algorithm for a given setup, for the case where the external memory of the setup is in a HDD. Again, we propose to consider the same two problem spaces as in Figure 4.9. The results are given in Figure 4.10.



(a) $N = 52^7 \approx 2^{40}$ **(b)** $N = 2^{56}$

Figure 4.9 – Best algorithm for a given RAM and SSD configuration

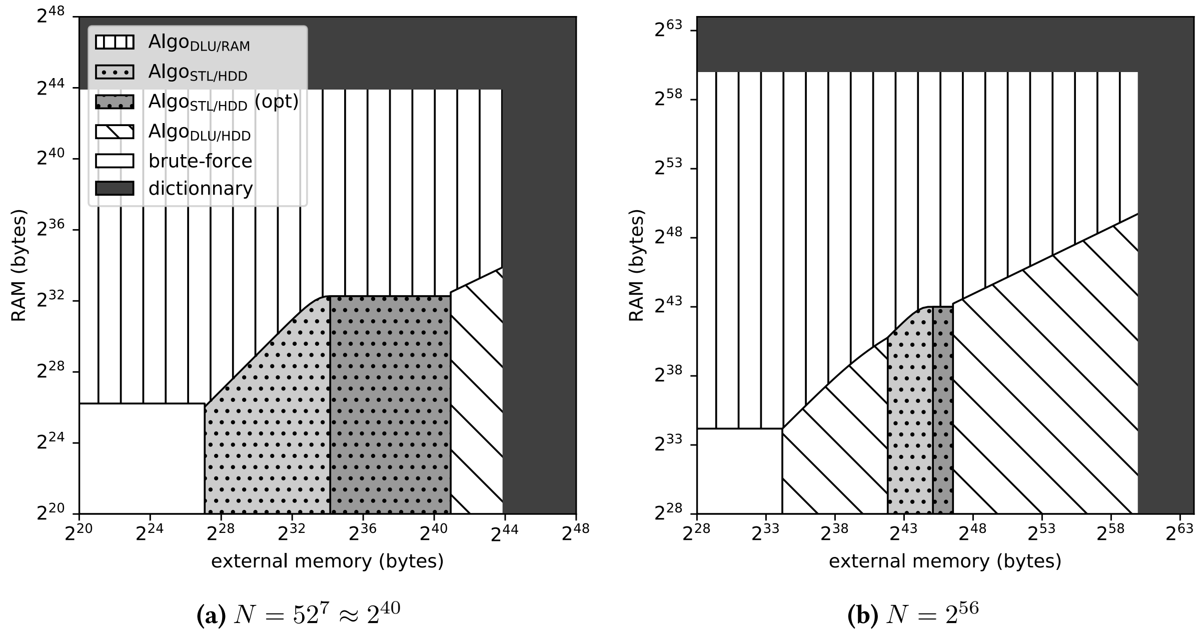
We can see from Figure 4.10a that $\mathsf{Algo}_{\mathsf{DLU}}$ loses its advantage for smaller problem space when there is not enough RAM to store the table. For the smallest amount of memory, where the brute-force approach is not the most efficient, $\mathsf{Algo}_{\mathsf{STL/HDD}}$ is the best algorithm to tackle the inversion. We can see clearly that the optimal configuration of $\mathsf{Algo}_{\mathsf{STL/HDD}}$ performs best in the range of the most common amounts of HDD, *i.e.*, from hundreds of GB up to several terabytes. For larger amount of external memory though, $\mathsf{Algo}_{\mathsf{DLU/HDD}}$ becomes better.

The case for the large problem space $N = 2^{56}$, presented in Figure 4.10, is similar to the one of the small space in SSD that was given in Figure 4.9a. This is explained by the fact that we simultaneously increased the external memory time constants, the problem space, and the amount of memory, thus not changing the overall situation. In practice, this corresponds to increasing the problem space and using hard drives of high capacity instead of the more expensive SSDs, in a way that the monetary cost stays the same. This cost consideration is discussed in further detail in the following.

4.6.4 HDD and SSD

We now present how the two algorithms compare with regard to the type of external memory. Figure. 4.11 compares the performance of $\mathsf{Algo}_{\mathsf{DLU}}$ and $\mathsf{Algo}_{\mathsf{STL}}$ on SSD and HDD. Again, we give the results for two problem sizes.

The dashed line represents points where SSD memory and HDD memory size are equal. Nowadays, HDD memory is cheaper than SSD. Such setups are represented in the region above the dashed line. The prices of both hard drive and SSD fluctuate, making it difficult to establish a correspondence between the amount of hard drive and SSD memory of equivalent prices. Nevertheless, Figure 4.11 gives an indication as to which algorithm is better when considering a fixed cost. If the price difference is linear in the size of the memory, the line corresponding to an equal cost will be the dashed line moved by a translation up the y-axis. The higher the difference of prices between the two types of disk, the less



(a) $N = 52^7 \approx 2^{40}$ **(b)** $N = 2^{56}$

Figure 4.10 – Best algorithm for a given RAM and HDD configuration

likely it is that $\mathsf{Algo}_{\mathsf{DLU/SSD}}$ will perform better.

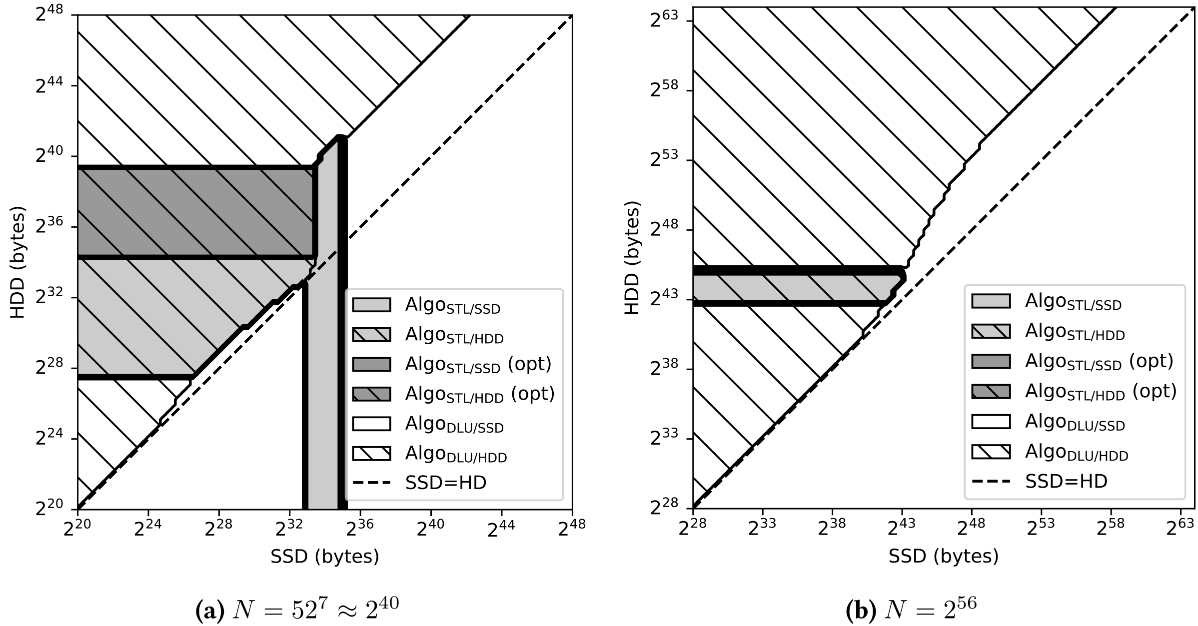
We can see in Figure 4.11a, with the right part of the graphics just above the dashed line, that as long as SSDs are not more than approximately 50-100 times more expensive than the corresponding amount of HDD, they are more suitable for TMTOs for moderate amount of memory, since $\mathsf{Algo}_{\mathsf{STL/HDD}}$ is the best algorithm in this case, as indicated by the bump of the solid grey zone above the dashed line. If HDDs are very cheap though, they are obviously the best choice in all situations, with $\mathsf{Algo}_{\mathsf{STL/HDD}}$ for moderate table size, and with $\mathsf{Algo}_{\mathsf{DLU/HDD}}$ for larger tables. For a very small range, $\mathsf{Algo}_{\mathsf{STL/SSD}}$ performs better, which means that $\mathsf{Algo}_{\mathsf{STL}}$ is not very cost-effective on SSD. For large tables, and SSD at a moderate price compared to HDD, $\mathsf{Algo}_{\mathsf{DLU/SSD}}$ is better, which means that an SSD should be preferred with $\mathsf{Algo}_{\mathsf{DLU}}$.

The case of a large problem space, shown in Figure 4.11b, shows that $\mathsf{Algo}_{\mathsf{STL/SSD}}$ is never a good option, which confirms that the use of SSD advantages $\mathsf{Algo}_{\mathsf{DLU}}$. Therefore, the choice is restricted to the type of external memory in terms of the difference of prices, since for most part, $\mathsf{Algo}_{\mathsf{DLU}}$ will be used.

4.6.5 Discussion

It is difficult to conclude unequivocally on $\mathsf{Algo}_{\mathsf{STL}}$ and $\mathsf{Algo}_{\mathsf{DLU}}$, as their respective performances highly depend on parameters. It can however be observed that $\mathsf{Algo}_{\mathsf{DLU}}$ typically outperforms $\mathsf{Algo}_{\mathsf{STL}}$ on large problems, and when the one-way function is expensive. Also, it can be noted that the seek time is of crucial importance for $\mathsf{Algo}_{\mathsf{DLU}}$, which performs poorly compared to $\mathsf{Algo}_{\mathsf{STL}}$ on devices (such as hard disk) with slow seek time but high sequential read performance.

The various comparisons done between $\mathsf{Algo}_{\mathsf{DLU}}$ and $\mathsf{Algo}_{\mathsf{STL}}$ show that many parameters influence the choice of the algorithm and memory type to use, and it is difficult to make a simple judgment as to which is best. These parameters include problem parameters (N, M, c, ℓ) and machine/technology



(a) $N = 52^7 \approx 2^{40}$ **(b)** $N = 2^{56}$

Figure 4.11 – Comparison of the best algorithm between SSD and HDD

parameters $(\beta, \tau_F, \tau_L, \tau_S)$. A few observations can be made however:

- $\mathsf{Algo}_{\mathsf{DLU}}$ performs best on larger problem spaces than $\mathsf{Algo}_{\mathsf{STL}}$,
- $\mathsf{Algo}_{\mathsf{DLU}}$ performs best on slower hash functions than $\mathsf{Algo}_{\mathsf{STL}}$,
- $\mathsf{Algo}_{\mathsf{STL}}$ handles the use of slower memories such as HDD better than $\mathsf{Algo}_{\mathsf{DLU}}$,
- in many scenarios, a large portion of where $\mathsf{Algo}_{\mathsf{STL}}$ is most appropriate is *not* in its optimal configuration,
- in some scenarios, the region where $\mathsf{Algo}_{\mathsf{STL}}$ is most appropriate is also close to the typical memory size.

Limitations of the Analysis. The analysis of Section 4.6 was based on [KHP13] and does not consider optimizations such as, *e.g.*, checkpoints, endpoint truncation, or compressed delta encoding for chain storage which were discussed in Chapter 3. Similarly, it does not consider optimizations exploiting the architecture, such as loading sub-tables while computing chains in $\mathsf{Algo}_{\mathsf{STL}}$.

Including these optimizations would make the analysis much more complex, and we believe that taking them into consideration would not change our conclusions, since they only apply to the computation part of the online phase, and are therefore equivalent to a faster one-way function. While some optimizations might favor one algorithm over the other, it is very unlikely that the frontiers between regions of the best performances would shift significantly.

4.7 Experimental results

Our objective is now to analyse the behaviour of $\mathsf{Algo}_{\mathsf{DLU}}$ in practice. The analysis and validation of $\mathsf{Algo}_{\mathsf{STL}}$ was previously done in [KHP13], and is not considered here. We have set up experiments in order to validate the analytical results described in Section 4.6. The formulas established in Section 4.6 assume that $\mathsf{Algo}_{\mathsf{DLU/SSD}}$ and $\mathsf{Algo}_{\mathsf{DLU/HDD}}$ do not use RAM at all. We show that, in reality, these algorithms actually do use RAM because operating systems use cache techniques to speed up lookups. Thus, a value read from an external memory is temporarily saved in cache to prevent (to some extent) from accessing the external memory again to get the same value. As a consequence, the results provided in Section 4.6 correspond to upper bounds in practice. We refine the formulas to take the caching effect into account, and we then show that the refined formulas describe the experimental results more accurately.

4.7.1 Parameters and methodology

We have conducted the experiments on two problems of size $N = 2^{31}$ and 2^{36}, which allowed us to pre-compute the matrices in a reasonable time frame. For each problem, $\ell = 4$ tables were computed with a matrix stopping constant of $\bar{c} = 1.92$, giving $P \approx 0.999$.

In order to get results for varying amount of memory, we have computed several TMTOs using these parameters, but their tables were constructed from chains of different lengths. This was done by first computing a TMTO matrix for $t = 100$, and producing perfect tables from it. Then, we reused the perfect matrix, as a base to compute a second TMTO, by extending the chains from $t = 100$ to $t = 200$, and then again producing perfect tables from it. The process was reiterated until $t = 900$. The matrices of the experiment are illustrated in Figure 4.12, with each matrix filled differently. The fact that we use a fixed matrix stopping constant, problem size, and a fixed set of chain length implies that the values of the m parameter are determined by the TMTO curve and are different for all the tables.

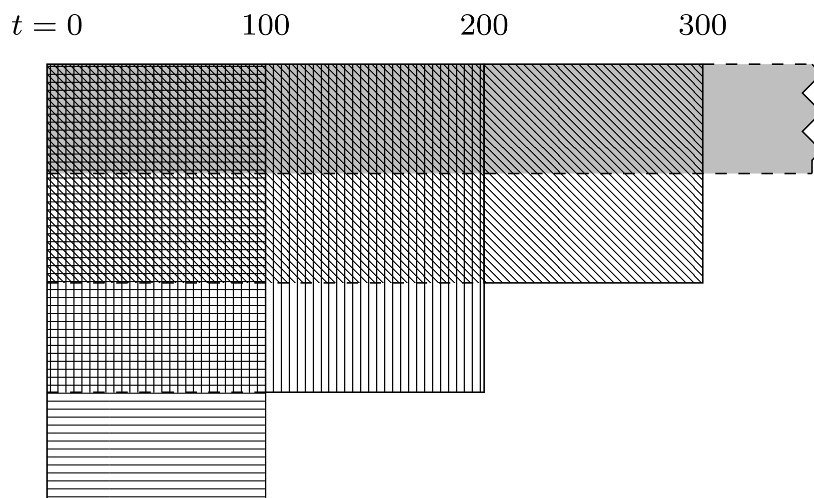


Figure 4.12 – TMTO matrix with intermediate processing of tables

We then obtained TMTOs with the parameter t taking values in $\{100, 200, \ldots, 900\}$. For the 2^{36}-problem, the pre-computation of the full matrix took 5 hours on 400 processor cores. Sorting the ending

points and removing the duplicated ones for every TMTO required a couple of days, due to the network latencies. Each (starting point, ending point) pair was stored on 8 and 16 bytes for $N = 2^{31}$ and $N = 2^{36}$, respectively. The size of the obtained tables are given in Table 4.2.

t	100	200	300	400	500	600	700	800	900
$N = 2^{31}$	624	320	215	162	109	130	96	82	73
$N = 2^{36}$	14935	8717	6156	4758	3878	3273	2831	2494	2229

Table 4.2 – Size of each of the computed tables (in MB)

We now discuss the methodology of the measurements. To evaluate the average running time of $\mathsf{Algo}_{\mathsf{DLU}}$, we average the measured attack time for the hashes of 1000 randomly-generated values in the problem space. As in Section 4.5, the timings are based on the processor time stamp counter (RDTSC). In order to keep the experiments independent, the environment is reset before each test. Indeed, there are side effects to be expected due to the way the operating system handles files, some of which also affect the measurements themselves. We discuss them below.

4.7.2 Paging and caching mechanisms

For every access to data stored on external memory, the full page containing the data is actually returned to the operating system. Due to the *caching mechanism*, the data is not fetched directly from the external memory every time we perform a lookup. Instead, the page containing the data is first copied from the external memory to the internal memory, and only then it can be read. Such a mechanism allows the system to speed up memory access: as long as the internal memory is not required for another use, and the RAM containing the cached page is not allocated by another process, the content of the page remains in it. This means that, if the same page is accessed again, it can be retrieved directly from the internal memory instead of waiting for the external memory.

If several lookups are performed on values that are located close enough in the external memory, then only the earliest lookup will require accessing the external memory. This phenomenon happens when a lookup is performed in the dichotomic search. As a consequence, at some point, every external memory access fetches elements that are located in the same page.

The problem of searching for an endpoint in the table then becomes a search of the page containing this endpoint. Each page contains B rows of a table, so such a search corresponds to a dichotomic search tree that is $\log_2 B$ levels shallower than the original search on endpoints. Thus, each lookup consists of $\log_2 m - \log_2 B$ page loads instead of $\log_2 m$, since the subsequent searches in a same page require no time per the assumption that RAM access times are negligible.

Taking paging and caching mechanisms into account, the formula of Theorem 4.1 of Section 4.4 can be refined to yield Equation (4.1). The $\mathsf{Algo}_{\mathsf{DLU}}$ algorithm's average wall-clock becomes

$$\tilde{T}_{\mathsf{DLU}} = \gamma \frac{N^2}{M^2} \tau_F + \frac{N}{m} (\log_2 m - \log_2 B) \cdot \tau_S. \tag{4.1}$$

Since the t is fixed to a small value in our experiment, it is not clear how much of the online time is spent on lookup, and how much of it on chain generation. Using Equation (4.1), we can infer the behaviour of this ratio. Keeping the success rate constant, we increase the chain length, while decreasing the amount of memory accordingly. We present in Figure 4.13 the evolution of the proportion of the

online time that is dedicated to the computation of the one-way function, either for the generation of the potential chains, or the verification of the endpoints' matches due to false alarms. The parameters used are those of the $N = 2^{36}$ problem.

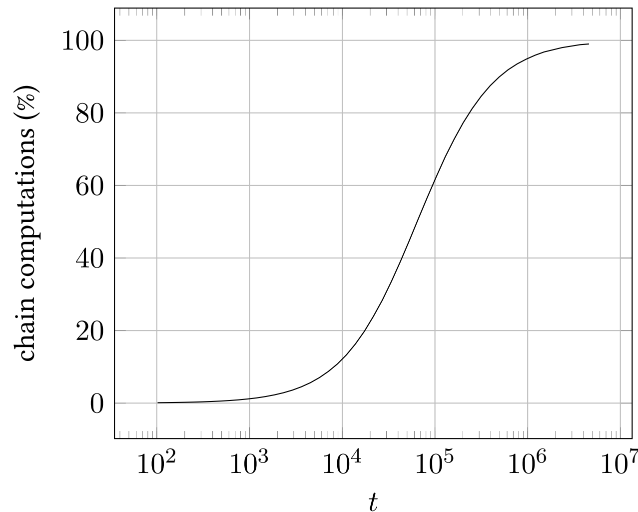


Figure 4.13 – Proportion of the online time spent computing the one-way function

We can see that we would need to increase the chain length by several orders of magnitude before the amount of computation becomes non-negligible. When reaching a large enough t, *i.e.*, of several millions, we somewhat fall back into the RAM model: the tables are not large enough anymore to induce a sensible amount of lookup time, and therefore the time spent computing the chains is approximately the time of the full online phase.

4.7.3 Reducing the caching impact

To get proper experimental results and bypass the operating system's built-in caching mechanism, we use a cache eviction technique and restrain the internal memory allocated to the program.

Every page that remains in memory after each experiment needs to be reset to a blank state. We use the `madvise` system call to tell the kernel that no additional pages are required. Although the kernel theoretically could ignore the setting and keep the pages in memory anyway, it does not most of the time, and in particular, such an event did not happen in our experiments. Alternative methods exist, such as requesting a cache drop, but they might affect the experiments by wiping data needed by the operating system.

We also restrain the internal memory that the program can use to a few megabytes, using the `cgroup` kernel subsystem. Software limitation was used instead of physically limiting the internal memory, because physical limitations may cause the operating system to starve for memory, which would greatly affect the results.

4.7.4 Results

The obtained results are presented in Figure 4.14 for $N = 2^{31}$, and in Figure 4.15 for $N = 2^{36}$. As expected, our implementation of $\mathsf{Algo}_{\mathsf{DLU/RAM}}$, which mainly depends on τ_F, closely follows the curve

given by the adjusted formula in Equation (4.1). We remark that the larger the tables are, the more accurate our model is.

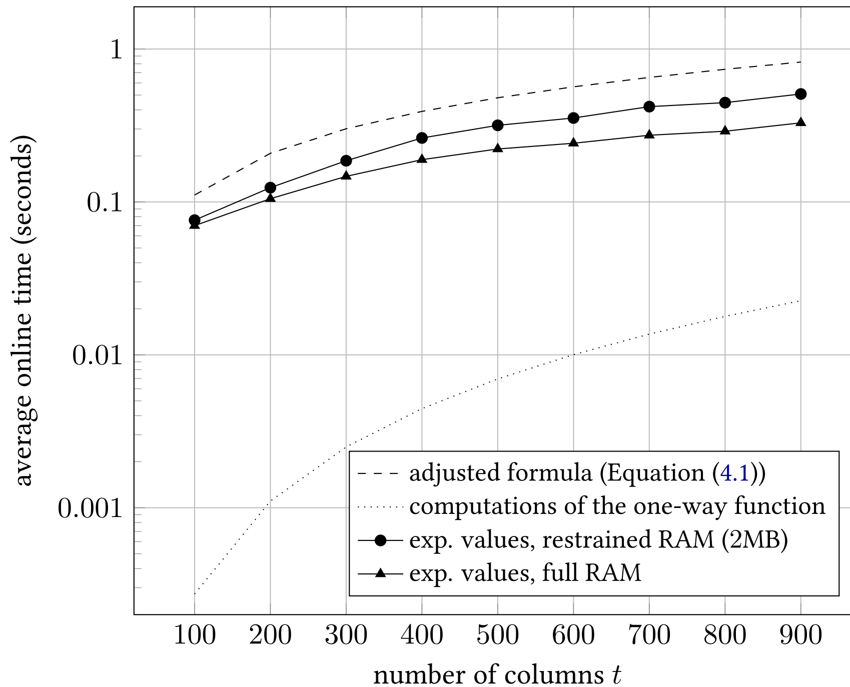


Figure 4.14 – Average online time for $N = 2^{31}$

The amount of time that is spent computing the potential chains is presented on both figures for reference. We used a modest t in the experiments, which translates into the fact that most of the online time is spent looking up endpoints in the table, and the computation of the one-way function is in fact negligible here.

This allows us to observe the impact of external memory when the problem size increases. Although the problem space is 32 times larger in the second experiment, t is fixed to the same value in both experiments, so the online time would stay the same in the RAM model, because the larger problem space is compensated by a larger amount of memory. However, we observe that, between Figure 4.14 and Figure 4.15, the online time has approximately doubled.

Some caching can still be noticed on the curves, but trying to restrain RAM even further resulted in failures which could be explained by the fact that there is no distinction for the OS between file caching and the caching of the executable itself. Thus, we have to over-provision the RAM to be certain that the program itself does not starve during the execution. Nevertheless, we observe that the experimental curve is below the analytic dashed-line curve. This confirms that Equation (4.1) is a more accurate upper bound to the practical wall-clock time of $\mathsf{Algo}_{\mathsf{DLU}}$ than the formula of Theorem 4.1.

Finally, we remark that even though we have restrained RAM drastically, we can still notice a gain in time due to the caching done by the OS. The zones concerning $\mathsf{Algo}_{\mathsf{DLU}}$ in figures of Section 4.6 would therefore have larger areas, in practice. The impact of the native caching operated by the OS is difficult to predict with exact precision, though. It shows that the analysis is close to the reality, but is in fact pessimistic due to caching in RAM. Exploiting the RAM in addition to external memory might give an extra edge to $\mathsf{Algo}_{\mathsf{DLU}}$.

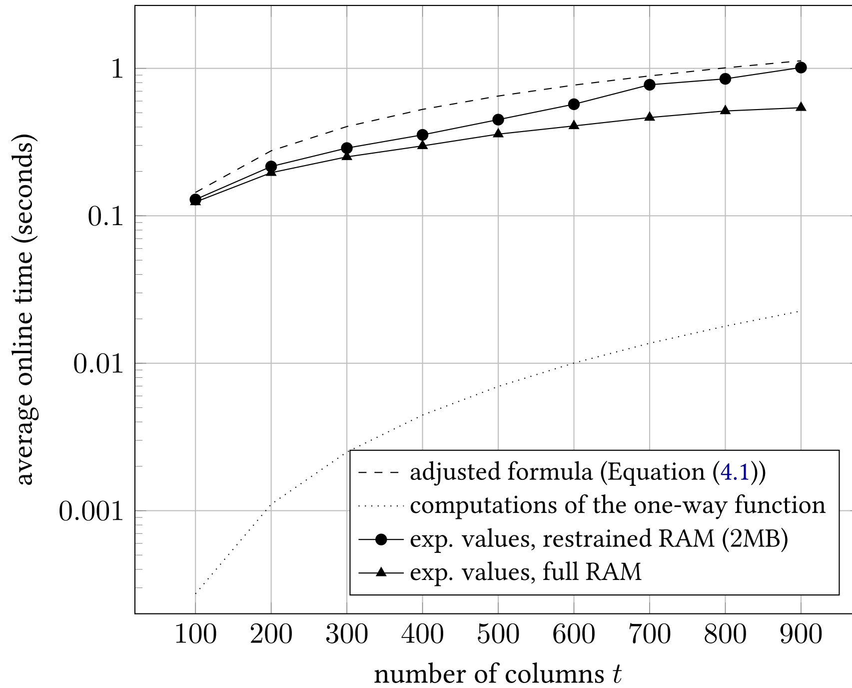


Figure 4.15 – Average online time for $N = 2^{36}$

4.8 Conclusion

In this work, we explored the use of external memory for TMTOs with large memory requirements. Two approaches were compared: the first one relying on the Algo_{STL} algorithm, presented in [KHP13] and used in existing implementations, which takes advantage of RAM by processing sub-tables in internal memory; and the second one, based on the Algo_{DLU} algorithm, which uses the standard RAM-based rainbow table algorithm directly on external memory.

We evaluated the two algorithms and compared their efficiency on different memory types and different problem parameters. Conclusions are subject to parameters, but several major observations were made. Algo_{DLU} performs better than Algo_{STL} on larger problem spaces, slower hash functions, and faster memories. At the very least, we cannot say that Algo_{STL} is unequivocally more efficient, contrarily to what was previously believed by the implementers, and stated in [KHP13].

Nevertheless, our results and observations are based on the measurements given in Section 4.5. Using clusters of many disks, such as, *e.g.*, redundant arrays of inexpensive disks (RAID), to reach high quantities of memory might affect τ_S and τ_L enough that the conclusions would be different. Costs of the various memory types were not considered formally in our analysis because of the great variability in prices and setups. The evolution of storage prices, and especially with NVM technology, is such that any given comparison would be obsolete in a matter of months. Such prices and comparisons might help an implementer make an informed decision regarding the memory type to use.

Finally, we implemented Algo_{DLU} and corroborated its efficiency analysis (analysis and validation of Algo_{STL} was previously done in [KHP13]). Our implementation shows that the analysis results are accurate with regard to the reality, but are in fact pessimistic due to caching in RAM. Exploiting the RAM in addition to external memory might give an extra edge to Algo_{DLU}. Optimizations on Algo_{STL}

might exist as well, such as computing chains and loading tables in parallel.

The work in this chapter highlighted that the classical algorithm that was introduced in Section 2.4 of Chapter 2 is actually relevant in external memory when considering the usage of modern hardware to store the tables. We will therefore focus on this algorithm for the rainbow trade-off when performing comparisons of the trade-off in Chapter 6.

References

[AAA+02] N. R. Adiga et al. *An Overview of the BlueGene/L Supercomputer.* In: *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. Nov. 2002, pp. 60–60 (cit. on p. 93).

[Aut06] Unknown Author. *Free Rainbow Table.* Accessed: Jan 2019. 2006. Link. (Cit. on pp. 98, 105, 107).

[AV88] Alok Aggarwal and Jeffrey Scott Vitter. "The Input/Output Complexity of Sorting and Related Problems". In: *Communications of the ACM* 31.9 (1988), pp. 1116–1127 (cit. on p. 96).

[BGB17] Matias Bjørling, Javier González, and Philippe Bonnet. *LightNVM: The Linux Open-Channel {SSD} Subsystem.* In: *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17).* 2017, pp. 359–374 (cit. on p. 107).

[CH12] Danny Cobb and Amber Huffman. *NVM express and the PCI express SSD revolution.* Intel, 2012 (cit. on p. 106).

[Cor11] Intel Corporation. *Intel® 64 and ia-32 architectures software developer's manual.* 2011 (cit. on p. 106).

[Cor13] Intel Corporation. *Intel®SHA Extensions.* Accessed: Jan 2019. 2013. Link. (Cit. on p. 95).

[CR72] Stephen A. Cook and Robert A. Reckhow. *Time-Bounded Random Access Machines.* In: *STOC.* ACM, 1972, pp. 73–80 (cit. on p. 94).

[ER64] Calvin C. Elgot and Abraham Robinson. "Random-Access Stored-Program Machines, an Approach to Programming Languages". In: *Journal of the ACM* 11.4 (1964), pp. 365–399 (cit. on p. 94).

[KHP13] Jung Woo Kim, Jin Hong, and Kunsoo Park. *Analysis of the Rainbow Tradeoff Algorithm Used in Practice.* Cryptology ePrint Archive, Report 2013/591. `http://eprint.iacr.org/2013/591`. 2013 (cit. on pp. 97–101, 103–106, 109–111, 116, 117, 121, 184).

[KPP+06] Sandeep Kumar, Christof Paar, Jan Pelzl, Gerd Pfeiffer, and Manfred Schimmler. *Breaking Ciphers with COPACOBANA - A Cost-Optimized Parallel Code Breaker.* In: *CHES 2006.* Ed. by Louis Goubin and Mitsuru Matsui. Vol. 4249. LNCS. Yokohama, Japan: Springer, Heidelberg, Germany, Oct. 2006, pp. 101–118 (cit. on p. 97).

[Oec03] Philippe Oechslin. *Making a Faster Cryptanalytic Time-Memory Trade-Off.* In: *CRYPTO 2003.* Ed. by Dan Boneh. Vol. 2729. LNCS. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 2003, pp. 617–630 (cit. on pp. 5, 35, 36, 56, 107, 153, 156, 157).

[Oec17] Philippe Oechslin. *Objectif Sécurité.* Accessed: Jul 2019. 2017. Link. (Cit. on p. 93).

[Shu09] Zhu Shuanglei. *RainbowCrack.* Accessed: Apr 2017. 2009. Link. (Cit. on pp. 7, 93, 187).

[Spi07] Stefan Spitz. *Time Memory Tradeoff Implementation on Copacobana.* MA thesis. Bochum, Germany: Ruhr-Universität Bochum, June 2007 (cit. on pp. 97, 98).

[TO05] Cedric Tissières and Philippe Oechslin. *Ophcrack*. Accessed: Apr 2017. 2005. Link. (Cit. on pp. 7, 35, 56, 93, 99, 162).

[TSG11] Guangming Tan, Vugranam C. Sreedhar, and Guang R. Gao. "Analysis and performance results of computing betweenness centrality on IBM Cyclops64". In: *The Journal of Supercomputing* 56.1 (Apr. 2011), pp. 1–24. ISSN: 1573-0484. Link. (Cit. on p. 93).

[WNS14] Martin Westergaard, James Nobis, and Zhu Shuanglei. *Rcracki-mt*. Accessed: Apr 2017. 2014. Link. (Cit. on pp. 98, 105).

[WW09a] Pinchas Weisberg and Yair Wiseman. *Using 4KB Page Size for Virtual Memory is Obsolete*. In: *Proceedings of the IEEE International Conference on Information Reuse and Integration – IRI 2009*. Las Vegas, Nevada, USA: IEEE Systems, Man, and Cybernetics Society, Aug. 2009, pp. 262–265 (cit. on p. 97).

[WW09b] Pinchas Weisberg and Yair Wiseman. *Using 4KB Page Size for Virtual Memory is Obsolete*. In: *Proceedings of the IEEE International Conference on Information Reuse and Integration, IRI 2009, 10-12 August 2009, Las Vegas, Nevada, USA*. 2009, pp. 262–265 (cit. on p. 110).

[YLL+11] Xue-Jun Yang, Xiang-Ke Liao, Kai Lu, Qing-Feng Hu, Jun-Qiang Song, and Jin-Shu Su. "The TianHe-1A Supercomputer: Its Hardware and Software". In: *Journal of Computer Science and Technology* 26.3 (May 2011), pp. 344–351. ISSN: 1860-4749 (cit. on p. 93).

[YSU+11] M. Yokokawa, F. Shoji, A. Uno, M. Kurokawa, and T. Watanabe. *The K computer: Japanese next-generation supercomputer development project*. In: *IEEE/ACM International Symposium on Low Power Electronics and Design*. Nov. 2011, pp. 371–372 (cit. on p. 93).

People have to be free to investigate computer security. People have to be free to look for the vulnerabilities and create proof of concept code to show that they are true vulnerabilities in order for us to secure our systems.

Edward Snowden

MPHF-based rainbow tables 5

Minimal perfect hash functions are a special kind of hash functions which provide a bijective mapping in a chosen set with a very low memory footprint. In this chapter, we introduce a novel technique to perform TMTOs involving these hash functions: the MPHF-based trade-off.

Contents

5.1 Introduction **126**
- 5.1.1 Rationale 126
- 5.1.2 Definitions 127

5.2 Related work on MPHFs **128**
- 5.2.1 Hash functions and hash tables 128
- 5.2.2 Construction of perfect hash functions 129

5.3 MPHF-based perfect rainbow trade-off **134**
- 5.3.1 Offline phase 134
- 5.3.2 Online phase 135
- 5.3.3 Signature 136

5.4 Analysis **137**
- 5.4.1 Pre-computation overhead 137
- 5.4.2 Online running time 138
- 5.4.3 Storage requirements 139
- 5.4.4 Comparison with the truncated point technique 139

5.5 Optimal parameters **141**
- 5.5.1 MPHF parameters 141
- 5.5.2 Optimal signature 142

5.6 Experimental validation **143**
- 5.6.1 Pre-computation 144
- 5.6.2 Online time performance 144

5.7 Conclusion **146**

5.1 Introduction

We saw in Section 3.2 that reducing the amount of memory required to store a TMTO allowed us to improve the performance, given that the saved memory can be used to store more chains. Several techniques aimed at reducing the storage required for the endpoints: these improvements assumed that endpoints were a necessary part of the table, and focused on how to compress them. In [ACFT19], we introduce a technique that no longer requires storing the endpoints.

Minimal perfect hash functions (MPHFs) are a tool that associates a small integer to each element of a chosen set, and that can be queried in constant time. Furthermore, an MPHF memory requirement does not depend on the size of the elements, only on their number, and the memory requirement per element is only of a few bits. This makes MPHF an interesting data structure to store indexes. We exploit the characteristics of MPHFs to no longer need to store the endpoints in the tables. We do so, during the online phase, by using MPHFs to perform the association between potential endpoints and starting points.

The technique introduced in [ACFT19] revisits the assumptions about the role of the endpoint in the online phase and provides a new table structure which is more space-efficient than a table with the same TMTO parameters in the classical rainbow trade-off. An analysis our new technique is presented. We show that this technique is similar to the classical rainbow technique with truncated endpoints, and perform a comparison which show that, for the same overhead, the MPHF requires less storage. Finally, an experimental validation of our analysis is proposed.

5.1.1 Rationale

We begin by briefly describing the key points of our MPHF technique. In the online phase of all TMTO variants, there is a step which requires to associate the endpoint with its corresponding starting point. In the existing online phase algorithms, this step comes “for free” in the sense that the tables are constructed in such a way that a corresponding starting point and endpoint are stored at the same index. Therefore, it is the structure of the table that provides the required link. Improvements such as compressed delta-encoding and prefix-suffix decomposition add a layer of complexity to the table structure. However, the fact remains that the starting point and the endpoint are linked by their position in the table.

If associating endpoints with starting points forms a separate step, the role of the endpoint becomes twofold. Its first purpose it to determine whether a given potential chain’s end matches the end of a chain in the matrix. This role usually corresponds to a membership query of the potential endpoint in the set of endpoints. Its second role is to allow the computation of the verification chain, by yielding the starting point that corresponds that belongs to the same chain.

In the technique of [ACFT19], we propose to use MPHFs in order to perform an association between endpoints and starting points, by hashing potential endpoints. Using MPHFs allows us to discard the endpoint from the table, and use an endpoint’s signature instead to perform the check of whether a given endpoint is potentially in the table. We show that the amount of data from the endpoint that is required for this check is actually lower than the one required for the association step, in such a way that our technique performs better than the truncated endpoint technique, even when taking the storage of the MPHF into account.

Since the association is made by the MPHF, which can only give a single associated starting point per endpoint, the MPHF scheme is naturally more suited for perfect versions of TMTOs. We choose to introduce our technique applied to the rainbow trade-off. Theoretically, while it may be possible to

apply our technique on the DP trade-off and its variant, the fuzzy trade-off, these trade-offs may not benefit much from the reduction in size allowed by the MPHFs. In particular, the DP trade-off is less efficient in terms of memory required, for a given success rate. We saw in Section 2.2, that the perfect version of the Hellman trade-off was not practical, so it is not considered either.

5.1.2 Definitions

We introduce additional definitions and vocabulary for the hash function concepts that will be used throughout this chapter. We first define what we mean by hash function. Note that a special case of hash functions was already defined in Definition 2.10, for the specific purpose of cryptographically secure hashing. The definition below is more general.

Definition 5.1 (Hash function). *Let* U *be a set called the* universe*, and* $\mathbb{Z}/m\mathbb{Z}$ *the set of residues modulo* $m \in \mathbb{N}$, e.g.*, the range* $\{0, \ldots, m-1\}$*. A hash function is a function of the form*

$$h : U \to \mathbb{Z}/m\mathbb{Z}.$$

The input set U, in most applications, is either a set $\{0,1\}^l$ for some $l > m$, the character string set Σ^* for some alphabet Σ, or $\mathbb{N}$. In the literature, the elements of U are often called *keys*, or *keywords* when they are strings.

We now introduce the concept of perfect hash functions.

Definition 5.2. *Given a hash function* $h : U \to \mathbb{Z}/m\mathbb{Z}$*, and a set* $I \subsetneq U$ *of size* $n < m$*,* h *is said to be a* perfect *hash function (PHF) on* I*, if the restriction of* h *on* I

$$h_{|I} : I \to \mathbb{Z}/m\mathbb{Z}$$

is injective.

An illustration of a PHF is given in Figure 5.1. Notice that, while only the space I is of interest, the PHF is defined on a larger set U. Therefore, the images that are not attributed to any element of I can still be attributed to an element of $U \setminus I$.

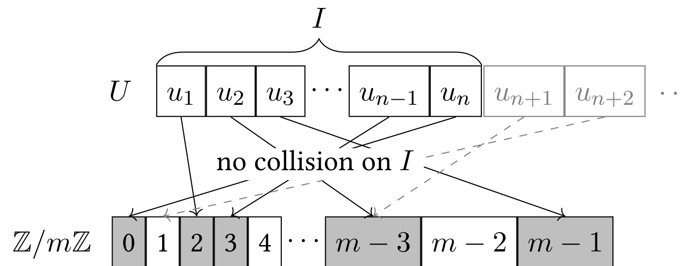


Figure 5.1 – Perfect hash function

Since we focus on the characteristics of $h(I)$, we will often disregard the fact that h is defined on U and simply say that $h : I \to \mathbb{Z}/m\mathbb{Z}$ is a perfect hash function. A further improvement of PHFs is to try to get a distinct image for every input elements, and fill the image space, in other words to be both injective on I and surjective. This gives a minimal perfect hash function.

Definition 5.3. *Let $h : I \to \mathbb{Z}/m\mathbb{Z}$ be a perfect hash function, and $n = |I|$. When $m = n$, h is said to be a* minimal *perfect hash function (MPHF).*

A MPHF is illustrated in Figure 5.2. All the elements in the range $\{0, \ldots, n-1\}$ are now linked to by exactly one element of I. Once again, elements of $U \setminus I$ also map randomly to this range.

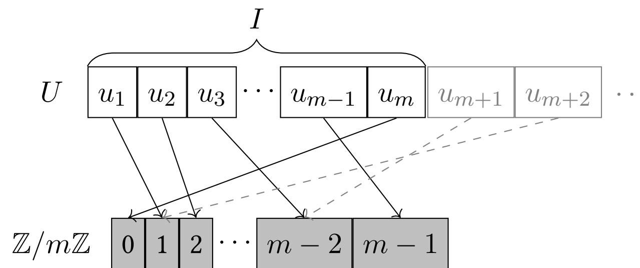


Figure 5.2 – Minimal perfect hash function

According to the context of use, additional properties on the distribution of the hash values, can be required by a MPHF. Most of them are properties that were already defined in Chapter 2 for both one-way functions (Definition 2.3) or cryptographic hash functions (Definition 2.10). In particular, MPHFs can be pseudo-random.

Note that the hash functions considered in this chapter are not necessarily cryptographically secure. In particular, the output space of MPHFs is of arbitrary size, and often too small to guarantee that the birthday paradox will not happen with high probability.

It can sometimes be useful for an MPHF to keep the structure of the input space. Such functions are called order-preserving.

Definition 5.4. *A minimal perfect hash function $h : I \to \mathbb{Z}/m\mathbb{Z}$ is said to be* order-preserving*, if*

$$\forall (x_1, x_2) \in I^2, \quad x_1 \lessdot x_2 \implies h(x_1) < h(x_2),$$

where $\lessdot$ is an order on I.

5.2 Related work on MPHFs

Since our novel technique heavily relies on the MPHF concept, and requires us to build one during the offline phase, before presenting our technique, we overview the related work on hash functions, with a particular focus on the evolution of perfect hashing in the scientific literature. We introduce the relative concept of hash tables, and we give the major results on (minimal) perfect hash functions in a constructive way to progressively build an intuition on how the modern MPHF schemes work.

5.2.1 Hash functions and hash tables

According to Knuth [Knu73], the idea of hashing appears within IBM in 1953. The first clear usage of the technique is made by Dumey in [Dum56]. There is no mentioning of “hashing” in the literature, until in the work of Hellerman in 1967 [Hel67], even though the term was in use throughout the 60s. The technique is then used as a mean to more easily locate keywords of machine assembly language

in tables. With a hash function h, keywords are ordered in tables in a way that any w is at index $h(w)$ in the table. Hash functions are then used as a tool to get the address of a keyword faster than with a linear search. Such tables associated with a hash function constitute fast retrieval data structures named *hash tables*. Hash tables can for instance be used, *e.g.*, to represent mathematical sets, in such a way that it is very efficient to determine the membership in the set of any given element, since a query with a hash function takes much less time compared to a linear search in a classical array.

Let $I \subset U$ be a set of n keys which are stored in a hash table of size m:

$$I = \{w_1, w_2, \ldots, w_n\}.$$

When filling in the table, a collision of h may happen, so there might be two elements that have to be stored at the index $h(w_i)$. Indexes of a hash are usually called *buckets* for this reason. The hash table therefore needs to handle these collisions. A classical method consists in storing the head of a linked list in the table, in a way that when an index is accessed, the list is browsed linearly and each element of the list is read until the right element is found. This technique, called *chaining*, has been in use since the first implementations of hash tables, and is probably the first documented usage of linked lists [Knu73]. The chaining method influences the running time of the hash table. At the extreme, if all elements are in the same bucket, the hash table query time complexity will be $O(n)$ instead of $O(1)$. An important metric with hash tables is the ratio between the number of elements and the size of the table.

Definition 5.5. *We consider a hash table of size m, containing n elements. The ratio given by $\beta = \frac{n}{m}$ is called the* loading factor *of the hash table.*

The hash table provides faster retrieval of data when there is only a few elements in each bucket. This is less likely to happen when we have a loading factor $\beta \ll 1$. When $\beta \approx 1$, the number of elements in each bucket will also be low on average, if we suppose that the hash function is pseudo-random. Nevertheless, this performance comes at the expense of a worst storage efficiency of the table, as empty buckets still take unused space.

The concept of hash tables is tangled with the one of hash functions, in such a way that the improvements of hash functions were often made with the goal of improving hash tables. We will see in the following that some more complex MPHFs rely on hash tables internally.

5.2.2 Construction of perfect hash functions

We now focus on works that deal with perfect hashing. When the set I of a given hash table is fixed and known beforehand, hash functions can be improved to optimise the distribution of the values in the table. In particular, a hash table can be made in a way that there is at most one element per index. When this is the case, the retrieval is in $O(1)$, since no chaining is required during the application of the hash function. The associated function of such hash tables are perfect hash functions.

Perfect hash functions are first used by Greniewski and Turski in [GT63], in their "characteristic function method", which aims at translating code words of the KLIPA language to numerical values efficiently. The first functions are constructed manually as the size of the considered set of keywords is very small.

Sprugnoli performs a first PHF analysis in [Spr77], where he establishes two techniques to build such functions: the quotient reduction and the remainder reduction. Both these techniques assume that $I \subset \mathbb{N}$, and that it is ordered, *i.e.*, for all $i > j$, $w_i > w_j$. A hash function h using quotient reduction is

of simple form:

$$\forall w \in I, \quad h(w) = \left\lfloor \frac{w+s}{N} \right\rfloor,$$

for some divisor $N \in \mathbb{N}$, and $s \in \mathbb{N}$ dependent on I. In order to get a full table, the best approach is to use $N \approx \frac{(w_n - w_1)}{n-1}$, but if I is non-uniform, the table is sparse. A "cut" technique is introduced to reduce this drawback: all elements above a certain threshold get hashed differently, so they can fill the lower indexes of the table. With the cut, we obtain *remainder functions* which are such that

$$\exists (d,q) \in \mathbb{N}^{*2}, \quad \forall w \in I, \quad h(w) = \left\lfloor \frac{(d+wq) \mod M}{N} \right\rfloor,$$

where $N \in \mathbb{N}$, and $M \in \mathbb{N}$ is not a divisor of $(w_n - w_1)$. The parameters d and q are found using an exhaustive search, so that h is indeed a PHF. While this technique provides a slower hash function than the quotient technique due to the additional modulo operation, it is more collision resistant against non-uniform sets I. Furthermore, it is possible to lower the query time by choosing N of the form 2^i, for some $i \in \mathbb{N}^*$, so that the modulo operation becomes a shift. This second technique seems to be preferred in the literature, as subsequent improvements rely mainly on remainders for their PHF.

Carter and Wegman introduce in [CW77] and [CW79] the notion of *universal hashing*. This consists in building complex hash function from simpler basic functions chosen at random in families of pseudo-random hash functions, such as the remainder functions considered by Sprugnoli. The main goal of universal hashing is to reduce the number of collisions, even in the case of a very skewed key space. Consider the function of the following form given in [CW77]:

$$g(x) = h_1(x) + h_2(x) \mod n, \tag{5.1}$$

where h_1 and h_2 are arbitrary simple hash functions. If a hash function h_1 were to perform poorly with the set I, that is $|h_1(I)| \ll n$, it is unlikely that a second hash function h_2 chosen at random would suffer from the same problem. Hence, the hash function g, which mingles the two functions, would still be pseudo-random. While perfect hashing is not considered in the paper, the construction of a hash function such as the one in Equation 5.1 is reused in later literature.

In [Cic80], Cichelli proposes an algorithm to build an MPHF based on the form of Equation 5.1, in practical time. The use case is a cross-reference program which deals with the keywords that are found in the Pascal programming language. Given a keyword l_i, for $i \in \{0, \dots, n-1\}$, we denote with $l_i[k]$ the kth character of the keyword. Cichelli's MPHF are of the following form:

$$\forall l_i \in I, \quad h(l_i) = \mathsf{length}(l_i) + v(l_i[0]) + v(l_i[\mathsf{length}(l_i) - 1]), \tag{5.2}$$

where $\mathsf{length} : I \to \mathbb{N}$ gives the length of a word, and v is a hash function.

A backtracking algorithm is used in [Cic80] to construct v, in such a way that the function h is an MPHF. While, in previous techniques, an exhaustive search is applied, the keys are ordered beforehand so that the search time is reduced. The author of [Cic80] claims a tenfold improvement with regard to [Spr77]. Nevertheless, the technique is only practical for a small set of keywords, because the algorithm is exponential in the number of keywords.

Fox, Heath, Chen and Datta note that the Cichelli's article introduces a general pattern for MPHF generation [FHCD92]. Cichelli's algorithm follows 3 phases abbreviated by MOS (for mapping, ordering and searching) in [FHCD92]):

1. **Mapping**, where the keys of I are translated by some simple operation, often affine.
2. **Ordering**, where the keys are rearranged to determine the order in which they get values.
3. **Searching**, where the hash values are assigned to each key.

It can be observed that this template is indeed followed by most of the literature on MPHF from then on, with a focus on improving the Ordering phase.

Up to this point, the different MPHF techniques were not guaranteed to produce MPHFs for all the possible sets I. Alternatively, for PHF construction algorithms, which are known to terminate with a loading factor $\beta < 1$, we do not have the same guarantee as with $\beta = 1$. More generally, it was not known if the construction of an MPHF for an arbitrary set was always possible. Jaeschke introduces *reciprocal hashing* in [Jae81], which consists in building remainder functions in a way that the input w is in the denominator. He gives in [Jae81] an existence proof for such a form of MPHFs, The corresponding theorem which is reproduced in Theorem 5.1.

Theorem 5.1 (Existence of MPHF). *Given a key space $I \subset U$, where $|I| = n$, and a function h defined by*

$$\begin{aligned} h : I &\rightarrow \mathbb{Z}/m\mathbb{Z} \\ w &\mapsto \left\lfloor \frac{C}{Dw+E} \right\rfloor \mod n, \end{aligned}$$

there exists $(C, D, E) \in \mathbb{N}^3$ such that h is an MPHF.

The proof of Theorem 5.1 from [Jae81] is accompanied by an algorithm to determine the constants C, D and E. The idea of this algorithm is to try to produce MPHFs of the form

$$h : k \mapsto \left\lfloor \frac{C}{k_i} \right\rfloor \mod n. \tag{5.3}$$

If the keys $k_1, k_2, \ldots k_n$ are pairwise relatively prime, the Chinese remainders theorem (CRT) can yield a C such that this function is indeed an MPHF. When this is not the case, the set of keys is transformed by an affine function $f : x \mapsto Dx + E$, whose parameters are determined with an exhaustive search, so that the set $f(I)$ can yield such a constant C. The time complexity of the algorithm is exponential in $|I|$.

In [Cha84], Chang expands the results of [Jae81], and proposes the first polynomial algorithm to construct MPHFs which are order-preserving. He proposes much simpler functions than those of the form of Equation 5.3, by offloading the flexibility given by the D and E parameters on a more complex modulo. The functions proposed in [Cha84] are of the form:

$$h : w \mapsto C \mod p(w),$$

were p is a "prime number function", such that, given $a, b \in \mathbb{N}$ satisfying $a < b$, for all $x \in [a, b]$, $p(x)$ is prime, and p is strictly increasing, that is,

$$\forall (x_1, x_2) \in [a, b]^2 \quad \text{s.t.} \quad x_1 > x_2, \quad p(x_1) > p(x_2).$$

While Chang's algorithm avoids performing a costly ordering like the one in [Jae81], he does not provide a way to effectively find such functions p. No explicit complexity in time is given in [Jae81], but a quick analysis from Sager in [Sag85] mentions that the algorithm's running time is in $O(M^2 \log M)$,

where M denotes the number of keywords. Sager also notes that the cost in space of the resulting hash function is prohibitive, because of the huge size of the C constant in practice.

In [Sag85] Sager reuses the concept of universal hashing to propose the "minicycle" algorithm to build (not order-preserving) MPHFs, improving on the technique of Cichelli. If h_1, h_2 and h_3 are three pseudo-random hash functions, Sager's algorithm finds a function g such that

$$\forall w \in I, \quad h(w) = (h_0(w) + g \circ h_1(w) + g \circ h_2(w)) \mod n$$

is an MPHF. The minicycle algorithm is based on a modification of Cichelli's algorithm in [Cic80]. In this variant, the hash function uses two different functions v_1 and v_2 instead of one as done in Equation 5.2. Sager introduces a hypergraph[1] representation of the values which are connected in such a way that, for each k, $h_1(k)$ and $h_2(k)$ are linked by an edge. The minicycle algorithm is a backtracking ordering procedure which selects the best unselected edge belonging to as many small cycles as possible. The idea is to incrementally cover the whole graph, in a way that, at each step, an MPHF can be extracted from the current subset of keywords by dealing with a reduced search space. The algorithm of Sager is faster than Cichelli's algorithm for large sets, but still requires $O(n^4)$ time and $O(n^3)$ space for the backtracking, and $O(n)$ for the search of values, where n is the size of the key space I. Sager claims that he did not encounter sets which he was not able to build an MPHF on. No proof of the algorithm is given in [Sag85]. A later analysis by Fox et al. in [FHCD92] establishes that an MPHF built using this algorithm takes $n \log_2 n$ bits of storage in average.

Haggard and Karplus further generalise the Cichelli method in [HK86]. While they use universal hashing, they do not restrict the number of hashing function, as in [CW77; Cic80; Sag85]. They search for MPHFs of the form

$$\forall w \in I, \quad h(w) = \mathsf{length}(w) + \sum_i g_i \circ \sigma_i(w), \tag{5.4}$$

where the σ_i are "selector functions" which compute a value based on a letter position in a word. Hypergraphs are used similarly to what is done in [Sag85]. The best selector functions are the functions where no two words correspond to the same edge of the hypergraph. A backtracking procedure similar to Sager's minicycle algorithm is used on the hypergraphs. The article hints that some heuristics can reduce the time taken by the backtracking, and lists some of them, but no running time analysis of their algorithm is done in [HK86].

Fox, Chen, Heath and Datta also improve Sager's minicycle algorithm. They describe in [FHCD92] a two step algorithm which builds a spanning tree on a bipartite hypergraph constructed from the key space. This is done with Prim's algorithm[2] [Pri57]. The second part of the algorithm consists in ordering keys by edge multiplicity and by number of cycles using heuristics which are provided in [FHCD92]. This Fox et al. to achieve an ordering step in $O(n^3)$ operations, which improves on Sager's algorithm's $O(n^4)$ time complexity.

Mehlhorn performs in [Meh82] a theoretical analysis of PHFs. He states some lower bounds on the storage of (M)PHFs. One of the main results is the count of all the possible PHFs.

[1] The paper describes them as graphs, but then proceeds to build the edge set using a multiset, which is exactly the characteristic of a hypergraph.

[2] As Prim's algorithm works on weighted graphs, the hypergraph is transformed into a classical graph with the multiplicity of edges used as weights.

Theorem 5.2. *Consider hash functions of the form $h : U \to \mathbb{Z}/m\mathbb{Z}$, which is a PHF on some $I \subset U$. Let $u = |U|$ and $n = |I|$. The size of the family of all such possible PHFs is*

$$\frac{\binom{n}{u}}{\left(\frac{u}{m}\right)^n \binom{n}{m}}.$$

This result allows us to compute the average storage size of such a PHF by applying $\log_2$ on this result to get the size in bits. We consider the MPHF case. We then have $n = m$. Moreover, recall that the numerator can be seen as the number of n-combinations of elements in U, that is, the number of possible sequences of n elements, without considering the ordering:

$$\binom{n}{N} = \frac{N^n}{n!}.$$

This fact, coupled with Stirling's approximation, the result of Theorem 5.2, for MPHFs, results in

$$\frac{\binom{n}{N}}{\left(\frac{N}{n}\right)^n} = \frac{n^n}{n!} \approx \frac{e^n}{\sqrt{2\pi n}}.$$

Then the storage of an MPHF is expected to require at least

$$\log_2 \frac{e^n}{\sqrt{2\pi n}} = n \log_2 e - \log_2 \sqrt{2\pi n} \approx n \log_2 e$$

bits when n is not small. Note that this lower bound is independent of u, it only depends on the numbers of keys. Then we can reason on the storage required for each key of the MPHF which is therefore at least 1.44 bits per key. An algorithm is presented in [Meh82], alongside its analysis, which gives a time complexity of $O(n \log_2 n)$.

Several subsequent propositions from the literature aim at optimizing the storage required, while overall keeping the same algorithm structure. In [FKS82], Fredman, Komlós, and Szemerédi, assume that $I \subset \mathbb{N}$ and propose an alternative ordering step to the one of [Meh82], to provide an expected $O(n)$ time complexity, with an MPHF described in $n + O(n)$ bits. Their time complexity assumes a word RAM model, which is a variant of the RAM model introduced in Section 4.2.1, where accessing a word of memory takes a unit time.

In [SS90], Schmidt and Siegel show that the construction in [FKS82] can also be made linear in storage. Their variant is a PHF whose storage requires $O(n + \log \log m)$ bits, which translates to $O(n)$ for an MPHF. However, while having an $O(1)$ time complexity, their scheme is non-practical due to a large constant.

It is shown in [TY79] that with what is called the *displacement* technique, the depth of an existing hash table, *i.e.*, the maximum number of elements per bucket can be arbitrarily reduced, if one is willing to accept a slightly worse storage complexity. In [Pag99], Pagh combines the universal hash function concept such as used in [FKS82] and the displacement technique to generate hash functions of the form,

$$\forall w \in I, \quad h(w) = (f(x) + d(g(x))),$$

where f and g are functions randomly chosen from a family of universal hashing functions, and d is a displacement function. The obtained MPHF can be stored in $n(2 + \varepsilon) \log n$ bits. The expected time of such a construction is $O(n)$. Note that this scheme assumes that random functions of the universal

hashing can be found for free. The space complexity is improved by Dietzfelbinger and Hagerup in [DH01] to get to $O(n(1+\varepsilon)\log n)$ bits.

In [BPZ07], Botelho, Pagh, and Ziviani propose a new way to base the universal hash function on hypergraphs, which results in the first algorithm that is below superlinear space complexity. With their scheme, the resulting MPHF is stored on $2.62n$ bits, with a construction performed in time $O(n)$.

An algorithm called *Compress, Hash and Displace* (CHD) is introduced by Belazzougui, Botelho, and Dietzfelbinger in [BBD09]. It is a refinement of [BPZ07] which makes use of a variant of the displacement technique from [Pag99]. A first hash function is computed with universal hashing to constitute an intermediate hash table of capacity $r > n$. The values in the intermediate hash table are then reduced to the range $\{0, \dots, n-1\}$ by using a second hash function, in such a way that the composition of the two hash functions gives an MPHF. The space complexity of the resulting MPHF is $O(n \log \frac{1}{\varepsilon})$. Their algorithm is constructed in $O(n(2^\beta + (\frac{1}{\varepsilon})^\beta))$, where β the loading factor of the intermediate hash table. This is the best storage complexity in the literature for not order-preserving MPHFs.

Further works, such as the construction of Genuzio, Ottaviano and Vigna [GOV16], or the algorithm proposed by Limasset, Rizk, Chikhi and Peterlongo [LRCP17], provide reduced generation times for the MPHF compared to [BBD09], but lose the ability to be arbitrarily close to the theoretical lower bound in their storage requirement. These algorithms may be considered when the computation time of the MPHF is worth reducing at the expense of a slightly worse time complexity than with the CHD algorithm.

5.3 MPHF-based perfect rainbow trade-off

We now introduce our new technique of constructing rainbow tables using MPHFs, hereafter called *MPHF rainbow tables*. We present the adjustments needed for both the offline and the online phase of the classical rainbow trade-off in order to make use of the MPHF to associate potential endpoints to starting points. This is the first time that an alternative way to store the table allows us to truly discard the endpoint in the table, which allows us to reduce the table storage significantly compared to the classical approach which store the full endpoint, hence resulting in a better performance of the trade-off for the same parameters.

5.3.1 Offline phase

Let us start by describing the pre-computation phase of our method. It is the same, for the most part, as the pre-computation of the classical rainbow trade-off, except for an additional step at the end of the pre-computation of the matrix, and a subsequent transformation of the tables. We describe the procedure for a single table. For additional tables, the only difference will be in the choice of the reduction functions.

Let $\mathcal{N}$ be our problem space, of size N. As with the classical perfect rainbow scheme, a family of t reduction functions is chosen. Afterwards, m_0 starting points are picked in $\mathcal{N}$, either at random, or using a counter, to build m_0 chains of length t. The chains are then sorted and those with duplicate endpoints are removed to obtain m chains. We thus get a rainbow matrix RM, of size $m \times t$. We extract the first and the last columns from RM to obtain a sequence $(SP_i)_{i\in\{1,\dots,m\}}$ of starting points, and a sequence $(EP_i)_{i\in\{1,\dots,m\}}$ of the corresponding endpoints, that are ordered. At this step, we have a classical perfect rainbow table.

We compute an MPHF h on the universe $\mathcal{N}$, in such a way that the input set $I = \{EP_i\}_{i\in\{1,\dots,m\}}$ maps to the range $\{0, \dots, m-1\}$. Therefore, we have that for every $k \in \{1, \dots, m\}$, the endpoint

EP_k is associated with a distinct value in $\{0, \ldots, m-1\}$. Let us denote with σ the permutation on $\{1, \ldots, m\}$ such that

$$\forall i < m, \quad \sigma(i) = h(EP_i) + 1.$$

The idea is to apply σ on the set of indexes of the set of starting points. This way, each starting point SP_i is placed at position $h(EP_i)$, as shown in Figure 5.3.

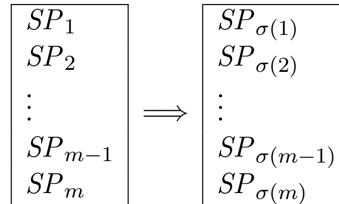


Figure 5.3 – Reordering of the table by the MPHF

The MPHF description and the newly ordered sequence of starting points are stored and constitute the MPHF rainbow table. Remark that with the MPHF, keeping the endpoints is no longer needed to obtain the association with the starting point and are thus discarded.

Note that the additional step we just described above can be performed on an existing perfect rainbow table, *i.e.*, a set of existing perfect rainbow tables can be transformed into MPHF rainbow tables. The conversion procedure is far less costly than the full pre-computation, so the fact that our method is a plug-in for the rainbow scheme is an advantage in practice.

5.3.2 Online phase

Since endpoints are not stored in the MPHF-based table, the online phase of the rainbow trade-off has to be modified accordingly.

For a given element y from which we want to find a preimage, we compute a series of potential chains of length varying from 1 to $t-1$, starting by a reduction of y to an element in $\mathcal{N}$, as it is done in the rainbow trade-off. Recall that ℓ denotes the number of tables computed for the trade-off. The order of visiting of the rainbow table is the same as in the classical rainbow scheme: we start from the leftmost part of the table and at each column, we test the ℓ tables. At each step, we get a potential endpoint PEP, to which we apply the MPHF h_i associated with the table i, for $i \in \{1, \ldots, \ell\}$. For a given table, the obtained value $s = h_i(PEP)$ gives us a value in $\{0, \ldots, m-1\}$, which we interpret as a position in the table[3]. We then fetch the starting point at position s in the table, and compute a verification chain. This first part of the online, which differ from the classical online phase, is illustrated in Figure 5.4. Once the verification chain is computed, either we find, at the end, a value x such that $h_i(x) = y$, or we do not. In the second case, one of two events arises. Either the starting point we got was from a chain which merges with the considered potential chain at some column, *i.e.*, we find a classical false alarm, or we found an unrelated starting point which had nothing to do with our chain.

This second type of event constitutes an MPHF-induced alarm that is not found in the classical rainbow trade-off. Recall that the MPHF is defined on $\mathcal{N}$, so it will give a value in the range $\{0, \ldots, m-1\}$ for all the elements of $\mathcal{N}$. We are guaranteed that if a given element is in $I \subset \mathcal{N}$, we will get the valid

[3]Note that the actual position of the starting point in the table is given by $h(PEP) + 1$ if we start counting from 1.

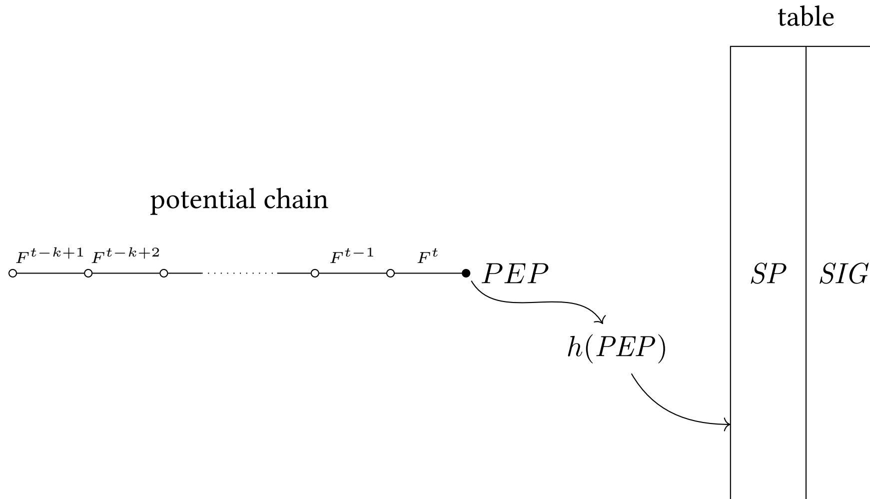


Figure 5.4 – Online phase of the MPHF technique with a potential chain of length k

position of a starting point matching this element. However, if the input of h_i is taken from $\mathcal{N} \setminus I$, we still get a value in $\{0, \dots, m-1\}$ which has no meaning. Note that, in particular, this implies that we get an alarm at every step. In the rightmost part of the table, the false alarms are frequent enough in the classical rainbow trade-off, that the behaviour is similar to the MPHF scheme. However, this is not the case further left in the table where the cost of verification at each step penalises greatly the MPHF scheme.

Therefore, we need to add a mechanism to reduce the MPHF alarms in our scheme. In order to help with the disambiguation of whether the verification chain is likely to be from a valid starting point, we introduce a *signature* of the correct endpoint associated with the starting point, which is described in the following section.

5.3.3 Signature

In our scheme, a signature is a way to characterise whether a given potential chain is related to a given starting point. In the table, the signature is stored alongside each starting point, in such a way that when a given starting point is fetched, the signature corresponding to this starting point comes along with it for free.

A good characterisation of the correct chain, which can be applied at every step, is partial information from the endpoint. We define our signature function by

$$\forall x \in \mathcal{N}, \quad \mathrm{sig}_b(x) = \left\lfloor \frac{x}{2^b} \right\rfloor .$$

This is a somewhat arbitrary choice, as any function extracting b bits from the endpoint would have the same effect.

Now that we introduced a signature to mitigate the effect of the MPHF-related alarms, the presentation of our scheme is complete. The full algorithm of the online phase of the MPHF rainbow trade-off with signatures, which, from now on, we will refer to simply as the MPHF rainbow trade-off, is given in Algorithm 5.5, where we use the r_k^i notation to designate the reduction function of the table i that

Algorithm 5.5 Online inversion in the MPHF rainbow trade-off

```
for k = 0 to t do
  for i = 1 to ℓ do
    x ← r_k^i(y_0)
    for j = k + 1 to t do
      x ← r_j^i ∘ f(x)
    end for
    s ← h(x)
    if sig(x) = SIG_s^i then
      x ← SP_s^i
      for j = 1 to t − k − 1 do
        x = r_j^i ∘ f(x)
      end for
      if f(x_0) = y_0 then return x_0
      end if
    end if
  end for
end for
```

applies to the element at the kth column. Similarly, a row in the table i, containing a starting point and a signature, is denoted with the pair (SP_k^i, SIG_k^i), for $k \in \{1, \ldots, m\}$.

5.4 Analysis

We now compute the expected time of the online phase, as well as the storage requirement of the method. We review the advantages of the method by discussing its modest pre-computation overhead, and comparing it with the truncation technique.

5.4.1 Pre-computation overhead

The objective is now to quantify the pre-computation overhead incurred by our method. By overhead, we mean the time-difference between the MPHF rainbow scheme offline phase and the classical rainbow scheme offline phase. The goal is to assess the practicality of the method.

During the pre-computation, the additional step to compute MPHF rainbow tables from classical rainbow tables involves a computation of the MPHF, and the reordering of the table. As overviewed in Section 5.2, with all the modern MPHF algorithms, the construction of an MPHF which associates m arbitrary elements to the range $\{0, \ldots, m-1\}$ is linear in m. Since the pre-computation itself requires $K\ell m_1 t$ computations of the one-way function, for some $K > 1$, we know that, at least asymptotically, the MPHF construction is negligible.

It remains to ensure that the $O(m)$ constant is not too high. If we consider modern algorithms [BZ07; BPZ07; BBD09], we can see that the generation time for 10^6 keys is in the order of dozens or hundreds of seconds. If we consider the same amount of time used to compute a one-way function, by reusing, *e.g.*, the timings given in Section 4.5 for the MD5 function, we have that about $5.5 \cdot 10^9$ computations of the one-way function can be done in about 1000 seconds. This corresponds to approximately $2^{32.37}$ computations. Therefore, a condition such that $m_1 \gg m$, which is not rare in practice, is enough to

neglect the MPHF construction time. The reordering of the table only requires $O(m)$ swap operations, which will not exceed the generation time of the MPHF.

Overall, the additional step required by the MPHF scheme is moderately time-consuming. In fact, even in the case where it is applied to an existing table, the process is very practical in terms of computation time.

5.4.2 Online running time

The expected amount of computation of the one-way function due to the computation of the potential chains is not altered in any way in the MPHF scheme, so we present the amount of computations due to false alarms in a self-contained way, in Theorem 5.3.

Theorem 5.3. *The expected amount of computations due to false alarms in the online phase of the MPHF rainbow trade-off with ℓ tables, and using a signature of b bits, is given by*

$$Q_{MPHF} = \sum_{k=1}^{t} \left(1 - \frac{m}{N}\right)^{(k-1)\ell} \left[\frac{1}{2^b} + \left(1 - \frac{1}{2^b}\right) \frac{m}{N} \left(k - \frac{mk^2}{4N}\right)\right] \ell(t-k). \tag{5.5}$$

Proof. The first condition for a false alarm to happen at a step $k < t$ is that none of the previous steps yielded the answer, which happen with probability

$$\left(1 - \frac{m}{N}\right)^{(k-1)\ell}$$

for ℓ tables. We consider that a step includes the trial of the ℓ tables, which implies that we count ℓ searches at each step. Therefore, we might still count up to $\ell - 1$ additional steps in the end while an implementation would stop right after the answer is found. Nevertheless, since ℓ is always small in the perfect rainbow trade-off, the resulting loss due to this difference is negligible compared to the $t \gg \ell$ total searches.

The bracket factor of Equation (5.5) gives the probability of a false alarm. In our scheme, an alarm happens at step $k \in \{1, \ldots, t\}$ with probability 1. The alarm induces a verification when two events happen: our endpoint does not belong to the endpoints of the rainbow matrix, and the signature failed to avoid the verification. This corresponds to picking a b-bit long value at random that matches our signature. This happens with probability $\frac{1}{2^b}$. When the signature does not match, which happens with probability $1 - \frac{1}{2^b}$, we can still have a false alarm due to a merge of the potential chain with a chain of the matrix. The probability of such a merge happening was already studied in Section 2.4 and is, when using the closed formula of Theorem 3.6, given by

$$\frac{mk}{N}\left(1 - \frac{mk}{4N}\right).$$

Finally, when a false alarm happens, at column k, the verification chain of each of the ℓ tables induces $\ell(t-k)$ computations of the one-way function. □

The total amount T_{MPHF} of computations of the one-way function can then be obtained by plugging the Q_{MPHF} value from the MPHF technique into Theorem 2.9. The result – an extension of Theorem 5.3 – is given in Theorem 5.4 for completeness.

Theorem 5.4. *The expected amount of computations of the one-way function in the online phase of the MPHF rainbow trade-off with ℓ tables, and using a signature of b bits, is given by*

$$T_{MPHF} = \sum_{k=1}^{t} \left(1 - \frac{m}{N}\right)^{(k-1)\ell} \left\{ (k-1) + \left[\frac{1}{2^b} + \left(1 - \frac{1}{2^b}\right) \frac{m}{N} \left(k - \frac{mk^2}{4N}\right) \right] \ell(t-k) \right\}.$$

We can see that the online time is now tunable with regard to the signature length parameter b. However, as with most improvements on TMTOs, we observe that the asymptotic online complexity is still $O(t^2)$, and does not change for any value of $b \in \mathbb{N}$. Notice that, when $b \to \infty$ in the formula from Theorem 5.4, we obtain T_{rainbow} from Section 2.4. In fact, with $b \approx \log_2 N$ and beyond, each signature uniquely determines a given endpoint and we are in the situation where the MPHF is redundant with regard to the classical scheme. We therefore introduced a new trade-off with the parameter b, where the online time can be reduced by increasing b, at the expense of a higher memory requirement, since a larger signature has to be stored. The storage aspect of our method is considered next.

5.4.3 Storage requirements

With the MPHF rainbow scheme, we aim at providing a lower storage for the table, compared to a classical table with the same online time. The storage in our scheme is given by Theorem 5.5.

Theorem 5.5. *A single full table in the MPHF rainbow trade-off using a signature of b bits, is stored on an amount of bits given by*

$$M_{MPHF} = m(\log_2 m_1 + (1+\varepsilon)\log_2 e + b),$$

with $\varepsilon > 0$.

Proof. Each row of the MPHF technique consists of a starting point and a signature. The starting point is in the range $\{1, \ldots, m_1\}$, so it is stored on $\log_2 m_1$ bits. The signature is stored on b bits. We factor in the description of the MPHF itself. For the CHD algorithm, presented in Section 5.2.2, which provides the best space complexity, the amount of bits required to describe an MPHF is $(1+\varepsilon)\log_2 e$, for some $\varepsilon > 0$. Setting ε to arbitrarily high values renders the claim valid for other algorithms with linear space complexity. □

The storage of starting points takes the same amount of bits as in the case of the classical rainbow table. Therefore, what remains to be shown is the comparison with the storage of the endpoint in the classical technique. A few observations can already be made. Apart from the starting point, the amount of bits needed to store a single row in the table is independent from the parameters of the rainbow trade-off. This was not the case in the classical trade-off, where the endpoint was directly linked with the problem size N. Also, the storage of the MPHF is parameterised with ε and b. Arguably, the second parameter will indirectly depend on the rainbow parameter since it influences the online time, so if we aim at reducing this online time, we will have to take these parameters into account to choose b.

5.4.4 Comparison with the truncated point technique

Due to the fact that the endpoint is not stored, the MPHF scheme lacks some information compared to the classical trade-off. When it is completed with a signature, it is reminiscent of the truncated point technique which was introduced in Section 3.2. Indeed, the two techniques rely on an incomplete

information about the endpoint to determine whether the verification should occur in the event of a false alarm. Therefore, a comparison with the truncated point method, rather than with the original trade-off, is more pertinent.

We provide a comparison between the truncated method and our scheme in Figure 5.5. The amount of computations due to false alarms is given in terms of amount of truncation for the truncated method, and in terms of size of signature for the MPHF method. The parameters are $N = 2^{40}$, $m = 3.5 \cdot 10^8$, $t = 5992$ and $\ell = 4$ tables. Since the computations due to the potential chains are identical in the two cases, we focus on the computations related to false alarms. For the MPHF technique, we use Q_{MPHF} from the formula given in Theorem 5.3. The false alarms for the classical rainbow trade-off, when we remove the b least significant bits of the endpoints, is given by the exact same formula as in Theorem 5.3 when replacing the probability of an MPHF related false alarm by the probability that a potential endpoint matches a partial endpoint, which is given by

$$\Pr(\exists i,\ \ \text{sig}(x) = SIG_i | \nexists j,\ \ x = EP_j) = 1 - \left(1 - \frac{m}{N}\right)^{2^b - 1}.$$

With this simple formula, we disregard the fact that some of the truncated endpoints can be equal to each other. This means that the online time with truncated points that is given here is close to the reality for small b values, and corresponds to an optimistic estimate of the real value when b increases. Note that for the two techniques the x-axis is reversed, because increasing b for the MPHF corresponds to adding information to the signature, whereas increasing b for the truncation technique corresponds to removing bits from the stored endpoints.

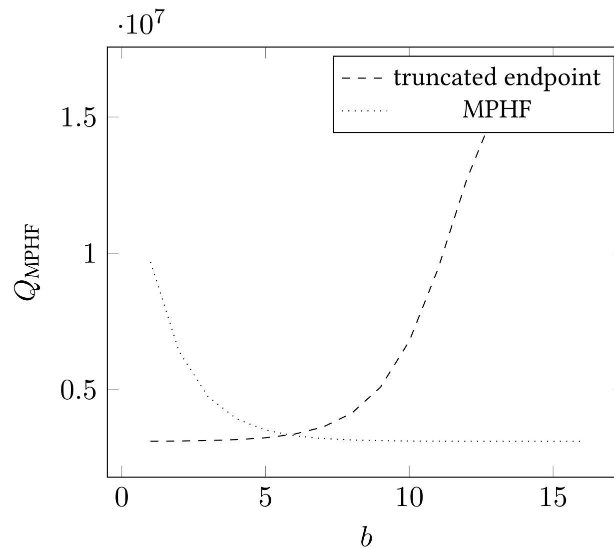


Figure 5.5 – False alarm-related computations of the truncated endpoint and the MPHF techniques

We can observe in Figure 5.5 that the two curves intersect, which means that for a certain b, the amount of false alarms, and therefore the total expected online time, is the same for the two techniques. We can see that the value is between 5 and 6, which corresponds to a signature of 5 bits for the MPHF technique, and a partial endpoint of $\log N - b = 35$ bits for the truncated point technique. Even when accounting for the description of the MPHF adding a few bits to the MPHF storage technique, we can already see that, for the same online running time, the MPHF technique requires much less storage.

What we can infer from the results of Figure 5.5 is that, in the classical trade-off, the information from the endpoint is more needed for the association to its corresponding starting point than for the characterisation of the chain it provides in the case of a truncation-related false alarm.

5.5 Optimal parameters

In the analysis presented in the last section, we computed the expected online time complexity and the amount of bits required to store a table. We can now exploit these to obtain the optimal parameters for our scheme. After an overview on the best MPHF to use for rainbow tables, we discuss the parameters that are optimal in the sense that they exploit the given memory optimally to achieve the best online time possible.

5.5.1 MPHF parameters

We now discuss the choice of the MPHF algorithm, to be used in our method. With our scheme, we aim at reducing the storage space required for the table. This is the metric that is optimised in the choice of the MPHF, as well as in the choice of the MPHF parameters. We also want a relatively "short" time for the generation of the MPHF. This last condition is loose in the sense that short only means that the generation time must not be significant enough with regard to the total pre-computation time. A generation of the MPHF that is linear in m satisfies this condition as long as the pre-computation constant is practical. From the algorithms presented in Section 5.2, we can settle on the CHD algorithm proposed in [BBD09], since it is the algorithm that provides the closest storage complexity from the theoretical lower bound on MPHF description size.

The CHD algorithm allows us to store the MPHF in $(1+\varepsilon)\log_2 e$ bits per key. The extent to which the value ε can be reduced is not discussed in [BBD09], except for the fact that setting $\varepsilon = 0$ is not practical since it would require in the order of n hash functions for the universal hashing, and a construction time which would be quasi-linear instead of linear. However, it is clear from [BBD09] that decreasing ε has two consequences. The first one is that the generation time is increased due to the fact that more hash functions have to be computed in the algorithm. The second is that the load factor β of the intermediate hash table used in the CHD algorithm has to be increased. Otherwise, if β is too low, it might not be possible to produce a suitable intermediate hash table with a maximal number of 1 element per bucket. Recall from Section 5.2 that the time complexity of the CHD algorithm is given by

$$O\left(n\left(2^\beta + \left(\frac{1}{\varepsilon}\right)^\beta\right)\right),$$

which means that increasing β increases the generation time of the MPHF.

In [BBD09], the performed experiment limits the values of β (which is referred to with λ in [BBD09]) to the range $\{1,\ldots,5\}$. For the MPHF, Belazzougui et al. have been able to reduce ε down to about 0.434 which gives a storage space of 2.07 bits per key. Such an MPHF's generation time, which is evaluated in [BBD09], is gathered in Table 5.1.

We can see that the generation time is in the order of the minute for a number of keys in the order of the millionth. Considering the numbers of keys in two or even three orders of magnitude higher, we can expect the generation time to be practical, considering that it seems to evolve in practice slightly faster than at a linear pace.

5

number of keys	generation time (s)
$5 \cdot 10^6$	41.4
$10 \cdot 10^6$	104.4
$15 \cdot 10^6$	185.6
$20 \cdot 10^6$	272.6

Table 5.1 – Generation time of the CHD algorithm for $\varepsilon \approx 0.434$

While it might be possible to further improve the storage of the MPHF, it might induce a prohibitive generation time, which would not be negligible with regard to the pre-computation. Nevertheless, the gain of such an improvement would be of about half a bit at most to get from the current practical 2.07 bits per key to the 1.44 bits per key theoretical lower bound. Therefore, in the following, we will reuse the parameters established in [BBD09] for an MPHF stored on 2.07 bits per key.

5.5.2 Optimal signature

We now consider how to choose the size of the signature of our scheme. If b denotes the signature length, then the MPHF-based TMTO needs $m(2.07 + b)$ bits to store the MPHF and all the signatures.

We take a different approach than in the analysis of Section 5.4. Here we fix a given amount of memory in which we want to store the table, and evaluate the performance of the MPHF scheme given a signature b, using the formula computed from Theorem 5.3.

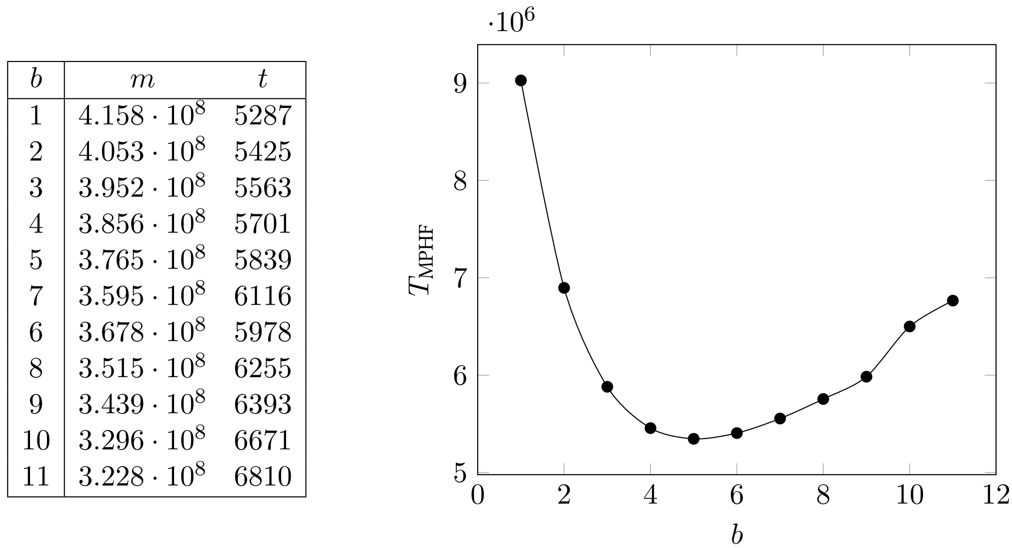
In what follows, we consider a practical problem of size of $N = 2^{40}$, with $\ell = 4$ tables. Assuming that the counter technique is used during the pre-computation, we fix $m_1 = Km^*$ for some $K > 1$, where m^* is the desired m in the perfect table. When K is large enough, the $m \approx m^*$ approximation holds – and we will assume that this is the case. We fix the success rate by the mean of the matrix stopping constant $c = \frac{mt}{N}$. This way, any choice of m implies that the t parameter is determined as well, and reciprocally, choosing t implies choosing m. As stated above, we also assume that the MPHF is stored on $2.07m$ bits.

We also fix, somewhat arbitrarily, a limit of 8GB per table, which, with $\ell = 4$, sets our memory at $M_{\text{max}} = 32$GB. This amount of memory is reasonable on nowadays' commodity computers[4].

The parameters m and t will be determined as follows. This memory M is the maximum amount of memory to store our tables, so the idea is to store a maximum of rows as possible in the memory, *i.e.*, to find m as large as possible so that $M_{\text{MPHF}} \leq M_{\text{max}}$, where M_{MPHF} is the value given in Theorem 5.5. Note that M_{MPHF} depends indirectly on m due to the storage required by the starting point, so this is a functional equation in m. This can be converted in a minimization problem that we chose to solve numerically here. For each signature size b, we find a value of m which satisfies the memory requirement. From there, the parameter t can be deduced from the matrix stopping rule, whose constant has been fixed.

Using the parameters m and t obtained for each b, we compute the online running time of the MPHF rainbow scheme. The results are presented in Figure 5.6. The configurations that were determined by setting values of b in the range $\{1, \ldots, 11\}$, are given in Figure 5.6a. The expected online time for each of these configurations is given in Figure 5.6b.

[4] We choose to use integer values of GB of RAM for simplicity, but in practice one would have to accommodate for the fact that the operating system requires some memory for itself to function properly, and would have to limit the memory slightly below ours, when using a computer with 32GB of memory.



b	m	t
1	$4.158 \cdot 10^8$	5287
2	$4.053 \cdot 10^8$	5425
3	$3.952 \cdot 10^8$	5563
4	$3.856 \cdot 10^8$	5701
5	$3.765 \cdot 10^8$	5839
7	$3.595 \cdot 10^8$	6116
6	$3.678 \cdot 10^8$	5978
8	$3.515 \cdot 10^8$	6255
9	$3.439 \cdot 10^8$	6393
10	$3.296 \cdot 10^8$	6671
11	$3.228 \cdot 10^8$	6810

(a) Fixed memory configurations

(b) Evolution of the online time with the size of signature

Figure 5.6 – Evolution of the signature length with a fixed memory $M = 32$GB

We can observe that making the signature longer rapidly decreases the amount of false alarms, and thus the online time. However, when the signatures are too long (here $b > 5$), what is saved by not storing the endpoints does no longer balance the memory cost of the signatures and the MPHF. In our example, the values $m = 3.765 \cdot 10^8$ and $b = 5$ are optimal, and yield $T = 5.34 \cdot 10^6$.

We now reiterate the whole procedure of finding an optimal b, for a varying problem spaces and memory size. The results are shown in Figure 5.7. Each point on the plot represents a combination of a memory limit and a problem space size. The data points are shaped differently depending on the value of b they represent. We use wide ranges of both parameters to encompass most of the achievable TMTOs in our observation. The problem spaces' sizes are chosen in the range $[2^{30}, 2^{60}[$. For each problem space, we evaluate the optimal b on the memory limits that make sense, *i.e.*, we keep the assumption that $t \gg 1$, so that we have $m \ll N$. We fix the upper bound of the memory on the petabyte (PB) order of magnitude, *i.e.*, $M = 2^{50}$.

We first observe that there is not a large variance in the optimal signature sizes. In fact, the value of $b = 5$ that we obtained above, in our first evaluation, is among the most common values for most TMTOs. It drops below 5 when the memory allowed for the TMTO is very low for each problem space. This indicates that when the memory is at a premium, adding new chains is more valuable in terms of information for the online algorithm than taking longer signatures. The maximum value of b in the range of problem spaces we observed is 6, which means that the memory required by our scheme is unlikely to gain more than a few bits in any conceivable TMTO configuration.

5.6 Experimental validation

We made an implementation of the MPHF technique, which we use to perform an experimental validation. We show that the obtained values are in agreement with the theoretical formulas given in Section 5.4.

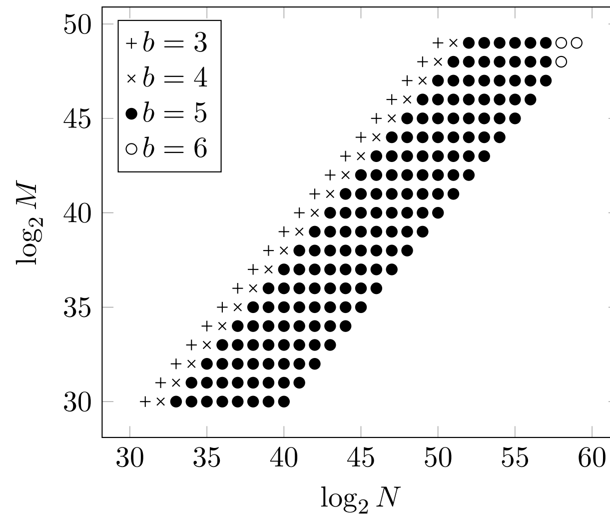


Figure 5.7 – Optimal signature for varying problem size and memory size

5.6.1 Pre-computation

We implemented the MPHF method by creating tools to convert classical rainbow tables into MPHF-based rainbow tables. The CMPH [CBBZ12] library, implemented by the authors of [BBD09], which provides the CHD algorithm, was used for the MPHF part.

To evaluate the method, we computed two sets of tables for a moderate problem size $N = 2^{36}$, and for a larger problem size $N = 2^{40}$. Since we are only interested in the online time in this section, we settle for a suboptimal naive storage: we store the starting points in $\log N$ bits. While this will not affect the online time results given in this section, it eases the implementation. For a problem of size $N = 2^{40}$, we chose an arbitrarily size of 2GB per table (without accounting for the description of the MPHF), which gave us a goal of $m^* = 366993200$ to construct maximum tables. For $N = 2^{36}$ the tables are stored in about 780MB and the targeted size for maximum tables is $m^* = 152709948$.

A number of $m_1 = 25m^*$ starting points was used to build the perfect tables. The factor of 25 allowed us to obtain tables that are close to being maximal, while keeping a reasonable pre-computation effort. The parameters of the tables obtained after removing chains with duplicate endpoints, as well as the optimal signature length for these parameters, are given Table 5.2. The sizes obtained for the tables are indeed close to maximal: we have that each table size is approximately $0.96m^*$.

The conversion of the perfect tables into MPHF-compatible tables took in average 553s for each of the 4 tables. The reordering of the starting points according to the index given by the MPHF took in average 171s. As expected, theses quantities, which are linear in m, require a negligible time compared to the full pre-computation, which took a little over a CPU-year.

5.6.2 Online time performance

To confirm our claim that the computation of the MPHF during the online phase can be neglected, we averaged the computation of 10^5 values over 10^5 runs. With our setup, we obtain 0.158μs per application of the MPHF. For reference, an application of our one-way function takes about 0.178μs on the same setup. The MPHF is performed once per step, whereas k computations of the one-way

Table 5.2 – Parameters of the computed tables

	$N = 2^{36}$	$N = 2^{40}$
chain length t	900	5992
ℓ	4	4
size of each of the ℓ tables	146042925, 146043361, 146055023, 146036907	352805857, 352806958, 352761350, 352826288
matrix stopping constant c	1.91	1.92
signature size b	5	5

function have to be performed at each step. Therefore, when we have $\ell t \gg 1$, we can indeed disregard the time taken by the MPHF itself.

The average online time is given in Table 5.3. We performed 10^6 inversions using our TMTO for both problem spaces. The number and cost of false alarms include both the false alarms related to potential chain merging, and the ones caused by incorrect signature matches, since it is difficult to distinguish of which type a given false alarm is in practice. The Q_{MPHF} and T_{MPHF} values are obtained via Theorem 5.3 and Theorem 5.4 respectively. The m value used in the formula was the average of the size of the tables obtained by the pre-computation. The values obtained experimentally, denoted with “exp” in Table 5.3, are the average values from all the inversions that were performed.

Table 5.3 – Average online time results for the MPHF technique

	$N = 2^{36}$	$N = 2^{40}$
success rate	99.97%	99.96%
T_{MPHF}	$1.34 \cdot 10^5$	$5.89 \cdot 10^6$
T_{MPHF} (exp.)	$1.34 \cdot 10^5$	$5.89 \cdot 10^6$
false alarms	113.5	750
Q_{MPHF}	$7.94 \cdot 10^4$	$3.51 \cdot 10^6$
Q_{MPHF} (exp.)	$8.01 \cdot 10^4$	$3.50 \cdot 10^6$

The results gathered in Table 5.3 show us that the success rate confirms that our choice for m_1 was reasonable, as we still have a success rate very close to 100%, even though our tables are not strictly maximal. When comparing the experimental values to their theoretical counterpart, we notice that the expected value of the formulas from Section 5.4 are within 1% of the values that we got from the experiment. Note that we used a high number of samples in our experiment. This is required due to the high variance of the result when the number of samples is too low. Indeed, when too few samples are used, there is a high probability that we do not encounter values that are only in the rightmost part of the rainbow matrix, and whose cost is much higher than most of the values. Similarly, we notice that, for a larger t of 5992, the error is smaller (0.2%).

Overall, the implementation of our method is rather straightforward, in the sense that the pre-computation part is independent from the classical pre-computation of the rainbow tables. By using a library such as CMPH, which implements the state of the art algorithms for MPHFs, the modification of the online phase is actually somewhat simpler than with the classical rainbow trade-off, which requires

an additional search method for the endpoints, such as a binary search procedure.

5.7 Conclusion

With the MPHF technique, we no longer need to keep the endpoint associated to a given starting point. Nevertheless, we get an associated starting point at every column of the online phase algorithm, whether our endpoint was in the rainbow matrix or not, so we have in fact a false alarm at every column. Therefore, to reduce these MPHF-induced false alarms, we introduced signatures as a mean to help us determine whether endpoints potentially belong to the table or not.

We chose to use the most significant bit of the endpoint as a signature, but it must be noted that a signature does not share the same nature as truncated endpoints of traditional techniques. We observed that we require less bits to characterize the endpoint in a signature than we need to store with a classical truncated endpoint. This gain is sufficient to compensate the fact that we have to store the MPHF itself.

Indeed, assuming we use the parameters in [BBD09] with 2.07 bits per elements, we get a better memory footprint than with the truncated endpoint technique, which is the closest improvement in terms of removing endpoint information, since we used the least significant bits of the endpoint as signature. Whether supplementary gain may be obtained by using a different set of bits for the signature remains an open question.

It might be possible to lower the storage required by the MPHF; however, it would require to spend more time on the pre-computation. The extent to which the parameters are tunable would require another kind of analysis, involving evaluation of the expected behaviour of randomly chosen hypergraphs.

It must be noted that this MPHF technique can be applied to already computed perfect rainbow tables as long as the complete endpoints are available. Overall, the novelty of the technique gives a new perspective on how to consider the endpoints, and might lead to further improvements of the online phase of the perfect trade-off.

While we provided some figures on the performance of our method, in particular compared to the truncated endpoint method, it is still unclear at this point how it performs when optimisations such as compressed delta encoding, or checkpoints, are involved. This is the subject of Chapter 6, which evaluates the trade-offs with optimisations, and in which we included our new MPHF-based technique.

References

[ACFT19] Gildas Avoine, Xavier Carpent, Barbara Fila, and Florent Tardif. *Cryptanalytic Time-Memory Trade-Offs: Recommended Configurations.* unpublished. 2019 (cit. on pp. xiii, 9, 126).

[BBD09] Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. *Hash, Displace, and Compress.* In: *Algorithms - ESA 2009.* Ed. by Amos Fiat and Peter Sanders. Vol. 5757. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 682–693. isbn: 978-3-642-04128-0 (cit. on pp. 134, 137, 141, 142, 144, 146).

[BPZ07] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. *Simple and Space-Efficient Minimal Perfect Hash Functions.* In: *Algorithms and Data Structures, 10th International Workshop, WADS 2007.* Ed. by Frank Dehne, Jörg-Rüdiger Sack, and Norbert Zeh. Vol. 4619. Lecture Notes in Computer Science. Halifax, Canada: Springer Berlin Heidelberg, 2007, pp. 139–150. isbn: 978-3-540-73951-7 (cit. on pp. 134, 137).

[BZ07] Fabiano C. Botelho and Nivio Ziviani. *External Perfect Hashing for Very Large Key Sets.* In: *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management.* CIKM '07. Lisbon, Portugal: ACM, 2007, pp. 653–662. ISBN: 978-1-59593-803-9 (cit. on p. 137).

[CBBZ12] Davi de Castro Reis, Djamel Belazzougui, Fabiano Cupertino Botelho, and Nivio Ziviani. *CMPH library.* Accessed: May 2019. 2012. Link. (Cit. on p. 144).

[Cha84] C. C. Chang. "The Study of an Ordered Minimal Perfect Hashing Scheme". In: *Commun. ACM* 27.4 (Apr. 1984), pp. 384–387. ISSN: 0001-0782 (cit. on p. 131).

[Cic80] Richard J. Cichelli. "Minimal Perfect Hash Functions Made Simple". In: *Commun. ACM* 23.1 (Jan. 1980). Ed. by M. Douglas McIlroy, pp. 17–19. ISSN: 0001-0782 (cit. on pp. 130, 132).

[CW77] J. Lawrence Carter and Mark N. Wegman. *Universal Classes of Hash Functions (Extended Abstract).* In: *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing.* STOC '77. Boulder, Colorado, USA: ACM, 1977, pp. 106–112 (cit. on pp. 130, 132).

[CW79] J. Lawrence Carter and Mark N. Wegman. "Universal Classes of Hash Functions". In: *J. Comput. Syst. Sci.* 18.2 (1979), pp. 143–154 (cit. on p. 130).

[DH01] Martin Dietzfelbinger and Torben Hagerup. *Simple Minimal Perfect Hashing in Less Space.* In: *Algorithms - ESA 2001.* Ed. by Friedhelm Meyer auf der Heide. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 109–120. ISBN: 978-3-540-44676-7 (cit. on p. 134).

[Dum56] Arnold I. Dumey. "Indexing for rapid random access memory systems". In: *Computers and Automation* 5.12 (Dec. 1956), pp. 6–9 (cit. on p. 128).

[FHCD92] Edward A. Fox, Lenwood S. Heath, Qi Fan Chen, and Amjad M. Daoud. "Practical Minimal Perfect Hash Functions for Large Databases". In: *Commun. ACM* 35.1 (Jan. 1992), pp. 105–121. ISSN: 0001-0782 (cit. on pp. 130, 132).

[FKS82] Michael L. Fredman, János Komlós, and Endre Szemerédi. *Storing a Sparse Table with $O(1)$ Worst Case Access Time.* In: *23rd FOCS.* Chicago, Illinois: IEEE Computer Society Press, Nov. 1982, pp. 165–169 (cit. on p. 133).

[GOV16] Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. *Fast Scalable Construction of (Minimal Perfect Hash) Functions.* In: *Experimental Algorithms - 15th International Symposium, SEA 2016.* Vol. 9685. Lecture Notes in Computer Science. L'Aquila, Italy: Springer International Publishing, June 2016, pp. 339–352. ISBN: 978-3-95977-070-5 (cit. on p. 134).

[GT63] Marek Greniewski and Wladyslaw Turski. "The External Language KLIPA for the URAL-2 Digital Computer". In: *Commun. ACM* 6.6 (June 1963), pp. 321–324. ISSN: 0001-0782 (cit. on p. 129).

[Hel67] Herbert Hellerman. *Digital computer system principles.* In: 1st ed. New York: McGraw-Hill, 1967. Chap. 2, p. 152 (cit. on p. 128).

[HK86] Gary Haggard and Kevin Karplus. *Finding Minimal Perfect Hash Functions.* In: *Proceedings of the Seventeenth SIGCSE Technical Symposium on Computer Science Education.* Ed. by Joyce Currie Little and Lillian N. Cassel. SIGCSE '86. Cincinnati, Ohio, USA: ACM, 1986, pp. 191–193. ISBN: 0-89791-178-4 (cit. on p. 132).

[Jae81] G. Jaeschke. “Reciprocal Hashing: A Method for Generating Minimal Perfect Hashing Functions”. In: *Commun. ACM* 24.12 (Dec. 1981), pp. 829–833. ISSN: 0001-0782 (cit. on p. 131).

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. In: vol. 3. The Art of Computer Programming. Addison-Wesley, 1973. Chap. 6, pp. 506–541 (cit. on pp. 128, 129).

[LRCP17] Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. *Fast and Scalable Minimal Perfect Hashing for Massive Key Sets*. In: *16th International Symposium on Experimental Algorithms, SEA 2017, June 21-23, 2017, London, UK*. Ed. by Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman. Vol. 75. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 25:1–25:16. ISBN: 978-3-95977-036-1 (cit. on p. 134).

[Meh82] Kurt Mehlhorn. *On the Program Size of Perfect and Universal Hash Functions*. In: *23rd FOCS*. Chicago, Illinois: IEEE Computer Society Press, Nov. 1982, pp. 170–175 (cit. on pp. 132, 133).

[Pag99] Rasmus Pagh. *Hash and Displace: Efficient Evaluation of Minimal Perfect Hash Functions*. In: *Algorithms and Data Structures, 6th International Workshop, WADS ’99*. Ed. by Frank K. H. A. Dehne, Arvind Gupta, Jörg-Rüdiger Sack, and Roberto Tamassia. Vol. 1663. Lecture Notes in Computer Science. Vancouver, British Columbia, Canada: Springer Berlin Heidelberg, 1999, pp. 49–54. ISBN: 978-3-540-48447-9 (cit. on pp. 133, 134).

[Pri57] Robert C. Prim. “Shortest connection networks and some generalizations”. In: *The Bell System Technical Journal* 36.6 (Nov. 1957), pp. 1389–1401. ISSN: 0005-8580 (cit. on p. 132).

[Sag85] Thomas J. Sager. “A Polynomial Time Generator for Minimal Perfect Hash Functions”. In: *Commun. ACM* 28.5 (May 1985), pp. 523–532. ISSN: 0001-0782 (cit. on pp. 131, 132).

[Spr77] Renzo Sprugnoli. “Perfect Hashing Functions: A Single Probe Retrieving Method for Static Sets”. In: *Commun. ACM* 20.11 (Nov. 1977), pp. 841–850. ISSN: 0001-0782 (cit. on pp. 129, 130).

[SS90] Jeanette P. Schmidt and Alan Siegel. “The Spatial Complexity of Oblivious k-Probe Hash Functions”. In: *SIAM J. Comput.* 19.5 (1990), pp. 775–786. ISSN: 0097-5397 (cit. on p. 133).

[TY79] Robert Endre Tarjan and Andrew Chi-Chih Yao. “Storing a Sparse Table”. In: *Commun. ACM* 22.11 (Nov. 1979), pp. 606–611. ISSN: 0001-0782 (cit. on p. 133).

If you optimize everything, you will always be unhappy.

Donald Knuth

TMTOs' optimisation 6

Comparisons of TMTOs are challenging in many ways, due to the versatility of the technique. We propose, in this chapter, to tackle such a comparison, with a new model that we introduce. The objective is to assess which combination of improvements achieves the best efficiency.

Contents

6.1 Introduction **152**
- 6.1.1 Goal 152
- 6.1.2 Difficulties of TMTO comparisons 153
- 6.1.3 Context of comparison 154

6.2 Preliminary comparison **154**
- 6.2.1 Worst case asymptotic analysis 155
- 6.2.2 TMTO characteristic for high success rates 156
- 6.2.3 Comparisons for arbitrary success rate 157
- 6.2.4 Conclusion 158

6.3 Fixed memory model **159**
- 6.3.1 Reducing the dimension 159
- 6.3.2 Combinations 161
- 6.3.3 Optimised storage parameters 164
- 6.3.4 Checkpoints optimisation 165

6.4 Comparison **170**
- 6.4.1 Parameters and terminology 170
- 6.4.2 Standalone improvements 171
- 6.4.3 Storage-oriented configurations 173
- 6.4.4 MPHF based rainbow trade-off 174
- 6.4.5 Combinations with checkpoints 176
- 6.4.6 Recommendation 177

6.5 Conclusion **178**

6.1 Introduction

In Sections 2.2, 2.3, and 2.4, we discussed the main TMTO variants: the Hellman, the DP, and the rainbow trade-off. In Chapter 3, we reviewed improvements for any or all of these variants. These improvements are sometimes incompatible due to some overlapping, but some can also be combined. This chapter aims at making sense of all of these improvements, classifying them in such a way that we can build an improved TMTO that leverages several improvements. In other words, our goal is the determination of the most efficient TMTO configuration, for a modern context of use which we will detail. We build upon existing comparisons, and extend the results of these works in order to include the improvements that were reviewed in this thesis in Chapter 3, as well as the MPHF technique that was introduced in Chapter 5.

The outline of this chapter is as follows. In Section 6.1, we expose in detail the problem of the comparison, and make some observations on the challenges it involves. We discuss the practical usage of TMTOs in order to make the assumptions that we use precise. We then review in Section 6.2 the existing work in the literature that tackled such comparisons. In this first step analysis which does not involve improvements, the existing comparisons, along with our assumptions, lead us to focus on the rainbow trade-off as a choice of TMTO variant. Then, in Section 6.3, we present our *fixed memory model*, which provides a more fine grained comparison than existing works by relying on exact storage requirements. While the model itself is straightforward, the consequences of fixing the memory require careful precautions, which are discussed. When it makes sense to do so, we combine different improvements in a multi-layered setup, where the storage-related improvements are first considered, and then the checkpoint technique is applied. We expose our methodology to determine the optimal parameters of each combination. The optimisation of the checkpoint technique happens to require prohibitive computation, and an analysis is performed to determine speedup heuristics. Finally, in Section 6.4, we present our results by comparing the different optimised combinations on an equal footstep with the help of our model.

6.1.1 Goal

Let us define what we mean by efficiency, and describe the objective of this chapter in more detail. The question we aim to answer is the following. Given a set of available resources such as physical memory and computation speed, and a given pre-image problem which is compatible with the use of TMTOs, what are the techniques and parameters that will give us the lowest average online time? In our attempt to determinate the best trade-off technique, we aim at providing our results in such a way that a TMTO implementation developer can make the right choices.

Some conclusions when comparing TMTOs might be overturned when using extremely large values for certain parameters. In our analysis, we provide wide, yet practical parameter ranges for our comparisons. While we cannot exclude some unprecedented evolution of the technology, it is very unlikely that, in the upcoming years, either computation capabilities or memory hardware improve in such a way that our conclusion would have to change.

Due to the high number of considerations that one has to make when constructing a TMTO, some choices have to be made in order to present a clear comparison. In particular, many TMTO improvements come with their own additional small trade-offs. Dealing with every possible variation would make the analysis impractical. Therefore, in order to expose the comparison more clearly, we focus on the most interesting parameters, namely the time and the memory, and fix other secondary parameters such as the probability of success and the problem space, somewhat arbitrarily. Once fixed, they do not

influence the behaviour of the main trade-offs.

In the rest of this section, we present the common difficulties and pitfalls encountered when attempting to compare TMTOs. We then present in more detail the practical context on which we base our comparisons, and the choices that derive from it.

6.1.2 Difficulties of TMTO comparisons

While it is easy to argue in favor of a technique using a single metric, *e.g.*, considering only the memory requirement when comparing, it does not make sense with TMTOs. In particular, with a given dimension taken in isolation, such as the memory, or the online time, it is always possible to come up with a "better" value for a given problem by, *e.g.*, giving up on the success rate, or taking arbitrary high, non-practical values, for others dimensions than the one considered. Another problem is that, for a given set of parameters, ranking high in a comparison might not reflect the performance of a technique as a whole. For example, a given technique's ability to trade memory against the online time might not be the same at low and high success rates.

In some TMTO analysis [Hel80; Oec03; BBS06], only the worst case time complexity formulas are studied[1]. While considering the worst case is useful in the sense that it allows us to derive an upper bound on the behaviour of the trade-off, the obtained curve are very rough. It is well-known that dealing with complexity rather than the exact worst times is not very convenient in practice for an implementer when considering a limited set of parameters. The hidden constant of the time complexities can easily overturn the conclusion given by an asymptotic analysis. Moreover, even exact worst case online times are of limited interest in some cases. We saw that the probability of finding a pre-image in a given column of the TMTO is not always uniformly distributed, which means that there are cases – in particular when the probability of success is high – where the average online time is much lower than the worst case. Obviously, accounting for an early exit of the algorithm in such cases allows us to perform better estimations of the practical performance of the TMTO.

Another simplification made by [Hel80; BBS06] is to discard the impact of the false alarms on the behaviour of the TMTO. Although it greatly simplify the analysis, we know that the additional computations induced by the false alarms are hardly negligible in most cases. Taking false alarms into account implies acknowledging all the possibilities of merging between the potential chains and the chains of the TMTO matrix, and a part of the literature on TMTOs consists in limiting this behaviour. In fact, an important feature of the best possible trade-off is the ability to mitigate the additional computations due to these false alarms, and a proper comparison cannot be made without considering them.

The main TMTO variants, along with their improvements, have different memory requirement to store their tables. Recall that the online time is highly dependent on the number of chains, so the ability to store more chains in a given amount of memory has a non-negligible impact on the performance of the online phase. Therefore, accurately predicting how many bits a given TMTO table takes in memory is necessary to evaluate the performance of a TMTO. While these approximation concerns are arguably of minor impact compared to direct variations of the main parameters of the trade-off curves, such considerations can be important in practice during an implementation.

Beyond these considerations, establishing precisely the context in which we perform the comparison is important, as it helps reducing the scope of study to something more manageable. This is considered next.

[1] While no average case formula is given, it must be noted that the experiment in [Oec03] includes the average online time.

6.1.3 Context of comparison

We now make some observations on the usage of TMTOs in practice, which will help us to narrow our context of analysis during the comparison. The first one is related to the use of high success rate, and the second one deals with how the pre-computation cost is a secondary concern in our analysis.

TMTOs were initially introduced as a cryptanalytic tool to recover a part of a given set of DES keys [Hel80]. At the time, a full cryptanalysis of the cipher was not within reach of common computing capabilities. A later usage was found in cryptanalysis of stream ciphers. This gave birth to time-memory-data trade-offs. Again, for the attack to be successful, it was only necessary to recover the key of one of many ciphertexts. The key point in these early uses was that the success rate was not particularly high. Therefore, a majority of the literature on TMTOs choose parameters such that the probability of success is arbitrary but rarely close to 1.

With the introduction of the rainbow tables, the use of TMTOs shifted towards hash inversion, and in particular password cracking. The idea became finding a plaintext pre-image of a password hash from a given subset of passwords. Two problems can be considered with password recovery. Either we have a list of hashes and we want to invert at least one of them, in which case, the probability of success can be arbitrary depending on how many hashes are expected. The other problem is a more restrictive version of the first one: we have at least one hash, or a list of hashes, and we want to find a pre-images of as many of them as possible. Works such as [AJO05; AJO08], introducing the concept of maximal tables, pursued this way. In this latter context, we want to construct TMTOs that are as close as possible as dictionaries, whose probability of success is 100% by design. This can be achieved by setting a success rate close to 100% when designing the TMTO, notably by increasing the number of tables ℓ.

Another observation that can be made with the evolution of TMTOs is that the pre-computation phase is often thought of as a separate independent process during the computation of the TMTO. While the pre-computation may not be neglected altogether in the comparison, it is somewhat decorrelated from the online phase in the sense that it is not performed by the same entity, or with the same means as the online phase. Therefore, the pre-computation effort should is not considered as a metric in our comparison, as long as it is not prohibitive.

6.2 Preliminary comparison

Three TMTO variants are considered in this thesis: the Hellman trade-off, the DP trade-off and the rainbow trade-off. We now adopt a high level view, and we first consider these 3 variants that were studied in Chapter 3. In later section, a thorough evaluation will be made of the effect of improvements, but in this section, comparisons usually do not include improvements, and when they do, approximations are used. These effect are unlikely to dominate over the differences induced by the very different structures of the variants. Therefore, a preliminary analysis of the variants makes sense in order to focus on the variant that allows for the lowest online time at a given memory, when the assumptions made in Section 6.1, about the high probability of success, are taken into account. Once a suitable variant is chosen, the improvements will be considered for the variant. In order to find this best variant, we revisit the existing works that deal with TMTO comparisons. We first consider works performing asymptotic analysis, then study more refined comparisons based on average online time.

6.2.1 Worst case asymptotic analysis

In [BBS06], Barkan, Biham, and Shamir present an analysis of a general unified trade-off, with a novel way to view the chains. While this work is not intended as a comparison, the unification places each variant on an equal footstep, which allows Barkan et al. to express some remarks about which variant is more efficient. We present hereafter the broad lines of their model in order to give some context to their conclusions.

The set of chains of the matrix is described in [BBS06] as a *stateful random graph*, which is a directed graph in which each node is associated with a *hidden state*, also called *color* in [BBS06]. An arrow between the nodes in the graph corresponds to the application of the one-way function f. The hidden state corresponds to the key information required in the online phase to associate a specific point to the correct starting point. In the Hellman scheme, this corresponds to the index of the table in which the corresponding chain resides, while in the rainbow trade-off, this corresponds to the index of the reduction function that must be applied to the node to correctly derive the right chain of the matrix. In what follows, we denote with s the number of possible hidden states.

In [BBS06], Barkan et al. consider the distinguished point technique as an improvement technique applied to the Hellman trade-off, rather than a variant on its own, and they only deal with the Hellman and rainbow variants in their analysis. In particular, their results hold whether the DP technique is applied to the Hellman technique, or not. Thus, without loss of generality, we assume the chain length is constant in the unified view presented below.

We denote with F_i, for $i \in \{1, \ldots, s\}$, the linking function $f \circ r_i$. In a chain, the linking functions are used in the following order:

$$\underbrace{F_1 \ldots F_1}_{k} \underbrace{F_2 \ldots F_2}_{k} \ldots \underbrace{F_s \ldots F_s}_{k},$$

for some $k \in \{1, \ldots, t\}$. We can see that the Hellman trade-off, as well as the rainbow scheme, can be instantiated in this unified scheme, by choosing the right parameter k. In the Hellman scheme, we only have one color per chain ($k = t$). The rainbow scheme lies on the other end of the spectrum with as many colors as there are columns in the matrix ($k = 1$).

The authors note that, once the hidden state (or color) is discovered, the determination of the correct starting point takes only a small amount of time – below $\sqrt{T}$ – during the online phase.

Upper bound for the coverage The concept of hidden state is tightly linked to the probability of success of the trade-off. More specifically, the amount s of distinct possible hidden states for each node describes the coverage of the problem space $\mathcal{N}$ by the TMTO. A TMTO designed with a higher value of s is going to provide a better probability of success. Nevertheless, increasing s at the expense of N has a limited interest because the product sN is constant. Indeed the expected coverage of both the Hellman and rainbow scheme is provided by the value $\sqrt{sNm}$, so for the same success rate, increasing either s or m will reduce N. A main result of [BBS06] is that when choosing a one-way function f at random, with an overwhelming probability, the maximum coverage $\mathcal{C}$ of a matrix is given by

$$\mathcal{C} \leq 2\sqrt{sNM \ln(sN)}. \tag{6.1}$$

This implies that a TMTO built with the best configuration according to a given f, that is, the TMTO built by choosing the best starting points which give the best coverage, only give an advantage of $\ln(sN)$ over a random choice of starting points. In other words, there is not much to be gained from carefully

choosing specific starting points when building the trade-off, and the curve performance of the Hellman trade-off is already optimal within a logarithmic constant.

Lower bound for the online time From Equation (6.1), a lower bound for s is obtained. Assuming the success rate of the TMTO is at least $\frac{1}{2}$, the number of distinct states is bounded by

$$s \geq \frac{N}{32M \ln N}.$$

Furthermore, keeping this assumption on the success rate, Biham et al. give a lower bound of the online time. They consider that the online phase performs the search of the hidden states of each node sequentially, and assume that for a given node, the maximum number of operations required to find the hidden state is the maximum distance between any node with the same hidden state as that node and an endpoint. Note that this condition is respected in general. The online time, in the worst case, is then at least

$$T \geq \frac{1}{1024 \ln N} \frac{N^2}{M^2}.$$

It was shown in Chapter 2 that the main variants of the TMTO technique, the Hellman, the DP, and the rainbow trade-off follow an asymptotic curve of $T = O(\frac{N^2}{M^2})$. This lower bound tells us that the variants are close to the optimum up to a logarithmic factor. The work [BBS06] therefore demonstrates that the three variant are in fact asymptotically equivalent with regard to their worst case online time. This is a somewhat disappointing statement when we are trying to compare the variants.

Nevertheless, we can notice that, in this unified view, a Hellman TMTO using a number t of tables of size m with chain length t is equivalent to a single rainbow table of the same size. In particular, a single Hellman table contains an equivalent amount of information as a single column in the rainbow trade-off. This fact itself hints that rainbow tables are likely to perform better in high success rate situations. Indeed, a single Hellman table of maximal size is very difficult to obtain, whereas it is not difficult to obtain distinct elements in a rainbow column. A similar remark can be made for the distinguished point trade-off which behaves much like the Hellman trade-off in this regard. Note that these observations do not account for the size required for each scheme, and it might still be that the low space requirement of the DP trade-off compensates this disadvantage.

In [Oec03], Oechslin provides some arguments in favor of the rainbow trade-off compared to the Hellman and DP trade-offs. The rainbow trade-off is said to halve the required computations compared to the previous approach, as well as avoiding the overhead due to varying chain lengths in the DP trade-off. However, the storage requirement is considered to be the same for all the variants and fixed at 8 bytes, so the particularities of the trade-offs are not considered in this regard. In fact, these statements are challenged in [BBS06], where Barkan et al. claim that the storage requirement of the DP trade-off[2] is half of the one of the rainbow trade-off, and therefore the DP trade-off would be superior when taking the storage requirements into account. However, their storage analysis is quite rough, so it is not possible to reach a conclusion based solely on their arguments.

6.2.2 TMTO characteristic for high success rates

All the previous observations were made in the worst case, when the false alarm impact is neglected. Avoine, Junod, and Oechslin in [AJO08], propose a formal comparison of the Hellman, DP, and rainbow

[2]The authors of [BBS06] mention the Hellman trade-off, but it is clear in their argument that they are dealing with distinguished endpoints.

trade-offs. They rely on the average online time of the variants instead of their worst case, and take false alarms into account. They introduce the concept of the *characteristic* of a trade-off, which is a metric allowing the comparison of the different variants.

The work of Avoine et al. focuses on the perfect version of the trade-offs. They choose to include the perfect Hellman trade-off in their comparison, although there is no known practical construction that allows us to build perfect tables for the Hellman variant. They also only consider the case where the tables are of maximal size.

In the case of the rainbow trade-off, we already saw that the online average time could be approximated by a polynomial in t^2, in such a way that we can write (see Theorem 4.1 from Chapter 4):

$$T_{\text{rainbow}} \approx \gamma \frac{N^2}{M^2}, \tag{6.2}$$

where γ is dependent on the matrix stopping constant only. The matrix stopping constant is tightly linked with the success probability, in the way that we can consider $\gamma(P^*)$ as a function of the success rate P^*. This gives us the characteristic of the trade-off, which is defined in [AJO08] as the evolution of γ in terms of the success rate. The characteristic is defined for the perfect Hellman trade-off and the distinguished point trade-off using the exact same relation that is given by Equation (6.2), although such an approximation for these two trade-offs is only valid when the success rate exceeds 50%, and the memory is not close to N.

While the storage considerations are not mentioned in [AJO08], Avoine et al. note that, while the rainbow trade-off does indeed require more than the DP trade-off, the rainbow trade-off gains more than a factor 2 in online time because it is less affected by false alarms than the two other trade-offs, which was not taken into account in [BBS06], where the authors actually claimed otherwise.

The results of the comparison in [AJO08] are the following. For all the success rates, the perfect DP trade-off characteristic is higher than the two other variants' characteristics. For a success rate below 90% the perfect Hellman trade-off and the rainbow trade-off are alternatively better, but beyond a success rate of approximately 93%, the rainbow trade-off has a lower characteristic, with a noticeable difference for high success rates exceeding 99%.

6.2.3 Comparisons for arbitrary success rate

In [HM13], Hong and Moon gather the average online time complexities of the Hellman, the DP, and the rainbow trade-off in their non-perfect version, and exploit them in a thorough comparison. Contrarily to the previous comparison, they formalise the storage required for each scheme in order to evaluate how they balance running time against memory. Moreover, the pre-computation cost, which was overlooked in [BBS06; AJO08], is included in the analysis.

A first preliminary analysis is made using the approximated curves: $TM^2 \approx N^2$ for both the Hellman and the DP trade-off, and the curve $TM^2 \approx \frac{1}{2}N^2$, that was estimated in [Oec03], for the rainbow trade-off. Note that these approximations are the actual curves and not asymptotic complexities. These are derived from the common matrix stopping rules used with the TMTOs, which are $mt^2 \approx N$, and $\ell \approx t$ for the Hellman and DP trade-off, and $mt \approx N$, and ℓ small, for the rainbow trade-off. The work [HM13] study the source of the disagreement between [BBS06] and [AJO08] and establishes that the approximations of [BBS06] lead to inaccurate storage requirements.

While a brief summary of the checkpoint technique is mentioned in [HM13], Hong and Moon choose not to include this technique in the analysis, which is claimed to have a small effect on the final result. For their comparison, Hong and Moon use an approximated polynomial in t of the average online time.

Then, they propose a storage unification of the different variants by applying factors depending on m to the t^2 term of the polynomial. The indexing technique, presented in Section 3.2.2, is considered in their analysis, as well as the endpoint truncation technique. However, the storage estimates of these techniques is rough, and the truncation technique is assumed not to impact the online time, which restricts its use to low values for the number of truncated bits.

Using this unified storage scheme, Hong and Moon observe that the Hellman and DP trade-offs are actually very close to one another in terms of performance. The conclusion is not clear, since the DP trade-off performs slightly better in low success rates, and the Hellman trade-off is slightly more performant for higher success rates. In their optimal configurations, *i.e.*, for the parameters of each trade-off that minimise the term in t^2 of the approximated online time, the Hellman provides a lower value, reached by a higher success rate than what can be done with the DP trade-off. Then, the rainbow trade-off is compared to these two variants. They provide comparisons for various fixed probabilities of success for the rainbow trade-off, since it is determined by the number of tables ℓ which has no corresponding parameter in the other two trade-offs. The difference between the rainbow trade-off and the two other variants is more visible than between the Hellman and the DP trade-off. In all the presented comparisons, for success rates varying from 25% to 99%, the rainbow trade-off performs better in a majority of cases. Hong and Moon show that the superiority of the rainbow trade-off is more noticeable for success rates exceeding 90%, where the comparisons give the rainbow trade-off as the best trade-off for all configurations.

The same methodology, that is used in [HM13] for the comparison of the non-perfect variants, is reused in the work [LH16] of Hong and Lee, which extends the results to the perfect versions of the trade-off. The perfect version of the Hellman is not considered, so the comparison concerns the perfect DP trade-off and the perfect rainbow trade-off. As in [HM13], several comparisons are provided, for fixed success rates of the rainbow trade-off. Since the work is dealing with perfect trade-offs, which aims at increasing the success rate, only high success rates $> 90\%$ are considered. The results are overall the same as with the non-perfect version. No configuration of the DP trade-off provides a better performance than the rainbow trade-off.

6.2.4 Conclusion

The observations made in this section are sufficient to determine, without ambiguity, which variant performs best. The rainbow trade-off clearly presents a better performance, compared to the other trade-offs. It is especially suitable in our context, where we seek a high probability of success for our TMTOs.

We consider high success rates, and they come with their price in pre-computation, which may be the bottleneck of the design of the TMTO. In that case, a reduction of the problem space may be considered. While, with a reduction of the success rate, elements are randomly dropped from the pre-computation matrix, the reduction of the problem space is overseen by the designer who can use heuristics to evict elements that are less probable to be encountered. Inversely, the problem space can be constructed in a way that elements that are more likely to need inversion can be kept with certainty.

If one wants to reach a high probability of success with a TMTO, the perfect version of said TMTO optimises toward a high success rate while keeping the same parameters equal to the non perfect version. Indeed, they increase the ratio $\frac{\mathcal{C}}{M}$ of the pre-computation matrix, where $\mathcal{C}$ designate the coverage of the matrix, and M the amount of memory required to store the table. This gives opportunity for more performance as more distinct elements are browsed in the same amount of time, or on the same amount

of memory. Therefore, we settle on the perfect version of the rainbow trade-off in our comparison.

For a given table with a fixed success rate and bounded online time, one will want to use maximum tables. Ideally, these require to build every possible chain for each table, which means that $Nt\ell$ computations should be made, in the classical approach to build tables, that is without additional processing during the offline phase. As stated before, there is a way to come close to the maximal bound without resorting to compute every chain. This incurs a loss in distinct chains which in turn incurs a loss of probability of success. However, the gain in both pre-computation time, and the ability to store the starting point on less bits, makes us choose this particular setting.

6.3 Fixed memory model

We now introduce our comparison model, the *fixed memory model*. This model manages the different parameters in a way similar to what would be done in practice to determine the parameters of a TMTO given a set of resources. It is made in such a way that it allows us to provide a fair comparison of TMTOs using different optimisations. The choices made in Section 6.2, in particular the fact that we settled on the perfect rainbow trade-off, are applicable in this section. Moreover, in the following, we assume that we are dealing with a TMTO with a high probability of success.

The comparison within the fixed memory model is an iterative process. We establish the compatibility between the different improvement techniques, and select suitable combinations. We then consider the storage optimisations and determine their optimal parameters. Afterwards, the checkpoint technique is considered and incorporated into the existing combination. Then, again, the optimal parameters of the technique are determined by studying the observations on the reduced storage-optimising combination.

While the concept of the model is simple – as the name suggests, we fix the memory beforehand – the implications of such a choice on the comparison are not. These are discussed hereafter. We first explain how the parameters are fixed in the model, hence reducing the dimension of all the possible choices of the trade-off. Then, the combinations that will make it into our comparison are described. Once the combinations are determined, optimal parameters, for both storage improvements, and the checkpoint technique, are required, to fit a combination into the fixed memory. The details of these optimisations are discussed in this section. In particular, the checkpoint technique is analysed and simplified in order to yield an optimisation in reasonable time, whereas a naive approach would require a prohibitive time.

6.3.1 Reducing the dimension

Given a problem space and one-way function size N, the number of elements in a table m, the chain length t and the number of tables ℓ, if we use the same starting points, we are guaranteed to obtain a unique TMTO. The probability of success of each table is determined by the matrix stopping constant $\frac{mt}{N}$, and so the probability of success of the whole TMTO is derived from it using the ℓ parameter. The main contribution of this chapter is to take improvements of the trade-off into account. However, improvements affect the memory M required to store the table, and the average online time T of the online phase of the trade-off, while keeping the same parameters m and t. Therefore, we cannot use m and t directly in our comparisons, and should focus on the T and M parameters instead. The main idea of what we are trying to do is to determine sets of parameters N, m, t and ℓ which are equivalent to each other, when different improvements techniques are applied to the TMTOs. This equivalence consists in taking into account whatever gain is provided by a given improvement, in such a way that we can see clearly, by observing the parameters T and M of two versions with two different improvements,

which one gives the best results, even when the improvements do not target the same parameter, *e.g.*, one is aimed at reducing the memory, and the other is reducing the average online time.

If we want to get a meaningful comparison, we cannot let all the parameters of the TMTO vary at the same time. We have to reduce the dimension of the comparison by fixing some of the parameters. To choose what to fix, we remark that all the parameters do not have the same role in the trade-off curve. The problem size and the probability of success of each table, along with the number of tables, are more related to the pre-computation phase, in the sense that they only impact the resources needed to build the tables, whereas the online time and the memory required to use the TMTO are more linked to the online phase. The choice of the parameters relating to the offline phase is determined by whether there are enough resources to compute the TMTO matrix. It does not matter at this step whether t is small and m large, or the other way around. The choice of the proportion of m, or t, in the product mt is however important during the online phase, where these parameters influence the resources needed for the usage of the TMTO. These last two parameters, and their relation, constitute the trade-off between time and memory and provide a meaningful metric of the performance of the trade-off. This means that the parameters related to the offline phase should be fixed for a pertinent comparison between various modifications of a TMTO. In the case of the parameter N, it is easy to see that the evaluation of the performance of a trade-off when considering different problem spaces is not intuitive at all. Also note that fixing the problem size in the context of TMTO comparisons is a common choice [AJO08; HM13; LH16] as seen for the variant comparison in Section 6.2.

The impact of the probability of success of a TMTO over the other parameters is even less clear with regard to the performance of the trade-off, as it is somewhat hidden from TMTO curves where it is considered constant. A varying probability of success makes the analysis of the performance of two given TMTO impossible, since the probability of success is determined by both the matrix stopping rule and the number of tables, and has a high influence on the trade-off performance. A subtle variation of the probability of success is likely to overturn the comparison conclusions, given moderate differences between improvements. Therefore, it is important that the probability of success of a TMTO is fixed for a comparison. In addition, one has to be careful when dealing with improvements that could decrease the probability of success and thus not yield a fair comparison.

We are dealing with the perfect rainbow trade-off, and we chose, in Section 6.2, to restrict ourselves to high success rate TMTOs. With the perfect rainbow trade-off, the number of tables is always a small value $\ell > 0$. If we consider maximum tables, we can determine the maximum probability of success $P_{\max}$ given a number ℓ of tables of maximal size. These probabilities are given in Table 6.1, where we can see that we reach a probability of 99.97% in as few as 4 tables. A common choice is to use only 4 tables, since a loss of probability of success 0.03% is deemed acceptable. We will make the same choice and consider a number $\ell = 4$ of tables. In particular, this loss is negligible compared to the loss incurred when we choose to deal with table sizes that are close to maximal, but not maximal, as will be the case in our comparison.

Table 6.1 – Maximum probability of success given ℓ tables in the rainbow trade-off

ℓ	1	2	3	4	5
$P_{\max}$	86.47%	98.17%	99.75%	99.97%	$> 99.99\%$

The usage of tables of nearly maximal size has two benefits. First, it allows to substantially cut the cost of the pre-computation by computing a fraction of the chains that would be required for tables of

maximal size, while only forfeiting a few percents in the probability of success, in such a way that we are still in what can be considered a high success rate. Second, as seen in Section 3.2, it allows to store the starting points on fewer bits. We therefore assume that the tables are computed from $m_0 = Km_{\max}$ starting points, where $m_{\max}$ is the maximal size of the corresponding table. The matrix stopping constant is therefore dependent on K. If K is large, the matrix stopping constant will be very close to 2, but the obtained tables will not be of maximal size anymore, while being very close. The choice of K is arbitrary, but as long as any two given TMTOs with the same probability of success in the maximal size setting use the same K, their probability of success will be the same, and these two TMTOs can thus be compared. We choose to use a value of $K = 25$, corresponding to what was used in the same context in [AC14], also arbitrarily chosen.

Now that we have fixed some of the TMTO parameters, it remains to deal with the time-memory trade-off parameters T and M. If we want to reduce our comparison to one dimension, one of these two parameters has to be fixed. The roles of t and m, from which these parameters are derived, are somewhat symmetrical, since their product is constant when N is fixed, because of the matrix stopping constant. The fact that a linear variation of M quadratically impacts T has no incidence on the comparison when one of the two is fixed, so either of T or M can be fixed to perform the comparison. However, we believe that T is a more flexible value than M in practice, since it is always possible to wait longer for a result, whereas increasing the memory to fit a table is a more complex endeavour.

We therefore propose to reduce the dimension of comparison in a model with fixed memory. Our model makes the parameter T the main metric of performance of a given TMTO. To keep an idea of how the trade-off would perform, we fix M to several values corresponding to a practical amount of memory, so when comparing several improvements applied on the TMTO, only T is variable. The idea is to compare a TMTO with different improvements using the same M. We set an amount of memory M (in bits), in which each table (in different configurations) must fit. The goal is then to determine the maximum amount of pairs (S_j, E_j) that once compressed will fit in M, and, ultimately, what configuration minimizes the average online time T.

Our model in fact follows what would be done in practice by someone implementing a TMTO to fit in a target memory. It therefore provides a clearer view of how which improvement relate to each other with a practical setting in mind. With our model, we can deal with exact values for the storage, as would be done in practice, rather than using approximations, and we can therefore account for even smaller differences between the improvements. The flexibility of our model also allows us to combine different improvements, while being accurate when dealing with the storage requirements.

6.3.2 Combinations

Given several techniques aimed at improving TMTOs, one may want to stack them all on a single implementation, to further decrease the resources requirements. However, among all the improvements that were discussed, some target the same aspect of TMTOs, in such a way that they cannot be used concurrently. We now discuss how we can arrange the improvements together to form combinations that provide the gain of both techniques to a TMTO. We describe the combinations that make sense. Such combinations, along with the optimal parameters, lead to TMTO *configurations*, that will be studied in Section 6.4, in which we will compare them in order to determine which one offers the fastest online time.

We first focus on storage techniques applied to the *vanilla* version of the rainbow trade-off, *i.e.*, the rainbow trade-off without any improvements. Optimisation of starting point storage was already dealt

with when designing the fixed memory model, in the sense that using nearly maximal cases with the counter technique already gives us a reduction in storage requirement. We saw in Section 3.2 that compressing randomly distributed starting points would not yield any significant gain. Therefore, we do not apply any further improvements on starting points, and we only consider endpoint compression techniques. As for storage improvement, we discussed two lossless compression techniques and a lossy compression technique, which are recalled hereafter. The first lossless technique, the indexing method, mentioned in Section 3.2.2, is known to be used in the implementation landscape [TO05]. Another lossless compression is the encoding of the endpoints delta[3], presented in Section 3.2.3, which, while less known, provides a better compression of the endpoints. It is slightly less convenient due to the fact that it requires to split the table in several blocks of deltas, and comes with a slight overhead (that can be neglected in practice). The truncation of the endpoint is a lossy technique which was mentioned in Section 3.2.4. It consists in omitting part of the endpoint in such a quantity that the benefit from the storage gained outweighs the additional computations due to the loss of information.

The lossless compression techniques mainly rely on the fact that the endpoints are sorted. When using the truncation technique, if the omitted part consists of the least significant bits of the endpoints, then the order is preserved. Therefore, the indexing technique and the delta encoding can be applied on the truncated endpoints, thus effectively combining the lossy technique with any of the two lossless techniques. Note that the approach we take here with this combination is different from the one in [LH16], because we do not assume that the online time overhead is negligible.

In the MPHF rainbow technique, the table is made up of an MPHF and an array of endpoints' signatures. The description of the MPHF itself is, by design, optimally stored, so no additional storage improvement is necessary. For each signature at a given position in the array, the corresponding endpoint is neither correlated to the starting point, because the chains involve random functions, nor to the given position in the array, because the MPHF construction scheme imposes a random distribution on the image of the set of keys (the endpoints). Therefore, there is no apparent structure on the list of signatures that can be exploited by a compression scheme. Since the MPHF rainbow technique does not store the endpoint at all, the endpoint truncation technique cannot be considered. This means that the MPHF rainbow scheme is not compatible with any of the storage techniques that were mentioned above.

We now consider improvements that deal with reducing the computations that are performed to verify alarms during the online phase. These are the checkpoint technique, presented in Section 3.3, and its generalisation, the fingerprint technique, presented in Section 3.4. These two techniques keep additional information on the chain alongside the endpoint to help distinguish between plausible potential chains, and chains that would have merged somewhere with a chain belonging to the TMTO matrix. The checkpoint technique is independent from the storage of the endpoint, and blends nicely with the storage combinations established for the classical rainbow trade-off. The checkpoints are stored alongside either each suffix in the indexing technique, or next to each encoded delta, in its blocks, for the delta encoding method. The same method applies whether the original endpoints are truncated or not.

Note that we do not explicitly deal with the fingerprint improvement in this work. We assume 1-bit checkpoints and a possibly truncated endpoint. We saw in Section 3.4 that this degenerate version was considered to behave like an optimum based on heuristic observations from the original publication that introduced fingerprints [ABC15]. Taking these observations into account, we will treat the fingerprint technique indistinguishably from the combination of the checkpoint and truncated endpoint techniques. Therefore, our comparison results include the fingerprint method, such as implemented in [ABC15]. In

[3]While it was called *compressed delta encoding*, no further compression is applied on the encoded delta in the scheme, so we only refer to the technique as *delta encoding*.

particular, our results are compatible with [ABC15].

The addition of the checkpoint technique into the MPHF rainbow scheme is also meaningful, even though this scheme does not involve storing endpoints. In the case of the MPHF technique, the checkpoints are in fact of the same nature as the endpoint signature. The checkpoints characterise the TMTO chain associated to a given endpoint in the same way as the signature characterises the endpoint. In a sense, the signature is very similar to a fingerprint, except for the fact that it is stored alongside a starting point instead of acting as an endpoint in the fingerprint scheme.

Table 6.2 – Combination of improvements for the rainbow trade-off

name	indexing	delta encoding	truncation	checkpoints
vanilla				
fingerprint			✓	✓
storage combination 1	✓		✓	
storage combination 2		✓	✓	
full combination 1	✓		✓	✓
full combination 2		✓	✓	✓
MPHF	signature			✓

The different combinations proposed are gathered in Table 6.2. While not a combination, the vanilla version is given for reference. Each column corresponds to a different type of improvement, *i.e.*, the compression techniques, and the checkpoint technique. We distinguish between the MPHF based rainbow trade-off, and the original rainbow trade-off, *e.g.*, the one used in the vanilla version. Each line corresponds to a distinct combination, incompatible with the others, that will be, once the optimal parameters are determined, one of the TMTO configurations compared in Section 6.4.

Other improvements, such as the ones used on TMTOs exploiting a non-uniform problem space, are not considered in this analysis since they are highly dependent on both the problem space, and the context of use of the TMTO. Indeed, the interleaved technique relies on the probability of a given image, whose preimage is sought, to be encountered in practice. Such consideration cannot be taken into account in a problem-agnostic comparison, such as the one made in this chapter. Overall, the analysis of the interleaved technique consists in providing the average online time of two distinct TMTOs, given the image distributions in practice. Therefore, the methodology of this analysis should be applicable to whatever TMTO combination happens to perform the best, if one is presented with such a skewed problem space that it allows the interleaved technique to reduce the average online time substantially.

In our comparison, each table of a same TMTO are of the same size, and with the same chains length. The heterogeneous technique, presented in Section 3.6, is not included in any configuration. This technique lowers the average online time at the expense of a worst case of the online phase that takes more time. This effect cannot be taken into account by the fixed memory model which deals with the average online time, for reasons discussed in Section 6.1. Including a technique which offers a gain without being able to account for its drawback would not be fair in our comparison. However, it is unlikely that the technique's gain changes significantly across the different combinations. Since the improvements are applied independently on each of the tables, if one is willing to trade the worst case for a better average time, the heterogeneous technique can be applied to any of the combinations, after the other improvements.

6.3.3 Optimised storage parameters

We now discuss in more detail the combinations involving storage reduction. In our approach, we first deal with the storage optimisations before applying the checkpoint technique. We now discuss the methodology used to obtain the optimal parameters of these partial combinations, concerning only the storage techniques. This methodology will be used when we expand these partial combinations with the checkpoint technique, to obtain full combinations.

There is a side effect of using the truncation technique in a perfect table setting. Indeed, we can expect collisions of truncated endpoint, which, once the duplicates are removed, incur a loss in probability. As we do not consider strictly maximal size tables, we can compute additional chains beforehand to compensate this loss. Therefore, in our comparison, in order to keep the same probability of success after the truncation, we use slightly more starting points to compute the table whose endpoints are to be truncated, in such a way that the table with truncated elements has the same number of endpoints as an equivalent table whose endpoints are not truncated.

We first provide a preliminary comparison of the endpoint compression schemes combined with the truncation technique. The idea is to assess the impact of the truncation technique on the lossless compression schemes. We give an example of the size obtained when using both methods in Table 6.3, for several amounts r of omitted bits of the endpoint for the truncation. The larger the elements are, the better they are compressed, so we use a somewhat large problem size of $N = 2^{56}$ to highlight the differences between the techniques.

Table 6.3 – Size in bits of an endpoint for $N = 2^{56}, m = 10^9$

	$r = 0$	$r = 4$	$r = 8$	$r = 10$
naive storage	56	52	48	46
indexing technique	33	29	25	23
delta encoding	28	24	20	18

The column with $r = 0$ in Table 6.3 corresponds to the results for the case where compression techniques are not used, and are given for reference. We observe from the other results, where $r > 0$, that the compression gain in bits is about the same as with non truncated endpoints, since the size obtained corresponds to the compressed size of the full endpoint truncated by as many bits as the endpoint itself. Since the values are given in bits, they have to be integers, and the precision of the evolution of the storage in terms of truncated bits is reduced, in the sense that the corresponding real-valued optimal parameter may be more or less close to the nearest integer. In particular, if the theoretical parameter obtain by a compression scheme corresponds to, *e.g.*, 22.5 bits, we have to store each endpoint on 23 bits, and as much as $m \times 0.5$ bits are lost since that they are not used to store information.

Each starting point in a table is stored on $\log_2 m_0$, where m_0 designates the number of starting points used. In our fixed memory model, we settled on using tables of nearly maximal size with a counter to choose the starting point, in such a way that $m_0 = Km_{\text{max}}$, for some $K > 1$ such that $Km_{\text{max}} \ll N$, so that we have $\log_2 m_0 < \log_2 N$ and we save space when storing the starting point, compared to a truly maximum table. However, this technique makes so that the storage of the starting depends on m_0, rather than on N. In particular, if m_0 is increased significantly, the storage of the starting point also increases since each endpoint will have to be stored on more bits.

Due to the dependence of the storage of the starting point on m_0, the truncation technique has an

additional impact on the storage of the table. When two endpoints are truncated, there is a chance that their truncated versions are equal if the most significant bits of the two original endpoints are equal. This means that a set of truncated endpoints is going to be smaller or of the same size as the set of original endpoints. In our model, we aim at providing an equal probability of success, and therefore an equal coverage of the TMTO matrix. This is the case if we assume that all the truncated endpoints of a table are distinct. However, this requires that more starting points are computed to provide more non-truncated endpoints, which, once truncated, give m elements, so that the matrix stopping constant is the same as with the other combinations. The overhead on m_0 increases exponentially with the truncation parameter r, so there is a threshold beyond which removing a bit from the endpoint corresponds to requiring an additional bit to store each starting point. Then, the truncation technique is only interesting if we stay beyond this threshold.

We now present the procedure used to determine the parameters of the combination of the truncation with any of the two compression techniques. Consider the amount of bits necessary to store an entry in the table. This quantity depends on the table parameters N and ℓ, on the number m of pairs, on the method used to store the endpoints (indexing technique or delta encoding), and on the number r of bits truncated. Starting points are stored on $\log m_0$ bits. The number of endpoints m that can fit in M can be expressed as:

$$m = \left\lfloor \frac{M}{\log(m_0) + w} \right\rfloor,$$

where w denotes the part of the entry that is not the starting point. It consists in either the endpoint for the classic rainbow table, or the signature in the case of the MPHF rainbow table. In the classic rainbow scheme, w itself depends on m (e.g., m is a parameter of both the lossless compression techniques) we solve the above equation to obtain m for a given parameter r for the two endpoint compression techniques (all other parameters being fixed). Solving the equation analytically is difficult, but it can be efficiently approximated numerically by iteration.

The next step is to determine the optimal truncation parameter r^* for which the average online time T is minimal. Higher truncation means a smaller w (and thus ultimately a higher m, for which T decreases), but also incurs an overhead on T. The search space is small (integer bounded between 0 and the size of the endpoint), so an exhaustive search can be used efficiently to determine r^*. A modest truncation reduces the online time up to a value r^*, but the overhead prevails for $r > r^*$. This phenomenon was already observed in Figure 5.5 of Section 5.4.3, when the MPHF technique storage requirement was compared to the truncated endpoint technique.

6.3.4 Checkpoints optimisation

Now that the storage improvement combinations have been studied, we discuss the next layer, which is the checkpoint technique. This layer adds a lot of complexity to the combination, which can result in a prohibitive amount of computations for the optimisation of the parameters. Our goal is to speed up the checkpoint-related optimisation in the fixed memory model by leveraging some heuristics, which are determined hereafter with an analysis of the technique.

Curse of dimensionality In the checkpoints technique, a set of n positions $c_1, c_2, \ldots, c_n$, among the t columns of the TMTO matrix, is determined. The more checkpoints there are, the better the decrease of the online time is. However, each checkpoint increases the amount of storage required by each entry of a table. In addition, the different checkpoints arrangements perform very differently with regard

to the online time reduction. Therefore, the checkpoints are a rather complex parameter to consider, since they affect both the storage and the online time, and for each given number of checkpoints, an optimisation procedure is required in order to determine the best distribution of the checkpoints in the TMTO matrix. Unfortunately, when including the checkpoint technique, our approach suffers from the so-called *curse of dimensionality* [BCC57], in the sense that the determination of optimal parameters for checkpoint technique requires consequent calculations. In what follows, we discuss the adjustment of our methodology to deal with this problem.

Cost of checkpoint optimisation We saw that the average online time for the rainbow trade-off with the checkpoint technique can be computed with a formula such as the one from Theorem 3.10, which computes the average online time in $O(t)$. For large t such as the ones used in the problem size range that we are interested in, the computation of the online time is not negligible, because performing tens of thousands of online time computations to obtain a result is not practical. A closed approximation exists for the non-checkpoint version of the trade-off. However, the same thing for the online time with the checkpoint technique seems unlikely due to the discrete nature of the checkpoints because each section between two endpoints requires a different treatment for the computation of the false alarms. For a given number n of checkpoints, an optimal set of positions must be determined from the $\binom{n}{t}$ possible configurations by relying on the average online time obtained with this configuration. For large t, when n increases, the number of configurations is important, and the related optimisation problem requires more computations of the formula to yield an optimal position. This can become a problem in our fixed memory model, where such an optimisation is already performed in order to determine the truncation parameter. Indeed, given the problem sizes (and thus values of t) we are dealing with, determining, for each possible combination, the arrangement of checkpoints in addition to determining the optimal truncation parameter, and the number of checkpoints (which influences the storage) would be difficult in reasonable time. In what follows, we study the behaviour of the checkpoints parameters in order to overcome this issue. We reduce the problem so that we only have to deal with the number of checkpoints, instead of both their number and their optimal arrangement.

Table 6.4 – Optimal position c_{opt} (starting from the end of the table) of a single checkpoint for various configurations sharing the same matrix stopping constant, with $N = 2^{25}$

t	m	$\frac{mt}{N}$	c_{opt}	$\frac{c_{\text{opt}}}{t}$
500	128849	1.92	52	0.104
1000	64424	1.92	105	0.105
2000	32212	1.92	210	0.105
5000	12884	1.92	525	0.105
10000	6442	1.92	1051	0.105
20000	3221	1.92	2102	0.105

Stability of the optimum It was shown in [Hon10] that optimal positions for a given checkpoint were approximately linear in t. Therefore, the relative position of the checkpoints, *i.e.*, all the values $\frac{c_1}{t}, \ldots, \frac{c_n}{t}$, are expected to be constant in this context. Recall that in our model, the matrix stopping constant $\frac{mt}{N}$ as well as N, are fixed across the various combinations. In other words, the optimal relative distribution of checkpoints will be the same. In Table 6.4, we give the optimal position of a single

checkpoint placed at position c_{opt} for various configurations, sharing the same matrix stopping constant. These optimal configurations were computed using the formula of Theorem 3.10. We observe that the first relative position, for $t = 500$, differs from the others. We can see from the second line that the optimal position is $c_{\text{opt}} = 105$, so this discrepancy is due to the rounding, since $\frac{\lfloor 105/2 \rfloor}{t} \neq \frac{105}{2t}$. This highlights the caveat of establishing optimal configurations with small values of t: the smaller t is, the lower the precision of the optimal configuration is. Inversely, we can notice that for $t = 10000$ and $t = 20000$, the optimal positions are slightly displaced from the "constant" $\frac{c_{\text{opt}}}{t}$. One must therefore be careful when extrapolating optimal positions. We quantify the impact of these inaccuracies next.

Any given set of n checkpoints can be represented by a vector of $\{1, \dots, t\}^n$. The set of all the possible configuration vectors can be represented in the usual euclidean space, and for simplicity, we will place ourselves in $\mathbb{R}^n$ to represent checkpoints' configurations. We consider the usual 1-norm $\|\cdot\|_1$ on $\mathbb{R}^n$, which we will refer to by its common name, the *Manhattan norm*, that is given by

$$\forall x \in \mathbb{R}^n, \quad \|\mathbf{x}\|_1 = \sum_{i=1}^{n} |x_i|.$$

We refer to by *Manhattan distance* the distance associated with the Manhattan norm. This distance allows us to quantify the difference between two sets of configurations of checkpoints. We can, *e.g.*, state that if we consider the relative configuration of a single checkpoint c, given by $\frac{c}{t} = 0.105$, as it was the case in Table 6.4, the exact configuration of the last line of Table 6.4 differs by 2 from this configuration. In order to give an idea of the impact of the variance of the checkpoints, we observe the variance of the corresponding online times.

Relative error The loss of precision incurred by a low t, as illustrated in Table 6.4, can easily be quantified, if we possess an exact optimal configuration $(c_1, c_2, \dots, c_n)$ of checkpoints for t, (*e.g.*, $t = 10000$), we can extrapolate that a configuration $(\frac{t'}{t}c_1, \dots, \frac{t'}{t}c_n)$, will be close to the optimal configuration at t'. We denote hereafter with t_0 the starting configuration for which we possess an exact optimum. For example, extrapolating our configuration for $t = 100000$, we know that the optimum differs at most by a Manhattan distance of $10n$ from the configuration at $t_0 = 10000$. Then we can determine an upper bound on the inaccuracy of the online time due to a slightly inaccurate configuration of checkpoints. To do this, for a chain length $t' > t_0$, compute a set of configurations that are of Manhattan distance $d = \left\lfloor \frac{nt'}{t_0} \right\rfloor$, and evaluate their standard deviation. This quantity, divided by the online time of the optimal configuration, corresponds to what we refer to as the *relative error*, which gives us an estimation of the impact of an error on the online time. In other words, we can quantify whether it is worth computing the optimal configuration at a higher precision, in terms of the number of checkpoints, and the chain length. The relative error is illustrated in Figure 6.1. For each value of t, the relative error $\tilde{d}$ is determined with a distance $d = \left\lfloor \frac{nt}{t_0} \right\rfloor$, with a starting configuration at $t_0 = 10000$. The three example cases of $n = 1$, $n = 5$, and $n = 10$ checkpoints are considered.

Each dot in Figure 6.1 corresponds to a combination (n, t, d), where d denotes the distance from the exact configuration at this chain length, compared to the approximation given by the configuration at $t_0 = 10000$. The distance, not represented on Figure 6.1, is $\left\lfloor n\frac{t'}{t} \right\rfloor$ for each combination. Expectedly, we can see that the more checkpoints there are involved, the more variation there is among the possible configurations, and the further they are from the optimal configuration. For the cases $n = 5$ and $n = 10$, we observe that the relative error is nearly constant, hence independent from t. We can infer that this

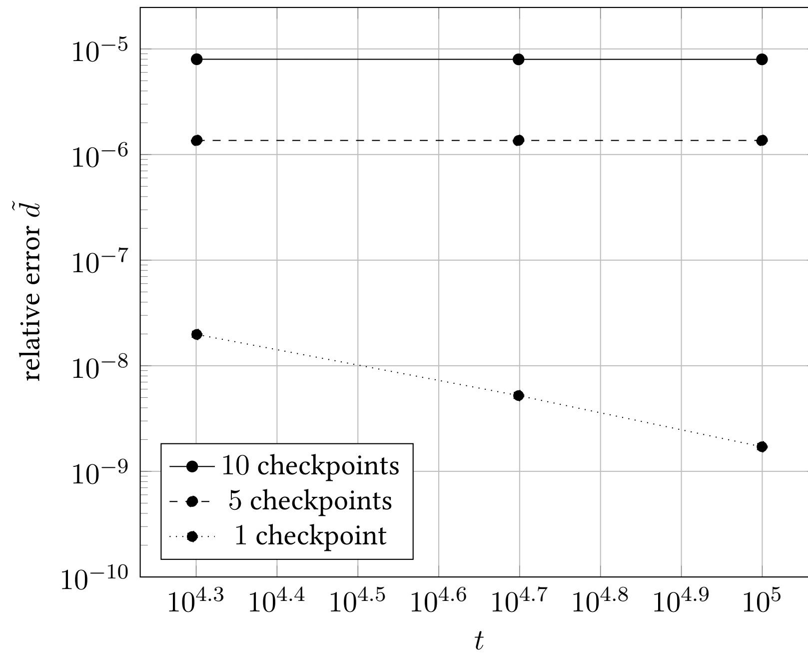


Figure 6.1 – Relative error on the online time with regard to the lack of precision of the checkpoint configuration, for $\frac{mt}{N} = 1.92$, and an exact optimal at $t_0 = 1000$

phenomenon corresponds to a convergence of stability when n increases, so that the situation is similar for any $n \geq 5$. This means that beyond a few checkpoints, the relative error only depends on n. The relative error values given in Figure 6.1 do not exceed 0.1%, so the error induced by up to 10 checkpoints can be neglected, but the pattern is not clear, so it is insufficient to conclude when n increases further. However, we can establish the relative errors in terms of n, by computing the optimal position of a single t, which applies to all values of t (since the relative error is independent from t). This is what is done in Figure 6.2, which presents the relative errors of the loss of precision when considering various numbers of checkpoints. The results from Figure 6.2 tell us that the loss in precision is somewhat moderate. By extrapolating the values based on the curve, we can see that the relative error is unlikely to exceed 0.01% for a few dozens of checkpoints. Then our approximation of the relative position holds even for more checkpoints than evaluated in Figure 6.1. Therefore, for any checkpoint position determination for values of $t < 10t_0 = 10000$, we will rely on the pre-computation of these relative positions, determined with $t_0 = 10000$.

The fact that the optimal performance is stable with both the matrix stopping constant, and slight variations of the checkpoint configurations, allows us to pre-compute the optimal positions of checkpoints. If we use a high enough t, we can be confident that the different combinations of storage improvements for various parameters will perform best with this particular configuration of checkpoints. This allows us to effectively offload a part of the process of finding optimal parameters for a given configuration, as we now only have to determine the right number of checkpoints to use for a given combination, and plug in the pre-computed optimal configuration, with the guarantee that it is indeed optimal up to a negligible error.

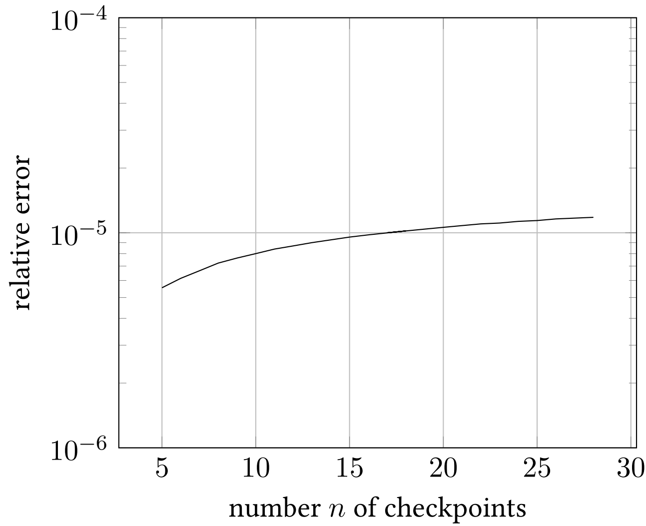


Figure 6.2 – Average relative error in terms of n

Optima with different matrix stopping constants While this is not a concern in the fixed memory model, let us briefly discuss the behaviour of optimal configurations of checkpoints considering different matrix stopping constants. It makes sense that the optimal position varies for a varying probability of success of a TMTO. Indeed, for higher success rates, we are likely to find an answer further on the right of the table, since more of the problem space is covered for a given range of columns of the TMTO matrix. However, this variation is very small, and the overall optimal position is somewhat the same in all of the high success rate range (>95%). In Figure 6.3, we give the evolution of the optimal position of an example of five checkpoints for a matrix stopping constant varying in the high success rate range.

We can see that, for a matrix stopping constant varying from 1.9 to 1.99, the optimal configuration barely changes. Indeed, all the positions of the highest success rate corresponding to 1.99 are within 1% of difference from the values at 1.9. We can also observe that the changes are gradual, in the sense that, for each change in the configuration, only one position shifts. In other words, the evolution of the configuration is made by atomic steps, *i.e.*, two consecutive configurations are distant from each other by at most 1, for the Manhattan distance. As a consequence, it is possible to reuse a computed optimal configuration for a slightly different matrix stopping constant. Alternatively, reusing any of the optimal configurations that were computed from a success rate that is not too far from the target can yield a significant speed-up in the iterative process of finding the optimal configuration at a given success rate, since the distance between the two configurations will be very low.

Given these considerations on checkpoint configuration determination, we can now complete our comparison procedure. With the checkpoint technique, an entry in a table is given by $\log_2 m_0 + w + n$, for n 1-bit checkpoints. Then, for a given amount of memory M that must not be exceeded, the number of entries in a table becomes

$$m = \left\lfloor \frac{M}{\log_2 m_0 + w + n} \right\rfloor .$$

In order to determine m, we have, for each combination of (r, n), where r is the truncation parameter, to find a suitable m. From this m, we determine t using the matrix stopping constant, to obtain an optimal configuration of each technique.

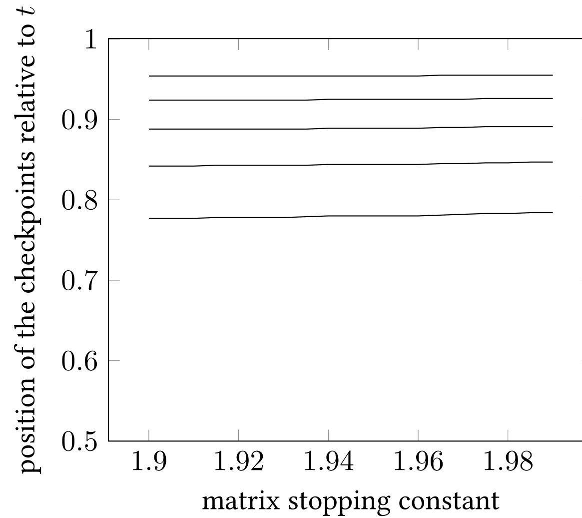


Figure 6.3 – Evolution of the optimal position of five checkpoints in terms of a high success rate matrix stopping constant

6.4 Comparison

We now have all the elements we need to perform the comparison of all the combinations of improvements that were described in Section 6.3.2. After establishing the parameters that will be used in the fixed memory model, as well as recalling the context of the comparison, we present the performance of the various improvements. The comparison follows the layout of the presentation of the configurations in Section 6.3. The improvements are first studied in isolation, then the storage techniques are combined and their performance evaluated. Finally, the checkpoint technique is incorporated in the combination to obtain the final comparison.

6.4.1 Parameters and terminology

We present the parameters of the comparison, and recall the context hereafter, in agreement with the choices that were made in Section 6.3. The perfect rainbow trade-off is used. We choose to perform our comparison on a problem of size $N = 2^{48}$. This is an arbitrary choice, but we believe it is representative of a large, yet practical space that can be considered today. The amount of memory per table is fixed at $M_{\text{max}} = 64$GB. This is a somewhat large value for commodity computers, but such a problem size is not designed with these in mind. It is, however, within the order of magnitude of what can be found in server grade computers, which would be considered for this kind of problem. We consider a trade-off using $\ell = 4$ tables. Therefore, the maximum probability of success is 99.97%. We assume that the tables are pre-computed using the counter technique when determining the starting points, and that tables of nearly maximal size are sought, by computing $25m$, where m is the wanted number of distinct elements in a given table. The constant 25 is chosen arbitrarily (see Section 6.3), and allows to reach about $0.96m_{\text{max}}$ distinct elements in each table, where m_{max} designates the number of elements in a table of maximal size, thus keeping a high probability of success. The matrix stopping constant, which depends on the probability of success, is therefore $\frac{mt}{N} = 1.92$. We refer to this set of parameter as the fixed memory parameters, and reuse them throughout this section.

All the parameters that were mentioned are sufficient to determine any of the (m, t) parameter pair that satisfy our fixed memory model, and thus allow us to compare the TMTO improvements. The studied configurations, *i.e.*, the combinations of improvements along with their optimised parameters, are those described in Table 6.2, The terminology of Table 6.2 will also be reused here. We will call *classic rainbow trade-off*, the vanilla rainbow trade-off, along with its improved derivatives, in contrast to the MPHF rainbow trade-off.

6.4.2 Standalone improvements

Now that the parameters of the comparison have been specified, we start by reviewing all the improvements, in isolation, in the context of the fixed memory model. We determine the optimal parameters for each of them. This gives us a preliminary vision of the impact of the improvements in a standalone setup, *i.e.*, when only one improvement is applied on the vanilla trade-off. The case of the vanilla trade-off is also considered, as a reference.

Vanilla version For the vanilla case, only the starting point storage depends on m, so we compute

$$\max\{x \in \mathbb{N}^* |\, x(\log_2(25x) + \log_2 N) \leq M\}$$

to find the optimal m. Then, we deduce t using the matrix stopping constant $t = \left\lceil \frac{1.92N}{m} \right\rceil$. Note that there is a slight inaccuracy due to the fact that we round to an integer, but the typical size of t makes so that it can be neglected altogether. The cases for the lossless compressions are variations of the vanilla case, except that the functional equations that have to be solved are more complex. In particular, the optimal parameters for both the indexing technique, and the delta encoding technique, are only dependent on m and N (and not t). Therefore, their only impact on the online time is the reduction of storage that they induce.

The truncation and the checkpoint technique require more attention, as they directly affect the online time, so determining their optimal parameters involves the online time estimation. We start with the classical version with truncation, the indexing technique, and the delta encoding.

Truncation method We first deal with the truncation technique. It is parametrised by r which denotes the number of least significant bits that were removed from the endpoint. To get the optimal value, we determine a pair (t, m) that acknowledges the removed bits, and fits in the memory $M_{\max}$. We then compute the online time of this configuration. The results are presented in Figure 6.4.

The decrease of the online time for each bit removed is almost linear until we reach a minimum. Afterwards, the online time increases due to truncation-related false alarms. The further we remove bits, the worse the impact of the alarm will be on the online time. We can therefore assume that the behaviour of the truncation method curve is unimodal.

Checkpoint technique In the checkpoint technique, both the number of checkpoints and the position of checkpoints influence the online time. In Section 6.3, we reduced the problem of establishing the optimal parameters to determine only the number of checkpoints, by using pre-computed optimal positions for a given number of checkpoints. Therefore, the treatment of the checkpoint technique is similar to the one of truncation. We first account for the additional memory storage induced by the checkpoints, and we find (m, t) pairs that fit in $M_{\max}$. Then, we iterate on the number of checkpoints and obtain the online time for these parameters, and the pre-computed optimal positions for this number

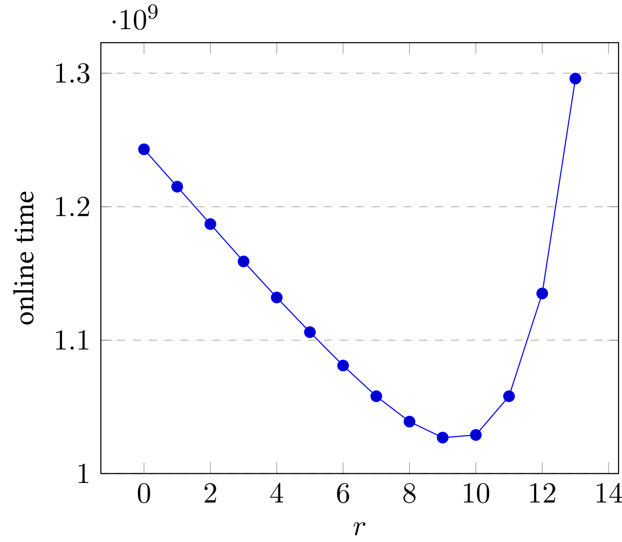


Figure 6.4 – Online time with a varying truncation for the classic rainbow trade-off with the fixed memory parameters ($M = 64$GB, $N = 2^{48}$)

of checkpoints. Here the pre-computed relative positions of checkpoints were so with a $t_0 = 10000$. The results are given in Figure 6.5. Once again, a clear minimum is seen. Beyond a certain point, the additional bits required to store the checkpoints are not compensated by the reduction of the online time. This behaviour is likely to remain the same for even more checkpoints. As with the truncation technique, we can assume that the curve for this parameter is unimodal.

Thanks to the fact that the curves of the parameters for both the checkpoint technique and the truncation technique are unimodal, we can speed-up the process of determining the optimal parameter. Indeed, instead of trying every possibility, we can settle on an early-exit algorithm which computes the parameter in an increasing order, and concludes to an optimal as soon as the next online time value exceeds the current one. Such an algorithm is used in the following to obtain the configurations for each improvement, and then later for the combinations.

We now present the results of the performance for each of the aforementioned improvements, in their standalone configuration. The data are gathered in Table 6.5. For each technique, the equivalent amount of storage required by each of the m elements in the table, in bits, is given in the "entry size" column. The optimal pair such that

$$\text{entry size} \times m \leq M_{\text{max}}$$

is given, as well as the online time T that is achieved by the optimal configuration.

The parameters that achieve the minimum online time can be deduced from the entry size of the vanilla method compared to the entry size of each technique. The truncation method is optimal with $r = 86 - 77 = 9$ truncated bits, and the checkpoint technique is using $n = 95 - 86 = 9$ checkpoints. We notice that the improvements on their own can reduce the online time of the vanilla method, up to 59%, in the case of the delta encoding method. The lossless compression techniques are far more efficient than the other two techniques, so if only one of these improvements had to be chosen for an implementation, compressing the endpoint is unequivocally the best approach.

Note that the fingerprint technique, as well as the MPHF technique, were left out as these techniques consist themselves in a combination. Indeed, recall that the fingerprint technique as used in practice

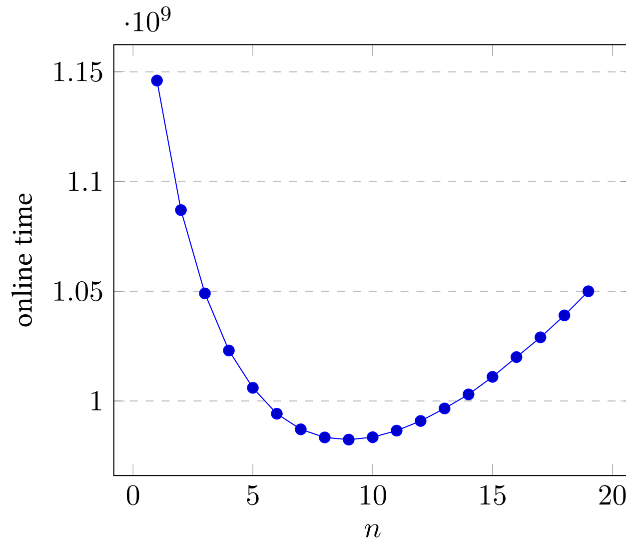


Figure 6.5 – Online time with a varying number of optimally positioned checkpoints, for the classic rainbow trade-off with the fixed memory parameters ($M = 64$GB, $N = 2^{48}$)

Table 6.5 – Standalone improvement performance in the fixed memory model

version	entry size $\lfloor \frac{M}{m} \rfloor$	m	t	T
vanilla	86 bits	$6.014 \cdot 10^9$	89857	$1.243 \cdot 10^9$
truncation	77 bits	$6.711 \cdot 10^9$	80525	$1.027 \cdot 10^9$
checkpoints	95 bits	$5.448 \cdot 10^9$	99207	$9.824 \cdot 10^8$
indexing technique	60 bits	$8.721 \cdot 10^9$	61973	$5.913 \cdot 10^8$
delta encoding	55 bits	$9.407 \cdot 10^9$	57451	$5.082 \cdot 10^8$

corresponds to the combination of the truncation technique and of the checkpoint technique. The MPHF technique is too different from the other ones to associate it with existing improvements, but the preliminary analysis made in Chapter 5 showed that it exhibits a similar behaviour of the truncated point technique, while proposing a reduced storage, like the compression techniques. These techniques are included in the combinations' comparisons hereafter.

6.4.3 Storage-oriented configurations

We now compare the storage improvements for studied combinations. These improvements are the lossless compression techniques and the truncation technique. The methodology is the same as for the determination of the optimal truncation parameter, except the memory storage takes the compression techniques into account. Figure 6.6 shows the optimal parameters found for several pairs (N, M). In addition to the parameters given above that we use for the comparison, we give the results for problem of various size and maximum amounts of memory, which allows us to observe the evolution in terms of N, The compression method performs better for larger N, and the behaviour might change accordingly. For clarity, we only provide the figures for the delta encoding technique, but the results are similar for the indexing technique, as it was hinted by their similar behaviour in Table 6.3, in Section 6.3.

We observe that the optimal value of r increases exponentially with the problem size, so the com-

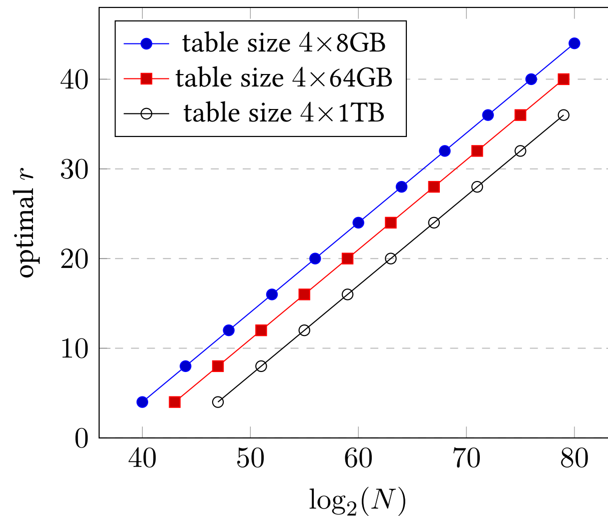


Figure 6.6 – Optimal truncation parameter using delta encoding, for different table sizes

bination brings a flat benefit, represented by the fact that the curves in Figure 6.6 are affine. In other words, there is a constant part of the endpoints, for a given configuration, that is expendable, in the sense that this part is better used to store additional chains. This result indicates that the behaviour of the combination of one of the lossless techniques with the lossy compression techniques is likely to be the same for a wide range of contexts.

Table 6.6 shows some results for the same (N, M) pairs as in Figure 6.6. For a simpler comparison, the values for the compression without truncation, *i.e.*, for $r = 0$, are given. The optimal parameter is indicated, as well as the online time obtained by this configuration. We can observe from the results that the optimal parameter seems to be independent from the technique used, and we also notice that the optimal parameter in the $N = 2^{48}$ case, is the same as the truncation technique without compression, given in Table 6.5. This indicates that the optimal truncation depends only on N, and not on the compression scheme that is used. The online time gained from the combination ranges from 11% and 12% in the small problem case, for the indexing and the delta encoding technique respectively, to 32% and 34% in the larger case. This means that the combination gives a substantial gain compared to the individual techniques taken in isolation.

A clear takeaway from Table 6.6 is the superiority of the delta encoding technique compared to the indexing technique. Nevertheless, the literature on TMTOs, and especially the one dealing with implementations, seldom mentions the usage of this technique, while the indexing technique seems to be quite popular in the implementations. While the delta encoding technique is less convenient to implement, and requires more tuning to reduce the online overhead, its performance indicates that such a effort yields a significant benefit.

6.4.4 MPHF based rainbow trade-off

We now deal with the MPHF technique. There is only one parameter to optimise in this case, which is the signature size b, in bits. We determine this parameter in the same way as for the truncation parameter. However, the MPHF technique relies on an algorithm that is external to the pre-computation process, in order to construct the MPHF. Depending on the parameters that we choose to build an MPHF

Table 6.6 – Performance (in terms of online phase cost) of lossy and lossless compression techniques, using a fixed memory.

(a) $N = 2^{40}$, $M_{\text{max}} = 8\text{GB}$

compression	m	r	T
none	$8.611 \cdot 10^8$	0	$9.266 \cdot 10^5$
prefix-suffix	$1.267 \cdot 10^9$	0	$4.285 \cdot 10^5$
delta encoding	$1.380 \cdot 10^9$	0	$3.610 \cdot 10^5$
none	$9.090 \cdot 10^8$	4	$8.560 \cdot 10^5$
prefix-suffix	$1.376 \cdot 10^9$	4	$3.794 \cdot 10^5$
delta encoding	$1.511 \cdot 10^9$	4	$3.160 \cdot 10^5$

(b) $N = 2^{48}$, $M_{\text{max}} = 64\text{GB}$

compression	m	r	T
none	$6.014 \cdot 10^9$	0	$1.243 \cdot 10^9$
prefix-suffix	$8.721 \cdot 10^9$	0	$5.913 \cdot 10^8$
delta encoding	$9.407 \cdot 10^9$	0	$5.082 \cdot 10^8$
none	$6.711 \cdot 10^9$	9	$1.027 \cdot 10^9$
prefix-suffix	$1.031 \cdot 10^{10}$	9	$4.420 \cdot 10^8$
delta encoding	$1.128 \cdot 10^{10}$	9	$3.703 \cdot 10^8$

(c) $N = 2^{56}$, $M_{\text{max}} = 1\text{TB}$

compression	m	r	T
none	$8.450 \cdot 10^{10}$	0	$4.127 \cdot 10^{11}$
prefix-suffix	$1.198 \cdot 10^{11}$	0	$2.053 \cdot 10^{11}$
delta encoding	$1.280 \cdot 10^{11}$	0	$1.799 \cdot 10^{11}$
none	$9.735 \cdot 10^{10}$	13	$3.191 \cdot 10^{11}$
prefix-suffix	$1.485 \cdot 10^{11}$	13	$1.389 \cdot 10^{11}$
delta encoding	$1.619 \cdot 10^{11}$	13	$1.173 \cdot 10^{11}$

with this algorithm, we can decrease the amount of storage required for the MPHF, while spending more time to compute it. The size of the MPHF in memory, and in particular the average size per key, influences the storage required per entry, and should be acknowledged in our analysis. We give in Figure 6.7 the evolution of the optimal online time T for each configuration in our fixed memory model with a varying MPHF description size. The online time is computed from the formula in Theorem 5.4.

The online time of the MPHF technique with the fixed memory model parameters ranges from $3.651 \cdot 10^8$ for an MPHF achieving the theoretical lower bound space complexity, to $3.756 \cdot 10^8$ for 2.08 bits per key in average, which approximately matches the parameters that were used for the introduction of the technique in Chapter 5. We also observe that the evolution of the online time is very close to linear with regard to the MPHF size. We can therefore approximate the online time by an affine function, whose parameters are derived from the values of Figure 6.7. This approximation is given in Equation (6.3). If we denote by s the average amount of bits per key required to store the MPHF, we can express the

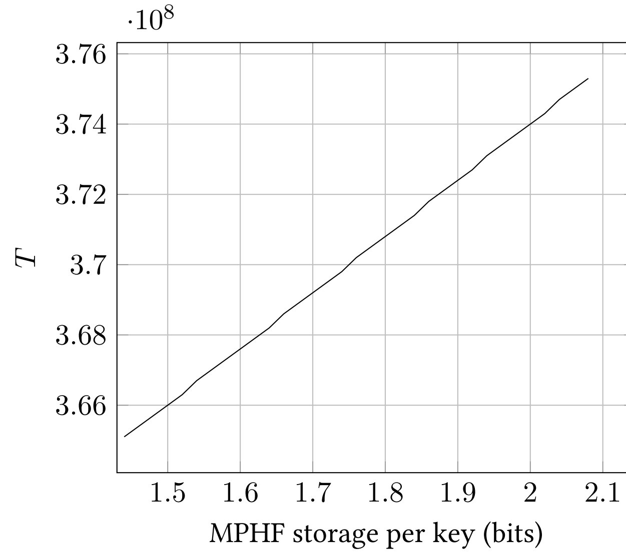


Figure 6.7 – Optimal MPHF online time for different MPHFs description sizes with the fixed memory parameters ($N = 2^{48}$, $M = 64$GB)

online time T with the parameters of the comparison by

$$T = as + b, \quad \text{where} \quad \begin{cases} a = 1.591 \cdot 10^7 \\ b = 3.422 \cdot 10^8 \end{cases} . \tag{6.3}$$

The MPHF description is a variable parameter which depends on a pre-computation trade-off in which one can trade more time in the MPHF construction to get a reduced storage for the MPHF. Equation (6.3) gives a more convenient way to compare with the other techniques without fixing this parameter.

6.4.5 Combinations with checkpoints

We now perform the optimisation of the checkpoint parameters on the partial combinations that were obtained with the storage technique and the MPHF technique. We iterate on both the truncation parameter, and the number of checkpoints. Since we will only deal with values of $t < 100000$, we compute optimal positions for the checkpoints for a configuration with $t_0 = 10000$, and the same matrix stopping constant of 1.92. The analysis in Section 6.3.4 ensured that the loss of precision that is incurred is negligible. We give the results in Table 6.8, where the optimal number of checkpoints for each combination is indicated by n, and the online time by T. We reproduce the results of the vanilla technique from Table 6.5, for reference. For the MPHF technique, the results depend on the description size of the MPHF, and in particular on the average amount of storage s required per key. We give the optimal configuration for three cases of interest. The first case ($s = 2.07$bits) is the configuration which was considered in the previous mentions of the MPHF, both in this section and in Section 5.5, which is known to require very reasonable construction time, and the only implemented practical MPHF so far. For the second case ($s = 1.75$bits), we use Equation (6.3) to obtain an online time corresponding to the online time of the combination of the delta encoding, the truncation technique, and the checkpoints. The third case ($s = 1.44$bits) is the theoretical lower bound for the description of the MPHF. We get that an MPHF requiring about 1.75 bits per key is equivalent to this combination.

Table 6.8 – Online time performance when including checkpoints in the fixed memory model ($M =$ 64GB, $N = 2^{48}$)

version	n	entry size[4]	m	t	T
vanilla	9	95 bits	$5.448 \cdot 10^9$	99207	$9.824 \cdot 10^8$
truncation + indexing	31	85 bits	$6.124 \cdot 10^9$	88245	$1.113 \cdot 10^8$
truncation + delta encoding	28	78 bits	$6.702 \cdot 10^9$	80643	$1.007 \cdot 10^8$
MPHF (s=2.07 bit)	34	84.07 bits	$6.009 \cdot 10^9$	89930	$1.015 \cdot 10^8$
MPHF (s=1.75 bit)	34	83.75 bits	$6.032 \cdot 10^9$	89598	$1.007 \cdot 10^8$
MPHF (s=1.44 bit)	33	82.44 bits	$6.125 \cdot 10^9$	88239	$9.995 \cdot 10^7$

The difference of the optimal online time is far less significant than in the previous comparisons of this section, with all the results being within a 10% difference from each other. However, the performance improvement of the combinations with the checkpoint technique is substantial compared to the one of storage-related combinations. The introduction of the checkpoints levels the performance of all the techniques whose online time come closer to around 10^8 in the context of our parameters.

An observation that can be made is that the number of checkpoint is very large. Recall that the optimal number of checkpoints in the vanilla trade-off was around 10, whereas the optimal values in Table 6.8 even exceed 30. This is a clear indicator of the impact of the checkpoint technique on the online time. With the classical techniques, it allows to compensate for the overhead incurred by the truncation technique, while benefiting from the fact that the endpoints are stored on very few bits, so the checkpoints' bit can take over the unoccupied space. With the MPHF technique, an interesting phenomenon arises. The optimal signature size is $b = 10$ in the three MPHF configurations. It is twice as much as with the optimal configuration without checkpoints. This means that the checkpoints can make up for the loss of 5 additional bits in terms of online time. These additional signature bits in turn reduce the number of false alarms induced by the lack of endpoints.

6.4.6 Recommendation

The efficiency results of the full combinations make it difficult to conclude about which technique performs the best. There are two challengers: the MPHF technique, and the delta encoding technique. The absolute minimum of online time is achieved by the MPHF technique in its lower space complexity. However, this complexity is theoretical in the sense that there is no guarantee that the construction algorithm for such an MPHF is actually feasible in practical time. A known practical setting of the MPHF construction, which yields MPHFs that are stored on 2.07 bits per keys, performs slightly worse than the delta encoding method. We already discussed in Section 5.5 that MPHFs might be constructed in such a way that their description takes less memory, but no analysis quantifies whether such an MPHF can be made in practical time. It turns out that the feasibility of the construction of such a reduced MPHF size can overturn the conclusion of whether it is the best technique or not. In particular, we determined that below a storage of about 1.75 bits per keys, the MPHF technique is the most efficient technique. In any case, the gain obtained by using the combination involving the theoretical lower bound for the MPHF is about 1.1%.

[4] The entry size is to be understood as exact for the classical trade-off techniques, since the index is negligible with regard to the rouding, and as an average in the case of the MPHF technique, where the signature is stored on an integral number of bits and description of the MPHF is viewed as “distributed” over the m elements in the table.

Therefore, we settled on what the practical setting mentioned above, and conclude that the MPHF technique, with the current state of the art, is not able to outperform the delta encoding technique. Then, the delta encoding method applied on truncated endpoint, and complemented with the checkpoint technique, is what should be recommended in order to achieve the best possible performance, in a high success rate setting for TMTOs. According to our analysis on the delta encoding technique above, these results hold for a wide range of problem sizes (see Figure 6.6), as long as the success rate remains high ($> 90\%$).

Practical case Let us present a practical case involving passwords. If we consider the passwords of 8 characters consisting of mixed case alphanumeric characters, plus two special symbols (*e.g.*, `-` and `_`), *i.e.*, those matched by the POSIX regular expression `[A-Za-z0-9_-]{8}`, we have a set of exactly 2^{48} elements. Let us assume we are interested in MD5 password digests. If we reuse the parameters above, we assume that a dedicated workstation, with at least 256GB of RAM is used to store the tables. On a recent processor, we can expect to perform about 10^7 MD5 calls per second in a single thread (no parallelism of the online phase is considered here). Then a dictionary attack corresponding to this problem would need $16 + 8 = 24$ bytes per row, which would amount to a 6PB memory requirement. A brute force attack would require in average 163 days. Using the vanilla version of the rainbow trade-off, a single inversion would take in average a little more than 2 minutes. With a combination of the truncation, the delta encoding and the checkpoint technique, which, as we saw above, has the best performance, the inversion time drops at 10 seconds in average.

6.5 Conclusion

Many analysis works in the literature on TMTOs perform comparisons between the different TMTO variants. However, such comparisons involve many challenges, and simplifications are prone to step into hidden ambushes. These difficulties were outlined in this chapter, which tackled the problem of determining the best possible TMTO in its best possible configuration.

We exposed the problem of the TMTO comparison, and described their difficulties. We then established assumptions based on real life considerations, such as the practical use of TMTO. These considerations, along with a review of the literature, allowed us to conclude on which TMTO variant to choose for our comparison. We settled on the use of the perfect rainbow trade-off, in the special setting of tables of maximal size, which we tweaked to consider tables of nearly maximal size, while remaining in a high success rate context.

We then built a model of comparison which allowed us to create a baseline on which all the improvements could be considered and compared easily, using the sole metric of the online phase. Our model overcomes the aforementioned difficulties, and improves upon the state of the art because it uses exact storage parameters instead of approximations. We presented arguments for the choices we made when reducing the dimension of parameters to this one metric. We then presented a taxonomy of the improvement techniques that can be applied to the rainbow trade-off, and highlighted which ones made sense in combinations.

We introduced the methodology for our comparison, which consists in a multi-layer optimisation. The first layer consists in the storage-related techniques, and the second consists in the checkpoint technique. The parameters of storage-related partial combinations are determined and tested against parameters of the other layer techniques.

An analysis of the checkpoint technique was made. In particular, the behaviour of the checkpoints' positions was studied, and its evolution for a high number of checkpoints was presented. It allowed us to make the optimisation of the parameters practical by carefully bounding the error induced by the inaccuracies of the position of the checkpoints.

The external memory model, as presented in Chapter 4, is not directly taken into account in our analysis, and left to future work. The MPHF rainbow scheme and the best configuration presented in Section 6.4 do not require the same access number in the table. In the MPHF scheme, only one access is required since the offset of the corresponding starting point is directly computed, whereas several accesses are required to get to the correct offset when using encoded deltas. This difference might influence the performance of the two trade-offs in particular situations, such as extremely high latency disks.

Finally, we established a set of parameters for our fixed memory model which was large, yet practical, for our comparison. We provided the online time of many improvements in our model, which allowed us to perform a clear comparison. We then introduced the combinations and provided average online time, with fixed memory, of all the previously introduced configurations. This allowed us to evaluate the relative impact of each improvement, and to establish the most efficient rainbow table configuration, using improvements from the state of the art.

In the process of the comparison, this chapter has exploited all the techniques studied in the previous chapters, and quantified their interest in terms of performance. Our work established a new baseline on which to build upon in order to exploit at best the versatility of the TMTOs, while being guaranteed to obtain the best performance.

References

[ABC15] Gildas Avoine, Adrien Bourgeois, and Xavier Carpent. *Analysis of Rainbow Tables with Fingerprints.* In: *ACISP 15.* Ed. by Ernest Foo and Douglas Stebila. Vol. 9144. LNCS. Brisbane, QLD, Australia: Springer, Heidelberg, Germany, June 2015, pp. 356–374 (cit. on pp. 73–75, 77, 79, 162, 163).

[AC14] Gildas Avoine and Xavier Carpent. *Optimal Storage for Rainbow Tables.* In: *ICISC 13.* Ed. by Hyang-Sook Lee and Dong-Guk Han. Vol. 8565. LNCS. Seoul, Korea: Springer, Heidelberg, Germany, Nov. 2014, pp. 144–157 (cit. on pp. 56–62, 161).

[AJO05] Gildas Avoine, Pascal Junod, and Philippe Oechslin. *Time-Memory Trade-Offs: False Alarm Detection Using Checkpoints.* In: *Progress in Cryptology - INDOCRYPT 2005.* Ed. by Subhamoy Maitra, C. E. Veni Madhavan, and Ramarathnam Venkatesan. Vol. 3797. LNCS. Bangalore, India: Springer, Heidelberg, Germany, Dec. 2005, pp. 183–196 (cit. on pp. 63, 64, 154).

[AJO08] Gildas Avoine, Pascal Junod, and Philippe Oechslin. "Characterization and Improvement of Time-Memory Trade-Off Based on Perfect Tables". In: *ACM Transactions on Information and System Security* 11.4 (4 July 2008), 17:1–17:22. ISSN: 1094-9224 (cit. on pp. 16, 27, 30, 37, 56, 64, 72–74, 77, 154, 156, 157, 160).

[BBS06] Elad Barkan, Eli Biham, and Adi Shamir. *Rigorous Bounds on Cryptanalytic Time/Memory Tradeoffs.* In: *CRYPTO 2006.* Ed. by Cynthia Dwork. Vol. 4117. LNCS. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 2006, pp. 1–21 (cit. on pp. 35, 40, 58, 153, 155–157).

[BCC57] R. Bellman, Rand Corporation, and Karreman Mathematics Research Collection. *Dynamic Programming*. Rand Corporation research study. Princeton University Press, 1957. ISBN: 9780691079516 (cit. on p. 166).

[Hel80] Martin Hellman. “A Cryptanalytic Time-Memory Trade Off”. In: *IEEE Transactions on Information Theory* IT-26.4 (July 1980), pp. 401–406 (cit. on pp. 24, 25, 29, 30, 42, 45, 55, 153, 154, 184).

[HM13] Jin Hong and Sunghwan Moon. “A Comparison of Cryptanalytic Tradeoff Algorithms”. In: *Journal of Cryptology* 26.4 (Oct. 2013), pp. 559–637 (cit. on pp. 16, 20, 30, 31, 33, 34, 44, 63, 157, 158, 160).

[Hon10] Jin Hong. “The cost of false alarms in Hellman and rainbow tradeoffs”. In: *Designs, Codes and Cryptography* 57.3 (Dec. 2010), pp. 293–327 (cit. on pp. 27, 29, 40, 63–65, 67, 69–72, 166).

[LH16] Ga Won Lee and Jin Hong. “Comparison of Perfect Table Cryptanalytic Tradeoff Algorithms”. In: *Designs, Codes and Cryptography* 80.3 (Sept. 2016), pp. 473–523 (cit. on pp. 30, 158, 160, 162).

[Oec03] Philippe Oechslin. *Making a Faster Cryptanalytic Time-Memory Trade-Off*. In: *CRYPTO 2003*. Ed. by Dan Boneh. Vol. 2729. LNCS. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 2003, pp. 617–630 (cit. on pp. 5, 35, 36, 56, 107, 153, 156, 157).

[TO05] Cedric Tissières and Philippe Oechslin. *Ophcrack*. Accessed: Apr 2017. 2005. Link. (Cit. on pp. 7, 35, 56, 93, 99, 162).

A conclusion is the place where you get tired of thinking.

Arthur Bloch

Conclusion 7

We wrap things up in this chapter, where we sum up the document, come back to our research question and review how the work we did can answer it. We then propose new avenues to pursue the work of this thesis, and more generally advance the TMTO field.

Contents

7.1 Summary **184**
7.2 Results **185**
7.3 Open problems **186**
7.3.1 TMTOs on GPUs 187
7.3.2 Distributed pre-computation 187

7.1 Summary

The work presented in this thesis is a study on cryptanalytic time-memory trade-offs based on the original concept introduced by Hellman [Hel80]. These are techniques that provide so called online algorithms to allow retrieving preimages from a one-way function values. These work on a defined subset of its input space, and allow retrieval in less time than an exhaustive search, and are described in less memory than what a dictionary storing every (preimage, image) pair would require. The algorithm relies on chains of elements linked with the application of one-way function. From a set of starting points, the one-way function is iterated to obtain as much endpoints. These are stored in tables which are part of the online algorithm retrieving the preimage. The technique is parameterised in such a way that one can choose to use more memory in order to reduce the running time of the algorithm, or inversely to spend more time to find the preimage while using less memory. However, building the algorithm's tables is costly in time, often requiring much more computations than in the case of a single exhaustive search. This restrains its use to cases where the problem is known ahead of time, so that a pre-computation can be performed. The description of these running time algorithms includes large amount of data, often called tables, which constitutes the memory of the trade-off. The TMTO technique was formally exposed in Chapter 2, where we presented all the TMTO variants introduced in the literature until now. While the description given in this thesis is problem-independent, our discussion is oriented towards large spaces for which we believe TMTOs are especially useful. Also, TMTOs were presented in the context of password cracking. Indeed, recovering as plaintext password given its digest by a hash function was shown to be a common usage of TMTOs.

Many improvements of TMTOs were proposed over the years in the scientific literature, some of them significantly reducing either the time or memory of the online algorithm. A comprehensive review of these improvements was performed in Chapter 3, which proposed their taxonomy. The space improvements tackle the compression of the tables, with techniques such as the indexing, or delta encoding which compresses the endpoints, or the truncation of the endpoint. They provide, in isolation, the most significant improvements of the TMTOs performance. A second category, including checkpoints and fingerprints, aimed at reducing the computational cost of false alarms, which are the TMTO algorithms' false positives. We saw with the interleaving technique how to obtain the average online performance of two TMTOs. The technique applies when two different subsets of different size from the same problem space happen to have different probability of success in practice, *i.e.*, the preimages are more likely to be found in the smaller subset. In general, a TMTO algorithm make use of several tables of the same size, to increase the probability of success. A similar idea as the one of the interleaving technique was applied on a single TMTO with heterogeneous tables, *i.e.*of different chain length, and provided the average online time in this case. Finally, we gave an overview of the implementation of the TMTO algorithm using GPUs, and we mentioned the underlying challenges.

In Chapter 4, we extended TMTOs to account for an external memory. While most of the literature assumed that the tables fit in the RAM, allowing them to discard the time spent in fetching the data from the tables, such an assumption does not hold in a large setting such as studied in this thesis. Indeed, for large problem spaces, the implementations are designed in a way that the tables of the TMTO algorithm are stored on a disk instead. We provided an analysis of the classical algorithm, such as the one studied in the literature in the RAM model, in the context of the external memory model. We studied the behaviour of the algorithm both on classical HDDs and on newer SSDs, which are replacing them nowadays as efficient storage drives. We compared our results for the classical algorithms with the most often implemented algorithm [KHP13]. We showed that the latter was, in fact, not adapted for

all situations, and that the classical algorithm outperformed it in the case of a problem with larger space on SSD.

A novel kind of TMTO algorithms was presented in Chapter 5. Contrarily to the previous methods which relied on endpoints to identify the chains, this new method used minimal perfect hash functions. Therefore, its description does not need to store the endpoints. However, in order to reduce the false positives, a signature of a few bits is stored alongside the starting points. An analysis of the MPHF-based technique was given, and it was shown to be comparable to the truncation of the endpoints in terms of information stored in the tables. However, our technique requires much less storage for an equivalent online time. An experimental validation was also performed on a problem of moderate size. Our technique introduced another kind of trade-off. The construction of the MPHF on which our method is built being itself tunable, one can reduce the memory footprint of the tables while spending more time on their construction.

Finally, in Chapter 6, we built upon the previous results to wrap them up into the most efficient TMTO that can be constructed by exploiting the existing improvements. We performed a global analysis and comparison of the TMTOs with various improvements combinations, extending the existing work in the literature, where the improvements were often analysed in isolation. We introduced a new model of comparison with a fixed memory. We reduced the problem to the only dimension of time, in such a way that all the improvements were easily comparable. We also provided a more fine-grained evaluation of the memory footprint of each algorithm, in such a way that we could acknowledge for minor implementation-related subtleties. We showed that the best combinations' results were quite close to each other, and that the best configuration was dependent on the choices made in the trade-off introduced by our MPHF technique. Indeed, the MPHF-based technique allows, given a large enough pre-computation time, to outperform the combination of the checkpoints and the delta encoding technique.

7.2 Results

In our work, we aimed at improving TMTOs' space efficiency. Since the memory in a TMTO is one of many interdependent parameters, we had to first overview the technique in detail, in order to highlight its versatility, and the subtleties involved in TMTO optimisations. We saw that the memory was indeed a scarce resource when one aims at constructing TMTOs with a probability of success close to 100%. This is why, in our various contributions, we focused on the rainbow trade-off, which is best suited for high success rate. This choice was further justified when we presented a performance comparison between the rainbow trade-off and the two main other variants of the TMTOs: the DP trade-off, and the Hellman trade-off.

Our treatment of the research problem involved two approaches. Our first approach, which was to consider additional memory, was not strictly an improvement of the space efficiency itself, but rather a way to make it cheaper to increase the memory parameter. The use of hard-drive to store the TMTO tables is not new in the implementation landscape. Using disks allows to construct TMTOs that exceed the maximum amount that a single computer can hold, so there is no possible experimental comparison with a RAM version. Moreover, no analysis has been made to quantify how it actually improved the efficiency of the TMTO, even for small problems' size, compared to the RAM version. We performed such an analysis, and showed that assumptions that date back to slow hard drives do not always hold for more modern disks such as SSDs. In particular, the same algorithm that was used in RAM is valuable when using modern disks, and there is no need in that case to preload tables in RAM, such as done in

implementations. This means that, in the end, we can compute a TMTO on large quantities of disk that is slightly slower, but provide faster online time than a TMTO using the maximum amount of RAM.

Our second approach was to seek memory optimisations on an already pre-computed TMTO. We reused existing techniques involved in lookup data structures such as hash tables. This allowed us to reduce the memory footprint of the TMTO technique significantly, since storing the endpoints is no longer necessary. However, this comes at the expense of additional false alarms which have to be mitigated by an endpoint signature. The memory required to store a row in a table, even when accounting for the space required by the signature, is lower than for any improvement proposed so far.

We pursued our second approach by proposing a combination of promising improvements, when until now, they were only introduced and compared in isolation. Combined memory improvements happen to provide a significant gain, compared each technique alone. In addition to memory optimisation, we included checkpoints in our combinations. While these are not memory improvements, they are the most significant improvements on TMTOs since the introduction of the rainbow trade-off, and should be considered in any serious implementation. Besides, a reduction in average online time can always be converted in a lower memory requirement if the parameters are balanced to compensate for said reduction. We observed that the MPHF technique is in fact comparable in efficiency with the combination of the endpoint truncation and the delta encoding of the endpoint. Considering the full combinations, *i.e.*, the above ones including in addition the checkpoint technique, did not change this fact, as all of them are very close to each other in terms of performance.

In conclusion, we stated that the combination of delta encoding and the truncation technique was the most efficient memory optimisation, with a slightly better performance than the MPHF technique. However, the MPHF technique, achieving an efficiency similar to this best combination, while relying on very different assumptions regarding the endpoint storage, might lead to new discoveries on TMTOs improvements. These results are also prone to change, should a better MPHF construction algorithm, closer to the theoretical space efficiency lower bound than what is proposed today, appear in the literature.

7.3 Open problems

We focused mainly on the online performance of the TMTO algorithms. As a result, the pre-computation aspect was not studied in depth. The pre-computation phase is one of the main areas left to explore for TMTOs. It is indeed hardly discussed in the literature, and when it is, the details are left out and a naive approach is used. This section describes the problem that we believe are susceptible to yield significant gains on the TMTO method.

During the pre-computation phase, TMTO matrices are computed from a set of m_0 starting points. This amounts to $m_0 t\ell$ computations of the one-way function, where t is the length of the chains, and ℓ the number of tables. In the non-perfect version of the trade-off, the (starting point, endpoint) pairs of all of the m_0 chains are kept. However, when one considers a perfect trade-off, only the chains ending in distinct endpoints are considered. This often results in practice in $m \ll m_0$ chains that are kept. Therefore, $m_0 - m$ discarded chains are not directly used in the final table.

Two chains with identical endpoints are due to one of the chains merging into another. This can happen in any of the columns. An improvement over the naive approach consists in checking for collisions before the end of the chain, so that the elements after the collision of one of the two chains do not have to be computed anymore. Unfortunately, doing so in the Hellman trade-off, or in the DP trade-off, would be equivalent to constructing a perfect Hellman table, which we know is not practical.

The case of the rainbow trade-off is different though, in that it suffices to only identify if there is a duplicate in the same column in order to determine a merge with a given chain up to this column. Then we can consider a scheme where at each column, the duplicate elements are discarded, hence reducing the cost of the pre-computation. It is clear that the benefit of such a method is maximal when a table of maximal size, with a corresponding TMTO matrix of size Nt, is considered.

The pre-computation can be substantially reduced when checking for duplicates before the end of the chain. For the case when the pre-computation is done on a single computer, with the TMTO matrix computed in RAM, checking for duplicates is fast. However, similarly to what was observed in Chapter 4 with the fetching times, when the pre-computation involves a more complex architecture than a single RAM, the cost of checking for duplicates can no longer be neglected. In practice, in order to perform the pre-computation, implementations make use of distributed systems, such as computer networks, and leverage GPUs to perform this highly parallel work. These constitute two of the main open problems in the study of TMTOs, which are detailed hereafter.

7.3.1 TMTOs on GPUs

We saw, in Section 3.7, an evaluation of a GPU-assisted online phase of the rainbow trade-off. We made a brief overview of the GPU architecture, and how it impacts the online phase. Several works, such as [KSH+12], focus on a given implementation which is then analysed. However, no formal analysis of the usage of GPUs in TMTOs has been made. Such an analysis would provide a framework balancing time and memory of the TMTO with regard to the constraints of the GPU architecture.

Beyond the consideration of the online phase, as with general pre-computations, which is not tackled by the TMTO literature, the question of how to handle the pre-computation with a GPU is yet to be treated. In particular, in the context of the perfect trade-off, with the pre-computation optimisations that were mentioned above, the GPU architecture makes the computation of the TMTO matrix a non-trivial task. For example, the checking of duplicates in intermediate columns is a complicated matter in the GPU, where shared memory is expensive in access time. Moreover, conditional branching on values from other chains is a difficult task in this architecture, that may impair the parallel capabilities of the GPU, which may not yield, in this context, an optimal performance.

A future work could establish an analysis in the like of the one of Chapter 4 applied to TMTOs on GPUs which would account for these architecture peculiarities, by quantifying the impact of separating the memory which contains the table, and the processing units performing the computation of the one-way functions.

7.3.2 Distributed pre-computation

The pre-computation consists in computing m_0 independent chains. This is a trivially parallel problem, which explains why known implementations of the pre-computation (see [Shu09]) are compatible with distributed computing. The distributed configuration is mentioned in [Sie13], which provides a summed-up explanation of the problem. However, an analysis of the problem, and the determination of the practical speed-up that can be achieved with a pre-computation optimisation, on a distributed system, remains to be done.

Distributed systems always bear a great complexity because they rely on a network which consists in several technological stacks. Performance can be altered by the protocol used, and the hardware connecting the nodes. In particular, congestions can happen, and they can make the prediction difficult, due to their chaotic nature. Therefore, the optimisation involving a duplicate check at each column

would be detrimental to the performance due to the fact that processing each column would require much more time than a single computation of the one-way function, and it would not be worth optimising as such compared to the naive approach. The challenge then becomes to assess the right amount of checks that have to be performed during the pre-computations in order to reduce the time, while keeping the check overhead moderate enough that it does not exceed the time gained.

When considering the distribution of the pre-computation, two different models, which represent two configurations of the computers performing the pre-computation, can be established:

- a *star model*, where a powerful master node performs the filtering, while delegating the computation of the chains to many slaves nodes, and
- a *peer-to-peer model* where all the nodes share the same function.

These are two of the most common configurations that can be found in distributed systems. On the one hand, the star model provides an easy scheduling of the pre-computations, but requires a great amount of memory in order to perform the sorting required to eliminate chains with duplicate endpoints. On the other hand, the peer-to-peer model performs the sorting in a distributed manner which lowers the time required to check for the duplicates. However, since there is no centralised node in this model, all the scheduling must be done in a peer-to-peer fashion, which implies some overhead.

The latency and the bandwidth of the configuration greatly depend on the hardware characteristics of the network. These parameters influence the behaviour of the configurations in terms of the number of nodes, in such a way that the performance of each of the models is in fact tied to practical considerations. Which of the two models provides the best performance in the pre-computation phase for practical settings remains to be determined.

References

[Hel80] Martin Hellman. "A Cryptanalytic Time-Memory Trade Off". In: *IEEE Transactions on Information Theory* IT-26.4 (July 1980), pp. 401–406 (cit. on pp. 24, 25, 29, 30, 42, 45, 55, 153, 154, 184).

[KHP13] Jung Woo Kim, Jin Hong, and Kunsoo Park. *Analysis of the Rainbow Tradeoff Algorithm Used in Practice*. Cryptology ePrint Archive, Report 2013/591. http://eprint.iacr.org/2013/591. 2013 (cit. on pp. 97–101, 103–106, 109–111, 116, 117, 121, 184).

[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: *Progress in Cryptology - INDOCRYPT 2012*. Ed. by Steven D. Galbraith and Mridul Nandi. Vol. 7668. LNCS. Kolkata, India: Springer, Heidelberg, Germany, Dec. 2012, pp. 303–316 (cit. on pp. 70, 84, 85, 187).

[Shu09] Zhu Shuanglei. *RainbowCrack*. Accessed: Apr 2017. 2009. Link. (Cit. on pp. 7, 93, 187).

[Sie13] Matthieu Sieben. *Accelerating the computation of rainbow tables for cryptanalytic Time-memory Trade-offs*. MA thesis. Louvain-la-Neuve: Ecole polytechnique de Louvain, 2013 (cit. on p. 187).

Bibliography

[AAA+02] N. R. Adiga et al. *An Overview of the BlueGene/L Supercomputer.* In: *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. Nov. 2002, pp. 60–60 (cit. on p. 93).

[AAC+17] Hamza Abusalah, Joël Alwen, Bram Cohen, Danylo Khilko, Krzysztof Pietrzak, and Leonid Reyzin. *Beyond Hellman's Time-Memory Trade-Offs with Applications to Proofs of Space*. In: *Advances in Cryptology – ASIACRYPT 2017*. Ed. by Tsuyoshi Takagi and Thomas Peyrin. Vol. 10625. LCNS. Cham: Springer International Publishing, 2017, pp. 357–379. ISBN: 978-3-319-70697-9 (cit. on p. 5).

[ABC15] Gildas Avoine, Adrien Bourgeois, and Xavier Carpent. *Analysis of Rainbow Tables with Fingerprints.* In: *ACISP 15*. Ed. by Ernest Foo and Douglas Stebila. Vol. 9144. LNCS. Brisbane, QLD, Australia: Springer, Heidelberg, Germany, June 2015, pp. 356–374 (cit. on pp. 73–75, 77, 79, 162, 163).

[AC14] Gildas Avoine and Xavier Carpent. *Optimal Storage for Rainbow Tables.* In: *ICISC 13*. Ed. by Hyang-Sook Lee and Dong-Guk Han. Vol. 8565. LNCS. Seoul, Korea: Springer, Heidelberg, Germany, Nov. 2014, pp. 144–157 (cit. on pp. 56–62, 161).

[AC17] Gildas Avoine and Xavier Carpent. *Heterogeneous Rainbow Table Widths Provide Faster Cryptanalyses.* In: *ASIACCS 17*. Ed. by Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi. Abu Dhabi, United Arab Emirates: ACM Press, Apr. 2017, pp. 815–822 (cit. on pp. 20, 81–83).

[ACFT19] Gildas Avoine, Xavier Carpent, Barbara Fila, and Florent Tardif. *Cryptanalytic Time-Memory Trade-Offs: Recommended Configurations.* unpublished. 2019 (cit. on pp. xiii, 9, 126).

[ACK+02] David P Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. "SETI@ home: an experiment in public-resource computing". In: *Communications of the ACM* 45.11 (2002), pp. 56–61 (cit. on p. 7).

[ACKT17] Gildas Avoine, Xavier Carpent, Barbara Kordy, and Florent Tardif. *How to Handle Rainbow Tables with External Memory.* In: *ACISP 17, Part I*. Ed. by Josef Pieprzyk and Suriadi Suriadi. Vol. 10342. LNCS. Auckland, New Zealand: springer, July 2017, pp. 306–323 (cit. on pp. xiii, 9).

[ACL15] Gildas Avoine, Xavier Carpent, and Cédric Lauradoux. *Interleaving Cryptanalytic Time-Memory Trade-Offs on Non-uniform Distributions.* In: *ESORICS 2015 , Part I*. Vol. 9326. LNCS. Vienna, Austria: Springer, Heidelberg, Germany, Sept. 2015, pp. 165–184 (cit. on pp. 20, 78–82).

[AJO05] Gildas Avoine, Pascal Junod, and Philippe Oechslin. *Time-Memory Trade-Offs: False Alarm Detection Using Checkpoints*. In: *Progress in Cryptology - INDOCRYPT 2005*. Ed. by Subhamoy Maitra, C. E. Veni Madhavan, and Ramarathnam Venkatesan. Vol. 3797. LNCS. Bangalore, India: Springer, Heidelberg, Germany, Dec. 2005, pp. 183–196 (cit. on pp. 63, 64, 154).

[AJO08] Gildas Avoine, Pascal Junod, and Philippe Oechslin. "Characterization and Improvement of Time-Memory Trade-Off Based on Perfect Tables". In: *ACM Transactions on Information and System Security* 11.4 (4 July 2008), 17:1–17:22. ISSN: 1094-9224 (cit. on pp. 16, 27, 30, 37, 56, 64, 72–74, 77, 154, 156, 157, 160).

[Aut06] Unknown Author. *Free Rainbow Table*. Accessed: Jan 2019. 2006. Link. (Cit. on pp. 98, 105, 107).

[AV88] Alok Aggarwal and Jeffrey Scott Vitter. "The Input/Output Complexity of Sorting and Related Problems". In: *Communications of the ACM* 31.9 (1988), pp. 1116–1127 (cit. on p. 96).

[Bab95] Steve Babbage. *A space/time tradeoff in exhaustive search attacks on stream ciphers*. In: *European Convention on Security and Detection – ECOS'95*. Vol. 408. Conference publication (Institution of Electrical Engineers). Institution of Electrical Engineers. Brighton, England: Inspec/IEE, May 1995 (cit. on p. 42).

[BBD09] Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. *Hash, Displace, and Compress*. In: *Algorithms - ESA 2009*. Ed. by Amos Fiat and Peter Sanders. Vol. 5757. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 682–693. ISBN: 978-3-642-04128-0 (cit. on pp. 134, 137, 141, 142, 144, 146).

[BBS06] Elad Barkan, Eli Biham, and Adi Shamir. *Rigorous Bounds on Cryptanalytic Time/Memory Tradeoffs*. In: *CRYPTO 2006*. Ed. by Cynthia Dwork. Vol. 4117. LNCS. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 2006, pp. 1–21 (cit. on pp. 35, 40, 58, 153, 155–157).

[BCC57] R. Bellman, Rand Corporation, and Karreman Mathematics Research Collection. *Dynamic Programming*. Rand Corporation research study. Princeton University Press, 1957. ISBN: 9780691079516 (cit. on p. 166).

[BD00] Eli Biham and Orr Dunkelman. *Cryptanalysis of the A5/1 GSM Stream Cipher*. In: *Progress in Cryptology - INDOCRYPT 2000*. Ed. by Bimal K. Roy and Eiji Okamoto. Vol. 1977. LNCS. Calcutta, India: Springer, Heidelberg, Germany, Dec. 2000, pp. 43–51 (cit. on p. 42).

[BGB17] Matias Bjørling, Javier González, and Philippe Bonnet. *LightNVM: The Linux Open-Channel {SSD} Subsystem*. In: *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*. 2017, pp. 359–374 (cit. on p. 107).

[BGS+05] Steve Bono, Matthew Green, Adam Stubblefield, Ari Juels, Avi Rubin, and Michael Szydlo. *Security Analysis of a Cryptographically-Enabled RFID Device*. In: *In 14th USENIX Security Symposium*. USENIX, 2005, pp. 1–16 (cit. on p. 5).

[BMS06] Alex Biryukov, Sourav Mukhopadhyay, and Palash Sarkar. *Improved Time-Memory Trade-Offs with Multiple Data*. In: *SAC 2005*. Ed. by Bart Preneel and Stafford Tavares. Vol. 3897. LNCS. Kingston, Ontario, Canada: Springer, Heidelberg, Germany, Aug. 2006, pp. 110–127 (cit. on p. 42).

[BP13] Fabian van den Broek and Erik Poll. *A Comparison of Time-Memory Trade-Off Attacks on Stream Ciphers.* In: *AFRICACRYPT 13.* Ed. by Amr Youssef, Abderrahmane Nitaj, and Aboul Ella Hassanien. Vol. 7918. LNCS. Cairo, Egypt: Springer, Heidelberg, Germany, June 2013, pp. 406–423 (cit. on p. 40).

[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.* Ed. by Peter de With and Mihaela van der Schaar-Mitrea. Veldhoven, The Netherlands, May 1998, pp. 111–118 (cit. on pp. 30–32).

[BPZ07] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. *Simple and Space-Efficient Minimal Perfect Hash Functions.* In: *Algorithms and Data Structures, 10th International Workshop, WADS 2007.* Ed. by Frank Dehne, Jörg-Rüdiger Sack, and Norbert Zeh. Vol. 4619. Lecture Notes in Computer Science. Halifax, Canada: Springer Berlin Heidelberg, 2007, pp. 139–150. ISBN: 978-3-540-73951-7 (cit. on pp. 134, 137).

[BS00] Alex Biryukov and Adi Shamir. *Cryptanalytic Time/Memory/Data Tradeoffs for Stream Ciphers.* In: *ASIACRYPT 2000.* Ed. by Tatsuaki Okamoto. Vol. 1976. LNCS. Kyoto, Japan: Springer, Heidelberg, Germany, Dec. 2000, pp. 1–13 (cit. on pp. 22, 42–44).

[BSW00] Alex Biryukov, Adi Shamir, and David Wagner. *Real Time Cryptanalysis of A5/1 on a PC.* In: *Fast Software Encryption – FSE'00.* Ed. by Bruce Schneier. Vol. 1978. Lecture Notes in Computer Science. New York, USA: Springer, Apr. 2000, pp. 1–18 (cit. on pp. 5, 56, 62, 63).

[BZ07] Fabiano C. Botelho and Nivio Ziviani. *External Perfect Hashing for Very Large Key Sets.* In: *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management.* CIKM '07. Lisbon, Portugal: ACM, 2007, pp. 653–662. ISBN: 978-1-59593-803-9 (cit. on p. 137).

[CBBZ12] Davi de Castro Reis, Djamel Belazzougui, Fabiano Cupertino Botelho, and Nivio Ziviani. *CMPH library.* Accessed: May 2019. 2012. Link. (Cit. on p. 144).

[CH12] Danny Cobb and Amber Huffman. *NVM express and the PCI express SSD revolution.* Intel, 2012 (cit. on p. 106).

[Cha84] C. C. Chang. "The Study of an Ordered Minimal Perfect Hashing Scheme". In: *Commun. ACM* 27.4 (Apr. 1984), pp. 384–387. ISSN: 0001-0782 (cit. on p. 131).

[Cic80] Richard J. Cichelli. "Minimal Perfect Hash Functions Made Simple". In: *Commun. ACM* 23.1 (Jan. 1980). Ed. by M. Douglas McIlroy, pp. 17–19. ISSN: 0001-0782 (cit. on pp. 130, 132).

[CK08] Guanhan Chew and Khoongming Khoo. *A General Framework for Guess-and-Determine and Time-Memory-Data Trade-Off Attacks on Stream Ciphers.* In: *SECRYPT.* INSTICC Press, 2008, pp. 300–305 (cit. on p. 42).

[Cor11] Intel Corporation. *Intel® 64 and ia-32 architectures software developer's manual.* 2011 (cit. on p. 106).

[Cor13] Intel Corporation. *Intel®SHA Extensions.* Accessed: Jan 2019. 2013. Link. (Cit. on p. 95).

[CR72] Stephen A. Cook and Robert A. Reckhow. *Time-Bounded Random Access Machines.* In: *STOC.* ACM, 1972, pp. 73–80 (cit. on p. 94).

[Cub09] Nik Cubrilovic. *RockYou Hack: From Bad To Worse.* Accessed: Apr 2017. 2009. Link. (Cit. on p. 78).

[CW77] J. Lawrence Carter and Mark N. Wegman. *Universal Classes of Hash Functions (Extended Abstract)*. In: *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*. STOC '77. Boulder, Colorado, USA: ACM, 1977, pp. 106–112 (cit. on pp. 130, 132).

[CW79] J. Lawrence Carter and Mark N. Wegman. "Universal Classes of Hash Functions". In: *J. Comput. Syst. Sci.* 18.2 (1979), pp. 143–154 (cit. on p. 130).

[Den82] Dorothy E. Denning. *Cryptography and Data Security*. Boston, Massachusetts, USA: Addison-Wesley, June 1982, p. 100 (cit. on p. 30).

[DFKP15] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. *Proofs of Space*. In: *Advances in Cryptology – CRYPTO 2015*. Ed. by Rosario Gennaro and Matthew Robshaw. Vol. 9216. LCNS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 585–605. ISBN: 978-3-662-48000-7 (cit. on p. 5).

[DH01] Martin Dietzfelbinger and Torben Hagerup. *Simple Minimal Perfect Hashing in Less Space*. In: *Algorithms - ESA 2001*. Ed. by Friedhelm Meyer auf der Heide. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 109–120. ISBN: 978-3-540-44676-7 (cit. on p. 134).

[DK08a] Orr Dunkelman and Nathan Keller. *A New Attack on the LEX Stream Cipher*. In: *Advances in Cryptology - ASIACRYPT 2008*. Ed. by Josef Pieprzyk. Vol. 5350. LCNS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 539–556. ISBN: 978-3-540-89255-7 (cit. on p. 5).

[DK08b] Orr Dunkelman and Nathan Keller. "Treatment of the initial value in Time-Memory-Data Tradeoff attacks on stream ciphers". In: *Information Processing Letters* 107.5 (Aug. 2008). Ed. by Andrzej Tarlecki, pp. 133–137 (cit. on p. 42).

[Dum56] Arnold I. Dumey. "Indexing for rapid random access memory systems". In: *Computers and Automation* 5.12 (Dec. 1956), pp. 6–9 (cit. on p. 128).

[ER64] Calvin C. Elgot and Abraham Robinson. "Random-Access Stored-Program Machines, an Approach to Programming Languages". In: *Journal of the ACM* 11.4 (1964), pp. 365–399 (cit. on p. 94).

[FHCD92] Edward A. Fox, Lenwood S. Heath, Qi Fan Chen, and Amjad M. Daoud. "Practical Minimal Perfect Hash Functions for Large Databases". In: *Commun. ACM* 35.1 (Jan. 1992), pp. 105–121. ISSN: 0001-0782 (cit. on pp. 130, 132).

[FKS82] Michael L. Fredman, János Komlós, and Endre Szemerédi. *Storing a Sparse Table with $O(1)$ Worst Case Access Time*. In: *23rd FOCS*. Chicago, Illinois: IEEE Computer Society Press, Nov. 1982, pp. 165–169 (cit. on p. 133).

[FN91] Amos Fiat and Moni Naor. *Rigorous Time/Space Tradeoffs for Inverting Functions*. In: *23rd ACM STOC*. New Orleans, LA, USA: ACM Press, May 1991, pp. 534–541 (cit. on p. 15).

[GKP89] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete mathematics: a foundation for computer science*. In: vol. 3. 5. AIP, 1989. Chap. 5, pp. 175–178 (cit. on p. 59).

[Gol66] Solomon Golomb. "Run-length encodings (Corresp.)" In: *IEEE Transactions on Information Theory* 12.3 (1966), pp. 399–401 (cit. on p. 59).

[Gol97] Jovan Dj. Golic. *Cryptanalysis of Alleged A5 Stream Cipher*. In: ed. by Walter Fumy. Vol. 1233. LNCS. Konstanz, Germany: Springer, Heidelberg, Germany, May 1997, pp. 239–255 (cit. on p. 42).

[GOV16] Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. *Fast Scalable Construction of (Minimal Perfect Hash) Functions*. In: *Experimental Algorithms - 15th International Symposium, SEA 2016*. Vol. 9685. Lecture Notes in Computer Science. L'Aquila, Italy: Springer International Publishing, June 2016, pp. 339–352. ISBN: 978-3-95977-070-5 (cit. on p. 134).

[GT63] Marek Greniewski and Wladyslaw Turski. "The External Language KLIPA for the URAL-2 Digital Computer". In: *Commun. ACM* 6.6 (June 1963), pp. 321–324. ISSN: 0001-0782 (cit. on p. 129).

[GV75] Robert G. Gallager and David C. van Voorhis. "Optimal source codes for geometrically distributed integer alphabets (Corresp.)" In: *IEEE Transactions on Information Theory* 21.2 (Mar. 1975), pp. 228–230. ISSN: 0018-9448 (cit. on p. 60).

[Hel67] Herbert Hellerman. *Digital computer system principles*. In: 1st ed. New York: McGraw-Hill, 1967. Chap. 2, p. 152 (cit. on p. 128).

[Hel80] Martin Hellman. "A Cryptanalytic Time-Memory Trade Off". In: *IEEE Transactions on Information Theory* IT-26.4 (July 1980), pp. 401–406 (cit. on pp. 24, 25, 29, 30, 42, 45, 55, 153, 154, 184).

[HJK+08] Jin Hong, Kyung Jeong, Eun Kwon, In-Sok Lee, and Daegun Ma. *Variants of the Distinguished Point Method for Cryptanalytic Time Memory Trade-Offs*. In: *Information Security Practice and Experience*. Ed. by Liqun Chen, Yi Mu, and Willy Susilo. Vol. 4991. Lecture Notes in Computer Science. Sydney, Australia: Springer, Apr. 2008, pp. 131–145 (cit. on p. 30).

[HJMM08] Martin Hell, Thomas Johansson, Alexander Maximov, and Willi Meier. *The Grain Family of Stream Ciphers*. In: *New Stream Cipher Designs: The eSTREAM Finalists*. Ed. by Matthew Robshaw and Olivier Billet. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 179–190. ISBN: 978-3-540-68351-3 (cit. on p. 5).

[HK05] Jin Hong and Woo-Hwan Kim. *TMD-Tradeoff and State Entropy Loss Considerations of Streamcipher MICKEY*. In: *Progress in Cryptology - INDOCRYPT 2005*. Ed. by Subhamoy Maitra, C. E. Veni Madhavan, and Ramarathnam Venkatesan. Vol. 3797. LCNS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 169–182. ISBN: 978-3-540-32278-8 (cit. on p. 5).

[HK86] Gary Haggard and Kevin Karplus. *Finding Minimal Perfect Hash Functions*. In: *Proceedings of the Seventeenth SIGCSE Technical Symposium on Computer Science Education*. Ed. by Joyce Currie Little and Lillian N. Cassel. SIGCSE '86. Cincinnati, Ohio, USA: ACM, 1986, pp. 191–193. ISBN: 0-89791-178-4 (cit. on p. 132).

[HKR83] Martin E. Hellman, Ehud D. Karnin, and Justin Reyneri. "On the Necessity of Cryptanalytic Exhaustive Search". In: *SIGACT News* 15.1 (Jan. 1983), pp. 40–44. ISSN: 0163-5700 (cit. on p. 18).

[HLM11] Jin Hong, Ga Won Lee, and Daegun Ma. *Analysis of the Parallel Distinguished Point Tradeoff*. In: *Progress in Cryptology - INDOCRYPT 2011*. Ed. by Daniel J. Bernstein and Sanjit Chatterjee. Vol. 7107. LNCS. Chennai, India: Springer, Heidelberg, Germany, Dec. 2011, pp. 161–180 (cit. on p. 30).

[HM13] Jin Hong and Sunghwan Moon. "A Comparison of Cryptanalytic Tradeoff Algorithms". In: *Journal of Cryptology* 26.4 (Oct. 2013), pp. 559–637 (cit. on pp. 16, 20, 30, 31, 33, 34, 44, 63, 157, 158, 160).

[Hoc09] Yaacov Zvi Hoch. *Security Analysis of Generic Iterated Hash Functions*. PhD thesis. Rehovot, Israel: Weizmann Institute of Science, Aug. 2009 (cit. on p. 78).

[Hon10] Jin Hong. "The cost of false alarms in Hellman and rainbow tradeoffs". In: *Designs, Codes and Cryptography* 57.3 (Dec. 2010), pp. 293–327 (cit. on pp. 27, 29, 40, 63–65, 67, 69–72, 166).

[Hon16] Jin Hong. "Perfect Rainbow Tradeoff with Checkpoints Revisited". In: *PLOS ONE* 11.11 (Nov. 2016), pp. 1–18 (cit. on pp. 64, 70, 73).

[Hun13] Troy Hunt. *Have I Been Pwned*. https://haveibeenpwned.com. Accessed: 2019-01. 2013. Link. (Cit. on p. 6).

[Jae81] G. Jaeschke. "Reciprocal Hashing: A Method for Generating Minimal Perfect Hashing Functions". In: *Commun. ACM* 24.12 (Dec. 1981), pp. 829–833. ISSN: 0001-0782 (cit. on p. 131).

[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". In: *IEICE Transactions* 92-A.1 (2009), pp. 11–21 (cit. on p. 42).

[KGL06] Khoongming Khoo, Guang Gong, and Hian-Kiat Lee. *The Rainbow Attack on Stream Ciphers Based on Maiorana-McFarland Functions*. In: *Applied Cryptography and Network Security*. Ed. by Jianying Zhou, Moti Yung, and Feng Bao. Vol. 3989. LCNS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 194–209. ISBN: 978-3-540-34704-0 (cit. on p. 5).

[KH13] Byoung-Il Kim and Jin Hong. *Analysis of the Non-perfect Table Fuzzy Rainbow Tradeoff*. In: *ACISP 13*. Ed. by Colin Boyd and Leonie Simpson. Vol. 7959. LNCS. Brisbane, Australia: Springer, Heidelberg, Germany, July 2013, pp. 347–362 (cit. on pp. 41, 42).

[KH14] Byoung-Il Kim and Jin Hong. "Analysis of the Perfect Table Fuzzy Rainbow Tradeoff". In: *Journal of Applied Mathematics* 2014 (2014), 765394:1–765394:19 (cit. on p. 41).

[KHP13] Jung Woo Kim, Jin Hong, and Kunsoo Park. *Analysis of the Rainbow Tradeoff Algorithm Used in Practice*. Cryptology ePrint Archive, Report 2013/591. `http://eprint.iacr.org/2013/591`. 2013 (cit. on pp. 97–101, 103–106, 109–111, 116, 117, 121, 184).

[Kie04] Aaron Kiely. *Selecting the Golomb Parameter in Rice Coding*. Tech. rep. 42-159. Jet Propulsion Laboratory, Pasadena, California, JPL Publication: The Interplanetary Network Progress Report, Nov. 2004, p. 18 (cit. on pp. 60, 61).

[KL07] Guang Gong Khoongming Khoo Guanhan Chew and Hian-Kiat Lee. *Time-Memory-Data Trade-off Attack on Stream Ciphers based on Maiorana-McFarland Functions*. Cryptology ePrint Archive, Report 2007/242. `https://eprint.iacr.org/2007/242`. 2007 (cit. on p. 5).

[Kle13] Andreas Klein. *The eStream Project*. In: *Stream Ciphers*. London: Springer London, 2013, pp. 229–239. ISBN: 978-1-4471-5079-4 (cit. on p. 5).

[KM96] Koji Kusuda and Tsutomu Matsumoto. "Optimization of Time-Memory Trade-Off Cryptanalysis and Its Application to DES, FEAL-32, and Skipjack". In: *IEICE Transactions* E79-A.1 (Jan. 1996), pp. 35–48 (cit. on p. 26).

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. In: vol. 3. The Art of Computer Programming. Addison-Wesley, 1973. Chap. 6, pp. 506–541 (cit. on pp. 128, 129).

[Knu85] Donald E. Knuth. "Dynamic huffman coding". In: *Journal of Algorithms* 6.2 (1985), pp. 163–180. ISSN: 0196-6774 (cit. on p. 56).

[KPP+06] Sandeep Kumar, Christof Paar, Jan Pelzl, Gerd Pfeiffer, and Manfred Schimmler. *Breaking Ciphers with COPACOBANA - A Cost-Optimized Parallel Code Breaker*. In: *CHES 2006*. Ed. by Louis Goubin and Mitsuru Matsui. Vol. 4249. LNCS. Yokohama, Japan: Springer, Heidelberg, Germany, Oct. 2006, pp. 101–118 (cit. on p. 97).

[KPPM12] M. Kalenderi, D. Pnevmatikatos, I. Papaefstathiou, and C. Manifavas. *Breaking the GSM A5/1 cryptography algorithm with rainbow tables and high-end FPGAS*. In: *22nd International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Aug. 2012, pp. 747–753 (cit. on p. 84).

[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: *Progress in Cryptology - INDOCRYPT 2012*. Ed. by Steven D. Galbraith and Mridul Nandi. Vol. 7668. LNCS. Kolkata, India: Springer, Heidelberg, Germany, Dec. 2012, pp. 303–316 (cit. on pp. 70, 84, 85, 187).

[LH16] Ga Won Lee and Jin Hong. "Comparison of Perfect Table Cryptanalytic Tradeoff Algorithms". In: *Designs, Codes and Cryptography* 80.3 (Sept. 2016), pp. 473–523 (cit. on pp. 30, 158, 160, 162).

[Li16] Zhen Li. *Optimization of Rainbow Tables for Practically Cracking GSM A5/1 Based on Validated Success Rate Modeling*. In: *CT-RSA 2016*. Ed. by Kazue Sako. Vol. 9610. LNCS. San Francisco, CA, USA: Springer, Heidelberg, Germany, Feb. 2016, pp. 359–377 (cit. on p. 42).

[LLH15] Jiqiang Lu, Zhen Li, and Matt Henricksen. *Time-Memory Trade-Off Attack on the GSM A5/1 Stream Cipher Using Commodity GPGPU - (Extended Abstract)*. In: *ACNS 15*. Ed. by Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis. Vol. 9092. LNCS. New York, NY, USA: Springer, Heidelberg, Germany, June 2015, pp. 350–369 (cit. on pp. 5, 42, 84).

[LRCP17] Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. *Fast and Scalable Minimal Perfect Hashing for Massive Key Sets*. In: *16th International Symposium on Experimental Algorithms, SEA 2017, June 21-23, 2017, London, UK*. Ed. by Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman. Vol. 75. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 25:1–25:16. ISBN: 978-3-95977-036-1 (cit. on p. 134).

[MBPV05] Nele Mentens, Lejla Batina, Bart Preneel, and Ingrid Verbauwhede. *Cracking Unix Passwords using FPGA Platforms*. SHARCS - Special Purpose Hardware for Attacking Cryptographic Systems. Paris, France: Ecrypt, Feb. 2005 (cit. on p. 5).

[Meh82] Kurt Mehlhorn. *On the Program Size of Perfect and Universal Hash Functions*. In: *23rd FOCS*. Chicago, Illinois: IEEE Computer Society Press, Nov. 1982, pp. 170–175 (cit. on pp. 132, 133).

[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". In: *IEEE Communications Letters* 11.12 (2007), pp. 988–990 (cit. on p. 42).

[MH08] Daegun Ma and Jin Hong. "Success probability of the Hellman trade-off". In: *Information Processing Letters* 109.7 (Dec. 2008), pp. 347–351 (cit. on p. 26).

[MN98] Makoto Matsumoto and Takuji Nishimura. "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator". In: *ACM Trans. Model. Comput. Simul.* 8.1 (1998), pp. 3–30 (cit. on p. 46).

[NS05] Arvind Narayanan and Vitaly Shmatikov. *Fast Dictionary Attacks on Passwords Using Time-Space Tradeoff*. In: *ACM CCS 05*. Ed. by Vijayalakshmi Atluri, Catherine Meadows, and Ari Juels. Alexandria, Virginia, USA: ACM Press, Nov. 2005, pp. 364–372 (cit. on p. 78).

[Oec03] Philippe Oechslin. *Making a Faster Cryptanalytic Time-Memory Trade-Off*. In: *CRYPTO 2003*. Ed. by Dan Boneh. Vol. 2729. LNCS. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 2003, pp. 617–630 (cit. on pp. 5, 35, 36, 56, 107, 153, 156, 157).

[Oec04] Philippe Oechslin. *OPHCRACK (the time-memory-trade-off-cracker)*. Accessed: Jun 2014. 2004. Link. (Cit. on p. 35).

[Oec17] Philippe Oechslin. *Objectif Sécurité*. Accessed: Jul 2019. 2017. Link. (Cit. on p. 93).

[Pag18] Pierluigi Paganini. *Chat app Knuddels fined €20k under GDPR regulation*. Accessed: Feb 2019. 2018. Link. (Cit. on p. 45).

[Pag99] Rasmus Pagh. *Hash and Displace: Efficient Evaluation of Minimal Perfect Hash Functions*. In: *Algorithms and Data Structures, 6th International Workshop, WADS '99*. Ed. by Frank K. H. A. Dehne, Arvind Gupta, Jörg-Rüdiger Sack, and Roberto Tamassia. Vol. 1663. Lecture Notes in Computer Science. Vancouver, British Columbia, Canada: Springer Berlin Heidelberg, 1999, pp. 49–54. ISBN: 978-3-540-48447-9 (cit. on pp. 133, 134).

[Pan00] Vijay Pande. *Folding@ home*. Accessed: Jun 2019. 2000. Link. (Cit. on p. 7).

[Pri57] Robert C. Prim. "Shortest connection networks and some generalizations". In: *The Bell System Technical Journal* 36.6 (Nov. 1957), pp. 1389–1401. ISSN: 0005-8580 (cit. on p. 132).

[Ric79] Robert F. Rice. *Some practical universal noiseless coding techniques*. Tech. rep. NASA-CR-158515. Jet Propulsion Laboratory, Pasadena, California, JPL Publication, Mar. 1979, p. 132 (cit. on p. 60).

[Saa02] Markku-Juhani Olavi Saarinen. *A Time-Memory Tradeoff Attack Against LILI-128*. In: *Fast Software Encryption*. Ed. by Joan Daemen and Vincent Rijmen. Vol. 2365. LCNS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 231–236. ISBN: 978-3-540-45661-2 (cit. on p. 5).

[Sag85] Thomas J. Sager. "A Polynomial Time Generator for Minimal Perfect Hash Functions". In: *Commun. ACM* 28.5 (May 1985), pp. 523–532. ISSN: 0001-0782 (cit. on pp. 131, 132).

[Shu09] Zhu Shuanglei. *RainbowCrack*. Accessed: Apr 2017. 2009. Link. (Cit. on pp. 7, 93, 187).

[Sie13] Matthieu Sieben. *Accelerating the computation of rainbow tables for cryptanalytic Time-memory Trade-offs*. MA thesis. Louvain-la-Neuve: Ecole polytechnique de Louvain, 2013 (cit. on p. 187).

[Spi07] Stefan Spitz. *Time Memory Tradeoff Implementation on Copacobana.* MA thesis. Bochum, Germany: Ruhr-Universität Bochum, June 2007 (cit. on pp. 97, 98).

[Spr77] Renzo Sprugnoli. "Perfect Hashing Functions: A Single Probe Retrieving Method for Static Sets". In: *Commun. ACM* 20.11 (Nov. 1977), pp. 841–850. ISSN: 0001-0782 (cit. on pp. 129, 130).

[SRQL03] François-Xavier Standaert, Gaël Rouvroy, Jean-Jacques Quisquater, and Jean-Didier Legat. *A Time-Memory Tradeoff Using Distinguished Points: New Analysis & FPGA Results.* In: *CHES 2002.* Ed. by Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar. Vol. 2523. LNCS. Redwood Shores, CA, USA: Springer, Heidelberg, Germany, Aug. 2003, pp. 593–609 (cit. on pp. 30–32).

[SS90] Jeanette P. Schmidt and Alan Siegel. "The Spatial Complexity of Oblivious k-Probe Hash Functions". In: *SIAM J. Comput.* 19.5 (1990), pp. 775–786. ISSN: 0097-5397 (cit. on p. 133).

[TO05] Cedric Tissières and Philippe Oechslin. *Ophcrack*. Accessed: Apr 2017. 2005. Link. (Cit. on pp. 7, 35, 56, 93, 99, 162).

[TOL13] Cedric Tissieres, Philippe Oechslin, and Pierre Lestringant. *Limites des tables Rainbow et comment les dépasser en utilisant des méthodes probabilistes optimisées.* In: *Actes du 11ème symposium sur la sécurité des technologies de l'information et des communications (SSTIC).* 2013 (cit. on p. 78).

[TSG11] Guangming Tan, Vugranam C. Sreedhar, and Guang R. Gao. "Analysis and performance results of computing betweenness centrality on IBM Cyclops64". In: *The Journal of Supercomputing* 56.1 (Apr. 2011), pp. 1–24. ISSN: 1573-0484. Link. (Cit. on p. 93).

[TY79] Robert Endre Tarjan and Andrew Chi-Chih Yao. "Storing a Sparse Table". In: *Commun. ACM* 22.11 (Nov. 1979), pp. 606–611. ISSN: 0001-0782 (cit. on p. 133).

[VGB12] Roel Verdult, Flavio Garcia, and Josep Balasch. *Gone in 360 Seconds: Hijacking with Hitag2.* In: *In 21st USENIX Security Symposium.* USENIX, Jan. 2012, pp. 237–252 (cit. on p. 5).

[VGE15] Roel Verdult, Flavio D. Garcia, and Baris Ege. *Dismantling Megamos Crypto: Wirelessly Lockpicking a Vehicle Immobilizer.* In: *Supplement to the Proceedings of 22nd USENIX Security Symposium (Supplement to USENIX Security 15).* Washington, D.C.: USENIX Association, 2015, pp. 703–718. ISBN: 978-1-931971-232 (cit. on p. 5).

[Whi18] Zack Whittaker. *At Blind, a security lapse revealed private complaints from Silicon Valley employees.* Accessed: Feb 2019. 2018. Link. (Cit. on p. 45).

[WL13] Wenhao Wang and Dongdai Lin. *Analysis of Multiple Checkpoints in Non-perfect and Perfect Rainbow Tradeoff Revisited.* In: *ICICS 13.* Ed. by Sihan Qing, Jianying Zhou, and Dongmei Liu. Vol. 8233. LNCS. Beijing, China: Springer, Heidelberg, Germany, Nov. 2013, pp. 288–301 (cit. on p. 70).

[WNS14] Martin Westergaard, James Nobis, and Zhu Shuanglei. *Rcracki-mt.* Accessed: Apr 2017. 2014. Link. (Cit. on pp. 98, 105).

[WW09a] Pinchas Weisberg and Yair Wiseman. *Using 4KB Page Size for Virtual Memory is Obsolete.* In: *Proceedings of the IEEE International Conference on Information Reuse and Integration – IRI 2009.* Las Vegas, Nevada, USA: IEEE Systems, Man, and Cybernetics Society, Aug. 2009, pp. 262–265 (cit. on p. 97).

[WW09b] Pinchas Weisberg and Yair Wiseman. *Using 4KB Page Size for Virtual Memory is Obsolete.* In: *Proceedings of the IEEE International Conference on Information Reuse and Integration, IRI 2009, 10-12 August 2009, Las Vegas, Nevada, USA*. 2009, pp. 262–265 (cit. on p. 110).

[YLL+11] Xue-Jun Yang, Xiang-Ke Liao, Kai Lu, Qing-Feng Hu, Jun-Qiang Song, and Jin-Shu Su. "The TianHe-1A Supercomputer: Its Hardware and Software". In: *Journal of Computer Science and Technology* 26.3 (May 2011), pp. 344–351. ISSN: 1860-4749 (cit. on p. 93).

[YSU+11] M. Yokokawa, F. Shoji, A. Uno, M. Kurokawa, and T. Watanabe. *The K computer: Japanese next-generation supercomputer development project.* In: *IEEE/ACM International Symposium on Low Power Electronics and Design*. Nov. 2011, pp. 371–372 (cit. on p. 93).

[ZL77] Jacob Ziv and Abraham Lempel. "A universal algorithm for sequential data compression". In: *IEEE Transactions on Information Theory* 23.3 (1977), pp. 337–343 (cit. on p. 56).

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Considérations Pratiques sur les Compromis Temps-Mémoire Cryptanalytiques

Mot clés : mots de passe, sécurité, compromis temps-mémoire, cryptanalyse

Résumé : Un compromis temps-mémoire cryptanalytique est une technique qui vise à réduire le temps nécessaire pour effectuer certaines attaques cryptographiques telles que l'inversion d'une fonction à sens unique. Une telle inversion intervient dans une des principales applications des compromis temps-mémoire : le cassage de mots de passe. La technique requiert un très lourd pré-calcul qui génère des tables utilisables pour accélérer la recherche exhaustive de l'attaque. L'attaque par compromis temps-mémoire est d'autant plus rapide qu'il y a de mémoire allouée à l'algorithme. Cependant, en pratique, la mémoire est souvent un facteur limitant. Nous évaluons l'impact d'un problème nécessitant une grande mémoire sur la technique des compromis temps-mémoire, notamment en se plaçant dans le contexte où une mémoire externe lente est utilisée à la place d'une mémoire rapide limitée (RAM). Nous établissons qu'une telle approche est applicable dans des cas pratiques, qui sont identifiés. Nous proposons ensuite une nouvelle construction de compromis temps-mémoire qui repose sur des fonctions de hachage minimales parfaites, et dont le stockage est moindre que sur les techniques de compression de tables existantes. Finalement, nous proposons une comparaison entre les améliorations existantes, possiblement combinées, et notre nouvelle technique.

Title: Practical Considerations on Cryptanalytic Time-Memory Trade-Offs

Keywords: passwords, security, time-memory trade-off, cryptanalysis

Abstract: A cryptanalytic time-memory trade-off (TMTO) is a technique that aims to reduce the time needed to perform a set of cryptanalysis attacks, such as inverting a one-way function. Such an inversion constitutes one of the main applications of TMTOs, which is password cracking. The technique relies on a large-scale pre-computation which outputs tables that allow to significantly speed up the attack's exhaustive search. The more memory is used by a TMTO, the faster the attack can be. In practice, the amount of memory available is often the limiting factor, so numerous approaches have been proposed to fit large tables in a restricted amount of memory. In this thesis, we focus on the rainbow tables variant, the most widely spread version of time-memory trade-offs. When the considered cryptographic problem is overwhelmingly sized, using an external memory is eventually needed. We analyse the relevance of using an external memory instead of RAM, and we state that it is fully suited for practical cases, which are identified. We then introduce a new technique, based on minimal perfect hash functions, whose storage complexity is better than any previous optimisation. Finally, we analyse and compare existing TMTO approaches as well as their combinations, along with our newly introduced MPHF rainbow technique. We are then able to provide a set of practical recommendations on how to configure the implementation of a TMTO in an optimal way.