PROPOSTA DE UM *TRADE-OFF* CRIPTOANALÍTICO NÃO PROBABILÍSTICO

Frederico Schardong[1]
Daniel Formolo[2]

Resumo: Este trabalho propõe um novo método de *trade-off* criptoanalítico não probabilístico. Ele apresenta os principais *trade-off* criptoanalíticos, fazendo um paralelo com o método proposto. Apesar do número de operações de hash para recuperar um elemento serem altas em comparação aos métodos tradicionais, o novo método tem como vantagem o sucesso garantido na recuperação de hashes e número de operações de leitura de disco mínimas e sequenciais, ao contrários dos principais *trade-off* probabilísticos existentes.

Palavras-chave: *Trade-off*. Hash. Criptoanálise

1 INTRODUÇÃO

Funções de hash são amplamente utilizadas para proteger informações sensíveis como senhas. Diferentemente das funções de criptografia simétrica onde uma chave é necessária para recuperar o conteúdo encriptado por esta mesma chave, as funções de hash não são reversíveis. Essa diferença fundamental permite que o resultado de uma função de hash possa ser armazenado em um ambiente inseguro, desde que essa função de hash seja considerada segura, ou seja, não possuir ataques ou falhas conhecidas. Entretanto, se o conteúdo utilizado como entrada da função de hash for de poucos bits ou seguir padrões específicos, atacantes que eventualmente obterem estes hashes poderão compará-los com hashes que possuem e conhecem seus textos planos. Caso algum seja igual então o atacante poderá utilizá-lo no sistema alvo e facilmente se passar pelo usuário legítimo, caso o sistema não possua métodos adicionais de autenticação (STALINGS, 2005).

O uso de apenas uma função de hash para proteger as senhas de usuários pode comprometer a segurança de sistemas criptográficos. Tais implementações são suscetíveis a vários tipos de ataques: força bruta, dicionário de senhas/palavras e mais atualmente *rainbow table*. Tanto o uso de dicionários como o uso de *rainbow table* são ataques onde uma pré-computação é necessária, neles é preciso criar a lista de palavras do dicionário ou computar a *rainbow table* para então aplicar o ataque. Atualmente, o poder de processamento e capacidade de armazenamento são grandes. Mesmo assim, esses recursos são muito demandados pelos ataques, portanto, o equilíbrio no uso desses recursos se adequando a situações e condições de hardware são essenciais para quebra de chaves criptográficas em tempo aceitável. O ataque de dicionário consiste em buscar por um hash específico em um dicionário onde senhas foram previamente geradas e armazenadas (NARAYANAN; SHMATIKOV, 2005), enquanto que *rainbow table* consiste em

[1]Aluno do curso de Segurança da Informação. Email: frede.sch@gmail.com
[2]Orientador, professor da Unisinos, mestre em Ciência da Computação pela Universidade do Vale do Rio dos Sinos (2009). Email: formolo.unisinos@gmail.com.br

um ataque de dicionário mais sofisticado pois é criado de tal forma que seu tamanho final seja muito inferior em comparação a um dicionário com o mesmo número de elementos, porém possui um custo de processamento extra em comparação ao ataque de dicionário (OECHSLIN, 2003).

Os ataques citados podem ser classificados como *trade-off*, pois são algoritmos onde deve-se escolher entre mais processamento ou maior consumo de memória (HELLMAN, 1980). O ataque de força bruta requer muito processamento pois todas as senhas de um determinado grupo são geradas, porém nenhuma é armazenada em memória. Ataques a sistemas criptográficos que usam hashes das senhas originais são realizados comparando-se hashes gerados com o hash que se deseja quebrar, se forem iguais o algoritmo é finalizado mostrando qual senha gerou aquele hash, caso contrário o hash testado é descartado e uma nova senha é gerada e seu hash comparado com o fornecido, e assim sucessivamente.

No caso do ataque de dicionário, um dicionário deve ser previamente gerado e armazenado em memória. Neste caso, muita memória é utilizada para armazenar o dicionário, porém pouco processamento é necessário durante o ataque visto que o hash a ser quebrado é comparado apenas com os hashes contidos no dicionário. Combinações entre o ataque de força bruta e o ataque de dicionário são possíveis, para cada palavra contida em um dicionário pode-se gerar uma série de números que são concatenados a esta palavra, estes resultados devem ser submetidos a função de hash e o seu resultado comparado com o hash a ser quebrado. Combinando estes dois métodos, o *trade-off* é mais equilibrado, pois tanto memória quanto processamento serão necessários no ataque.

1.1 *Objetivos*

Grande parte dos trabalhos atuais sobre *trade-off* para quebra de hashes se baseiam em *rainbow table* (YING; THING, 2011), (ÅGREN; JOHANSSON; HELL, 2009) e (KIM; HONG, 2013). Neste trabalho será proposto um novo algoritmo de *trade-off* que difere das idéias de Oechslin (2003) e Hellman (1980), criando uma nova abordagem para este problema.

1.2 Objetivos Específicos

Implementar, analisar e comparar o desempenho do algoritmo proposto com o método de Hellman e o de Oechslin.

1.3 *Organização do Artigo*

Este artigo inicia abordando as funções hash no Capítulo 2, explicando onde são utilizadas e ainda cita algoritmos de hash considerados seguros e inseguros atualmente. O Capítulo 3 descreve os tipos de ataques relativos as funções de hash. Posteriormente, no capítulo 4 o mé-

todo de Hellman e trabalhos recentes relevantes sobre esta técnica são abordados em detalhes. O Capítulo 5 descreve o principal trabalho publicado baseado na técnica de Hellman, *rainbow table*. O Capítulo 6 descreve a metodologia de pesquisa empregada neste trabalho e o Capítulo 7 aborda o funcionamento do algoritmo de delta encoding, bem como a definição do algoritmo de *trade-off* criptoanalítico não probabilístico através de um teorema indutivo e uma análise teórica de sua operação. No Capítulo 8 exemplos de uso são fornecidos para mostrar as características de *trade-off* do algoritmo proposto e por fim uma comparação é feita com os métodos de Hellman (HELLMAN, 1980) e Oechslin (OECHSLIN, 2003). O Capítulo 9 encerra este artigo com considerações finais em relação aos resultados obtidos bem como sugere trabalhos futuros.

2 FUNÇÕES DE HASH

Funções de hash são funções de mão única, ou seja, são funções fáceis de serem computadas para qualquer entrada porém é difícil recuperar suas entradas tendo posse dos resultados gerados. Esse tipo de situação também é conhecido como o problema de P versus NP (DIFFIE; HELLMAN, 1976).

Funções de hash são amplamente utilizadas desde assinaturas digitais (STALLINGS, 2007) até mecanismos para autenticação de mensagens (BELLARE; CANETTI; KRAWCZYK, 1996), (KRAWCZYK; CANETTI; BELLARE, 1997) e (BLACK et al., 1999), geradores pseudo-aleatórios (IMPAGLIAZZO; LEVIN; LUBY, 1989), indexar dados (PARK; CHEN; YU, 1995), detecção de partes de arquivos duplicadas (PUGH; HENZINGER, 2003) e também detecção de arquivos inteiramente duplicados simplesmente calculando o hash de dois arquivos e comparando-os.

Armazenamento seguro de senhas também utiliza funções de hash, normalmente em conjunto com *salt*, uma técnica que consiste em concatenar algum valor ao texto plano (senha) antes do calculo do seu hash, tal valor pode ser constante ou não (GAURAVARAM, 2012).

MD5 é uma função de hash implementada por Rivest (1992) com o objetivo de substituir o MD4, que é uma função hash considerada insegura devido a facilidade de gerar colisões (SASAKI et al., 2007). Colisões acontecem em funções de hash quando duas entradas diferentes produzem a mesma saída. A não ser pelas funções de hash perfeitas (FREDMAN; KOMLÓS; SZEMERÉDI, 1984), todas as funções de hash produzem colisões, isso acontece devido a relação entre a infinita quantidade de entradas e o número finito de possibilidades de saídas que as funções de hash produzem.

O MD5 gera um resultado de tamanho constante de 128 bits. Wang e Yu (2005) criaram um método que com 2^{39} operações encontra colisões entre hashes MD5, provando assim que esta função não é segura, e consequentemente não deve ser utilizada.

Outra função de hash largamente utilizada é o SHA-1, publicado pelo NIST (*National Institute of Standards and Technology*) em 1995 (STANDARD, 1995). Ao contrário do MD5, o

SHA-1 possui uma saída de 160 bits. Rijmen e Oswald (2005) foram os primeiros pesquisadores a reportarem colisões no SHA-1 com 2^{80} operações. Em seguida, Wang, Yin e Yu (2005) criaram um método capaz de encontrar colisões em apenas 2^{69} operações. Desde então o SHA-1 é considerado inseguro e seu uso desaconselhável.

Alguns anos depois o NIST lançou o SHA-2 que produz hashes de tamanhos diferentes: 224, 256, 384 ou 512 bits (PUB, 2004). Até a publicação deste trabalho, nenhuma criptoanálise efetiva foi proposta e este algoritmo de hash continua sendo considerado seguro.

3 ATAQUES TRADICIONAIS

Os dois principais métodos para a quebra de hashes são a força bruta e o ataque de dicionário. Ambos ataques objetivam descobrir o valor de entrada aplicado a uma função de hash conhecida H, que gerou um hash h, igualmente conhecido. Cada uma das duas abordagens tem suas características de uso de recursos computacionais. Para entender melhor essas características e como os métodos tradicionais e modernos balanceiam o uso de processamento e memória foi criado o conceito de *trade-off*.

Hellman (1980) define *trade-off* como: "se existem N possíveis soluções a serem buscadas, o *trade-off* entre tempo e memória permite que uma solução seja encontrada em T operações (tempo) usando M palavras de memória onde TM igual a N". A definição de *trade-off* de Hellman (1980) permite os dois extremos, quando T for igual a 1 e M for igual a N, apenas uma operação de hash é necessária porém o número de palavras de memória utilizadas é igual a N, este é o ataque de dicionário. Ou também, quando M for igual a 1 e T ser igual a N, caracteriza-se o ataque de força bruta.

3.1 Força Bruta

Dado um código hash h, deseja-se descobrir o texto plano do qual h foi gerado então, pelo ataque de força bruta um conjunto de textos planos é escolhido, um ou mais computadores se empenham em gerar este conjunto e simultaneamente aplicam a mesma função de hash H em cada um dos textos planos. Todo hash $H(x)$ gerado é comparado com o hash h fornecido, caso forem iguais então o texto plano que gerou o hash h é o mesmo texto plano que gerou o hash $H(x)$ (ignorando a baixa probabilidade de uma colisão). Caso nenhum hash gerado for igual ao hash fornecido h então o texto plano que gerou h não está contido no intervalo de textos planos definido no início da execução do algoritmo.

Supondo que a lista de textos planos é sempre fixa, o processamento necessário para gerar os hashes de todos os elemento desta lista será repetido a cada execução pois nenhum hash é armazenado em memória para uso posterior. Esta característica pode ser tanto uma vantagem quanto uma desvantagem pois sistemas computacionais com pouco poder de processamento e muita memória disponível poderiam armazenar a lista de textos planos e o hash de cada

Tabela 1 – Exemplo de um dicionário de hashes MD5

Texto Plano	Hash
a	$0cc175b9c0f1b6a831c399e269772661$
c	$4a8a08f09d37b73795649038408b5f33$
d	$8277e0910d750195b448797616e091ad$
b	$92eb5ffee6ae2fec3ad71c777531578f$
e	$e1671797c52e15f763380b45e841ec32$

Fonte: Elaborado pelo autor.

elemento em memória, não sendo vantajoso gerar todos os hashes novamente. Entretando, para um sistema computacional com muito poder de processamento e pouca memória para armazenamento pode ser trivial gerar todos os hashes a cada execução.

Utilizando a definição de Hellman (1980) sobre *trade-off*, o ataque de força bruta é o extremo do *trade-off* onde $T = N$, sendo N o total de elementos da lista e T o total de operações de hash necessárias para cobrir todos os textos planos.

3.2 Dicionário

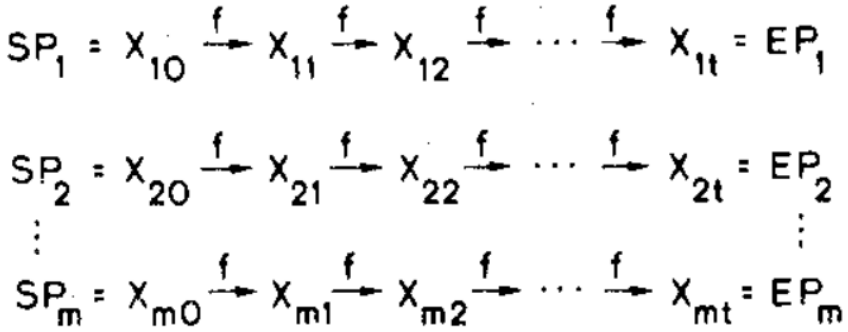
O ataque de dicionário é similar ao ataque de força bruta, sua principal diferença é que os hashes de todos os elementos da lista de textos planos é armazenado (juntamente com os seus respectivos elementos), ou seja, os elementos da lista de textos planos são submetidos a função de hash H apenas uma vez. Este processo de criação do dicionário é chamado de pré-processamento e o seu custo é normalmente desprezível pois é executado apenas uma vez. Por exemplo, se a criação do dicionário custar 1 milhão de reais (considerando fatores como aluguel do hardware, energia elétrica, manutenção, etc) e este dicionário for utilizado 1 milhão de vezes com o custo de R$ 100,00 reais por utilização (energia elétrica, manutenção, etc) então o custo da geração do dicionário pode ser diluído entre as execuções e representaria menos de 1% de cada execução.

Depois de sua criação, o dicionário é ordenado pelos hashes, de modo que seja possível encontrar qualquer elemento em até $log_2(N)$ operações, sendo N o número total de elementos no dicionário (SLEATOR; TARJAN, 1985).

A Tabela 1 exemplifica um dicionário simples, com apenas 5 entradas. A lista de elementos utilizada foi a, b, c, d, e e a função de hash escolhida foi o MD5.

Segundo a definição de *trade-off* criada por Hellman (1980) o ataque de dicionário seria o extremo oposto do ataque de força bruta, pois para realizar o ataque, o único processamento T necessário é a busca do hash h fornecido no dicionário. Como todos os elementos do grupo N estão salvos em memória então $N = M$.

Figura 1 – Criação de EP_i a partir de SP_i



Fonte: Hellman (1980)

4 MÉTODO DE HELLMAN

Para demonstrar que informações importantes não devem ser confiadas a sistemas com chaves curtas, Hellman (1980) propôs um *trade-off* capaz de inverter um elemento encriptado de um sistema com N elementos em $N^{2/3}$ operações usando $N^{2/3}$ palavras de memória. Desconsiderando o custo de pré-computação para criar as tabelas, Hellman foi capaz de reduzir o custo de quebra do DES em modo de bloco de $\$5000$ através de força bruta para $\$10$ utilizando o seu método.

Dado um conjunto de N elementos a serem armazenados neste sistema, e como parte da pré-computação, m elementos são aleatóriamente selecionados do conjunto $\{1, 2, ..., N\}$ criando os pontos inicia $SP_1, SP_2, ..., SP_m$. Obtendo-se:

$$X_{i0} = SP_i \qquad 1 \leq i \leq m \tag{1}$$

E então computa-se:

$$X_{ij} = f(X_{ij-1}) \qquad 1 \leq j \leq t \tag{2}$$

Sendo t o número de rodadas que uma função f será executada. Após t execuções da função f o último elemento de uma coluna i será:

$$EP_i = f^t(SP_i) \tag{3}$$

A função f consiste no resultado da aplicação de uma função de redução r em um elemento encriptado ou aplicado a uma função de hash. A função de redução r deve reduzir a saída de uma função de hash ou de criptografia simétrica aplicada aos elementos do conjunto N de tal maneira que o resultado seja algum elemento aleatório do mesmo conjunto. A Figura 1 ilustra a criação da tabela de Hellman.

Para reduzir a quantidade de memória, todos os pontos intermediários são removidos e $\{SP_i, EP_i\}_{i=1}^{m}$ são organizados pelos pontos finais (EP) e armazenados em uma tabela. Múl-

tiplas tabelas podem ser criadas onde cada tabela deve usar uma função de redução r diferente para evitar colisões.

Dado um elemento encriptado ou algum hash K busca-se na tabela $Y_1 = r(K)$, se não for encontrado significa que texto plano que gerou K não é nenhum $X_{i,t}$. Aplica-se então $Y_2 = f(Y_1)$ e este elemento então é buscado na tabela, se não for encontrado significa que não é $X_{i,t-1}$. Repetindo o passo anterior aplica-se $Y_3 = f(Y_2)$ e então outra busca é realizada na tabela e este processo se repete até que algum EP_i seja encontrado ou até atingir $X_{i,0}$.

Supondo que EP_i é igual a Y_3, isto significa que o texto plano que gerou K está na posição $X_{i,t-2}$. Entretanto, como todos os pontos intermediários foram descartados, toda a linha a partir de SP_i precisa ser recriada para que o elemento na posição $X_{i,t-2}$ seja encontrado e então encriptado para testar se é igual a K. Caso seja diferente, um alarme falso foi encontrado e a execução do algoritmo continua. Alarmes falsos são uma consequência da natureza probabilística desde método. Como o resultado de uma função de encriptação ou de hash é encurtado a probabilidade de alarmes falsos, ou seja, de colisões acontecerem pode ser grande dependendo das configurações utilizadas.

Segundo Hellman se todos os mt elementos forem diferentes então a probabilidade de sucesso $P(S)$ de uma tabela é de $P(S) = mt/N$. A probabilidade de sucesso de uma busca por força bruta é $P(S) = t/N$ e de uma busca por dicionário é de $P(S) = t/N$. Hellman afirma que se a função f for modelada como uma função que mapeia aleatoriamente elementos de N para ele mesmo então a probabilidade de sucesso de uma tabela é limitada por:

$$P(S) \geq (1/N) \sum_{i=1}^{m} \sum_{j=0}^{t-1} [(N - it)/N]^{j+1} \tag{4}$$

A prova matemática desde teorema pode ser encontrado no artigo de Hellman (HELLMAN, 1980). Ele ainda afirma que se $mt^2 = N$, então a resolução da equação 4 resultará em uma probabilidade de sucesso de pelo menos 80%.

Hellman define que o número esperado de alarmes falsos por tabela é:

$$E(F) \leq mt(t + 1)/2N \tag{5}$$

Utilizando o método de Hellman quebrar os 56 bits do *Data Encryption Standard* ou DES (STANDARD, 1977) é menos complexo do que aplicar força bruta em um sistema com uma saída de 38 bits (HELLMAN, 1980).

4.1 Otimização de Parâmetros

Hellman (1980) sugere que os parâmetros para a criação de tabelas devem ser escolhidos tal que $mt^2 = N$ sendo N o total de elementos, m o número de linhas da tabela e t o comprimento de cada cadeia de elementos. Escolhendo os parâmetros como proposto pelo autor então processamento e memória estarão balanceados no *trade-off*.

Hellman (1980) afirma que o limite inferior da taxa de sucesso de $mt^2 = N$ é $0,80$, Saran e Doganaksoy (2009) propuseram a seguinte fórmula para calcular a exata taxa de sucesso de uma tabela Hellman:

$$R(m,t) = \sqrt{\frac{2N}{mt^2}} \tanh\left(\sqrt{\frac{mt^2}{2N}}\right) \tag{6}$$

Através desta fórmula é possível calcular a taxa de sucesso exata de $mt^2 = N$, que é de $0,86$. Saran e Doganaksoy (2009) definem a função λ tal que:

$$\lambda = \frac{mt^2}{N} \tag{7}$$

A equação 1 pode ser simplificada para:

$$R(m,t) = \sqrt{\frac{2}{\lambda}} \tanh\left(\sqrt{\frac{\lambda}{2}}\right) \tag{8}$$

Com esta fórmula é evidente que uma tabela Hellman pode ser facilmente projetada de forma a permitir a melhor escolha de parâmetros para cenários específicos que usem mais processamento que memória ou vice-versa. Os pesquisadores mostram através de resultados empíricos que é possível obter uma maior taxa de probabilidade sem alterar o tamanho total das tabelas Hellman. A Tabela 2 mostra múltiplas configurações, variando λ de 4 até 1/4 para $N = 2^{21}$ utilizando o algoritmo hash SHA-1 reduzido. A coluna r representa o número total de tabelas criadas.

Tabela 2 – Comparação de SHA-1 reduzido com $N = 2^{21}$

λ	m	t	r	$R(m,t)$
4	128	256	64	$0,63$
2	64	256	128	$0,76$
1	128	128	128	$0,86$
$1/2$	64	128	256	$0,92$
$1/4$	128	64	256	$0,96$

Fonte: Saran e Doganaksoy (2009)

É evidente que quanto menor for λ, menor será a probabilidade de uma cadeia de elementos colidir com outra, aumentando assim a taxa de sucesso de cada tabela. Entretando, muitas tabelas deverão ser criadas para cobrir todo o conjunto N.

Os pesquisadores ainda mostram como é possível atingir altas taxas de sucesso com uma quantidade de processamento t constante de $2^{13,3}$ bem como o tamanho total de todas as tabelas constante, apenas alterando m e r conforme mostrado na Tabela 3. Além desse estudos outros métodos baseados em Hellman foram desenvolvidos, dentre eles, o de maior contribuição chama-se *Rainbow Table*.

Tabela 3 – Diferentes construções da tabela de Hellman para $N = 2^{40}$

m	r	λ	$R(m,t)$
11	10327588	2^{-10}	$0,999$
169	645474	2^{-6}	$0,997$
676	161369	2^{-4}	$0,989$
2702	40342	2^{-2}	$0,960$
5405	20171	2^{-1}	$0,924$
10809	10086	2^{0}	$0,861$
216189	5043	2^{1}	$0,076$
43238	2521	2	$0,628$

Fonte: Saran e Doganaksoy (2009)

5 *RAINBOW TABLE*

Oechslin (2003) propõem que a tabela de Hellman seja alterada de forma a usar múltiplas funções de redução ao invés de apenas uma. No método de Hellman, em cada rodada de encurtamento utiliza-se a mesma função r enquanto que na *rainbow table* cada rodada utiliza uma função de encurtamento r_i diferente. A primeira rodada sempre utilizará a função de encurtamento r_1, a segunda r_2 e assim por diante até a última rodada t utilizar a função de redução r_{t-1}.

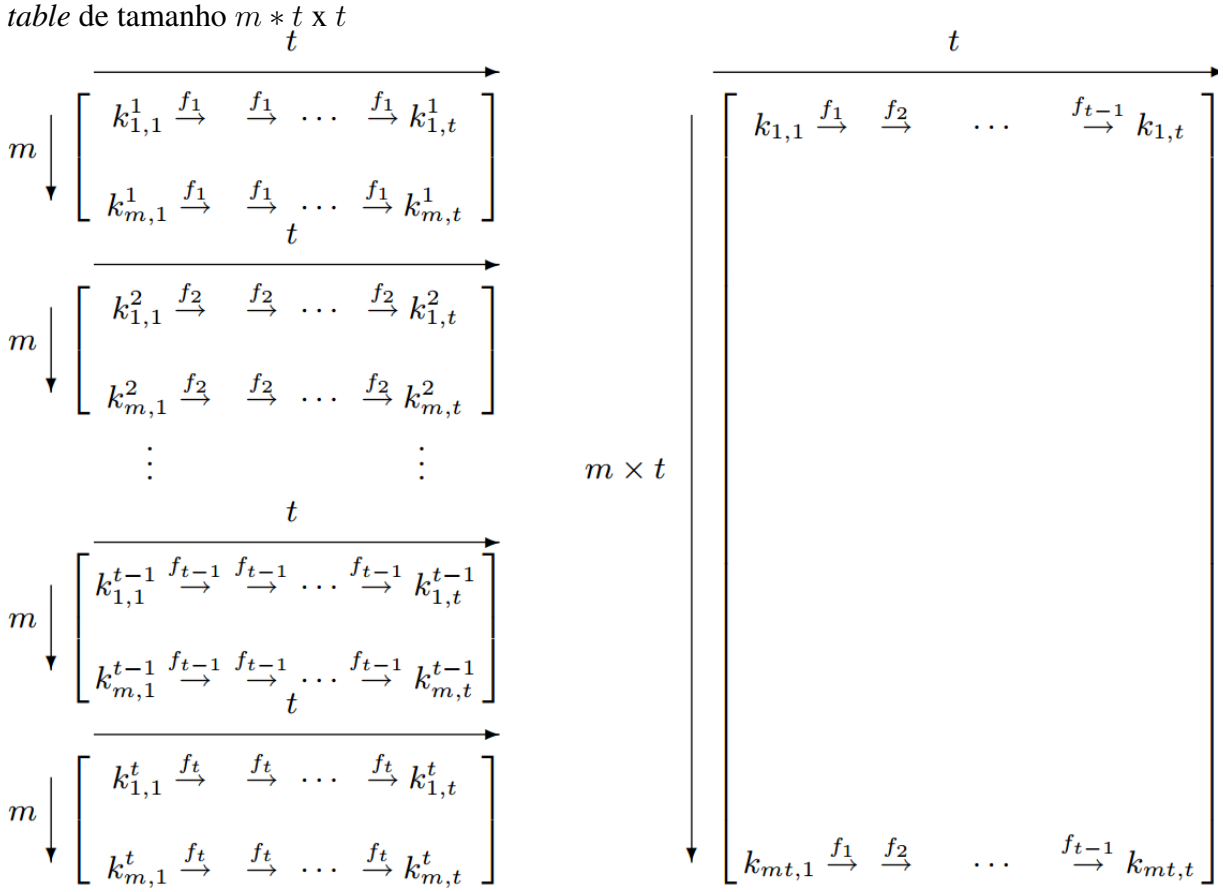
Através desta alteração a probabilidade que colisões aconteçam é reduzida drasticamente. A Figura 2 compara múltiplas tabelas de Hellman com uma *rainbow table*, ilustrando como t funções de encurtamento são utilizadas em uma única tabela.

Para recuperar um valor K em uma *rainbow table* primeiro aplica-se $r_{t-1}(K)$ e caso este valor não esteja presente nos pontos finais da tabela então aplica-se $r_{t-2}(f_{t-1}(K))$, se este resultado não estiver presente nos pontos finais da tabela então o próximo passo é aplicar $r_{t-3}(f_{t-2}(f_{t-1}(K)))$ e assim por diante. Recuperar um elemento em uma *rainbow table* resulta em $t(t-1)/2$ operações enquanto que recuperar um elemento em t tabelas de Hellman com dimensões m linhas x t colunas resulta em t^2 operações.

Oechslin criou 4666 tabelas de Hellman com dimensões de $m_h = 8192$ e $t_h = 4666$ e uma *rainbow table* com as dimensões $t_r = 4666$ e $m = m_h * t_h = 38223872$ ambas usando o mesmo conjunto N de 2^{37} hashes diferentes de 8 bytes cada. Nesta implementação a probabilidade de sucesso foi mensurada através da busca de 500 hashes aleatórios em 78.8% da *rainbow table* e 75.8% das tabelas de Hellman. Como resultado do experimento, a busca por elementos na *rainbow table* foi 7 vezes mais rápida com uma média de 9.3 milhões de cálculos de hash contra 67.2 milhões usando o método de Hellman.

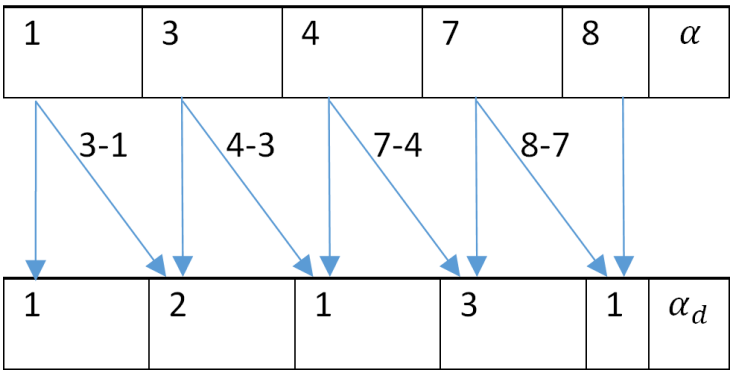
Zhang et al. (2010) propõem que a *rainbow table* seja dividida em partes menores e distribuidas entre computadores numa relação cliente-servidor de modo que cada cliente receba uma parte diferente da tabela e possa contribuir para a busca de chaves, transformando o problema de duas dimensões $\{M, T\}$ em um problema de três dimensões $\{M, T, R\}$.

Figura 2 – Criação de t tabelas de Hellman de tamanho m linhas x t colunas versus uma *rainbow table* de tamanho $m * t$ x t



Fonte: Oechslin (2003)

Figura 3 – Criação de α_d a partir de α



Fonte: Elaborado pelo autor.

O protocolo que coordena o trabalho entre cada cliente e o servidor consiste em chamadas síncronas que não representam um custo significativo. A implementação prova a análise teórica feita pelos autores de que o tempo do ataque diminui linearmente com o incremento do número de clientes (recursos).

6 DELTA ENCODING

Delta encoding é uma técnica de compressão de dados que consiste em calcular a diferença entre dois objetos, sejam eles arquivos, números, strings ou outros. Existem inúmeras aplicações que se beneficiam desta técnica simples (complexidade $O(n)$) desde backup incremental (BURNS; LONG, 1997), sistemas de desduplicação (PAULO; PEREIRA, 2014), compressão de arquivos (SUEL; TRENDAFILOV, 2002), cache dinâmico de conteúdos para web (PSOUNIS, 2002) até compressão de quadros HTTP inteiramente (MOGUL et al., 1997) e sincronização de arquivos remotos (SUEL; MEMON, 2002).

Segundo Mogul et al. (1997) a técnica de calcular a diferença entre objetos já era utilizada pelo padrão de compressão de video MPEG-1. O MPEG-1 ocasionalmente envia um quadro inteiramente novo, ou seja, uma imagem nova e os próximos quadros são apenas informações para reconstruir o que foi alterado em comparação com o quadro anterior.

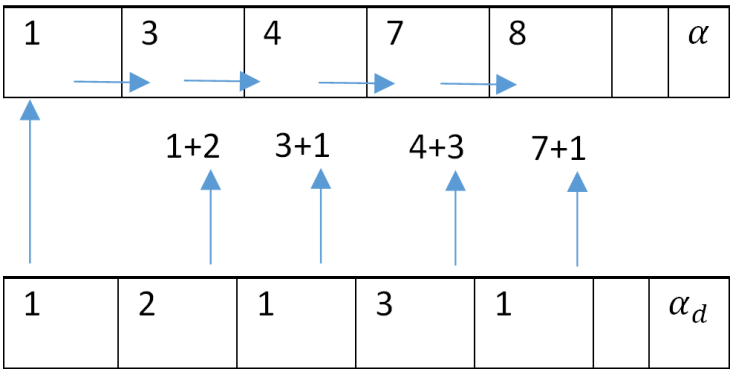
Naturalmente quanto menor a diferença entre os dados, maior será a performance do delta encoding. Por exemplo o conjunto $\alpha = \{1, 3, 4, 7, 8\}$ resultará em $\alpha_d = \{1, 2, 1, 3, 1\}$ tal que $\alpha_d = D(\alpha)$ sendo que $D(x)$ é uma função que calcula o delta encoding de x. O conjunto α precisa de pelo menos 20 bits para ser armazenado pois o maior elemento (8) ocupa 4 bits, logo $5 * 4 = 20$ bits. Entretanto, o delta encoding desde conjunto (α_d) pode ser armazenado em apenas 15 bits.

O algoritmo de delta encoding que transforma o conjunto α em α_d é ilustrado na Figura 3.

Delta encoding também é capaz de codificar conjuntos desordenados. Por exemplo um conjunto $\{2, 5, 7, 4, -3, 8, 23, 9, 4\}$ resultará em $\{2, 3, 2, -3, 7, 11, 15, -14, -5\}$.

A recuperação de deltas também possui complexidade $O(n)$ e é ilustrada na Figura 4.

Figura 4 – Recuperação de α a partir de α_d



Fonte: Elaborado pelo autor.

7 METODOLOGIA DE PESQUISA

O trabalho utiliza o método quantitativo através de uma comparação entre o número médio de operações de hash necessárias para recuperar um hash no método de Hellman, *rainbow table* e no algoritmo proposto. O método qualitativo também é utilizado para comparar as vantagens dos métodos de Hellman e Oechslin com as vantagens do método proposto.

Para realizar esta comparação o mesmo conjunto de dados selecionado por Oechslin para medir o desempenho da *rainbow table* e compará-lo ao método de Hellman será utilizado, bem como a mesma condição de tamanho máximo que todas as tabelas podem ter. Foi comparada a quantidade média de operações de hash necessárias para recuperar um elemento. Para realizar esta comparação foram utilizados 2^{37} números inteiros e positivos sendo eles $\{1, 2, 3, ..., 2^{37} - 2, 2^{37} - 1, 2^{37}\}$, porém qualquer outra progressão aritmética poderia ter sido utilizada porque o algoritmo tem o mesmo comportamento para qualquer conjunto de entrada.

8 ALGORITMO PROPOSTO

Nesta sessão o funcionamento e as limitações do algoritmo proposto serão descritos em detalhes. Este algoritmo consiste em aplicar uma rodada de delta encoding e, em seguida, ordenar os elementos resultantes para, finalmente, remover todos os duplicados. Através desta técnica cria-se um novo *trade-off* pois quanto mais elementos duplicados forem removidos, mais espaço será salvo. Por outro lado, mais processamento será necessário para recuperar o conjunto original. O conjunto ordenado resultante pode ainda sofrer outra rodada de delta encoding ou qualquer outra técnica de compressão escolhida pelo analista como bitmap (LEMIRE; KASER; AOUICHE, 2010).

8.1 Delta Encoding com Complexidade $O(n^2)$

Dado o conjunto $\alpha = \{3, 12, 20, 21, 30\}$ e sendo n o número de elementos do conjunto, o resultado de $\alpha_d = D(\alpha)$ é o conjunto $\{3, 9, 8, 1, 9\}$. Como explicado anteriormente, é possível remontar o conjunto α com complexidade $O(n)$. Entretanto, ordenando o conjunto α_d de forma crescente através de uma função qualquer $S(x)$ obtém-se $\alpha_s = S(\alpha_d)$ tal que $\alpha_s = \{1, 3, 8, 9, 9\}$. Removendo os números duplicados de α_s obtém-se $\beta = \{1, 3, 8, 9\}$ que pode novamente ser submetido a função de delta encoding $D(\{1, 3, 8, 9\})$ criando o conjunto $\beta_d = \{1, 2, 5, 1\}$. O conjunto β_d deve começar com o primeiro elemento do conjunto α, resultando em $\beta_d = \{3, 1, 2, 5, 1\}$. O conjunto β_d pode ser salvo em apenas 15 bits, considerando que cada elemento ocupa 3 bits, enquanto que o conjunto original precisa de 25 bits, desconsiderando qualquer algoritmo de compressão.

Recuperar o conjunto β a partir de β_d pode ser feito com complexidade $O(n)$, ou seja, 4 operações (ignorando o primeiro elemento de β_d pois na verdade é o primeiro elemento de α), obtendo $\beta = \{1, 3, 8, 9\}$. Após adicionar o primeiro elemento de β_d a β obtém-se $\beta = \{3, 1, 3, 8, 9\}$. Recuperar o conjunto α consiste em um problema de maior complexidade. Uma maneira de recuperar o conjunto α seria selecionando o primeiro elemento do conjunto β, que é o primeiro elemento do conjunto α, percorrer o conjunto β e a cada iteração somar o primeiro elemento do conjunto α a cada um dos elementos do conjunto β. Desta forma em alguma das interações o segundo elemento do conjunto α será encontrado. Quando o próximo elemento do conjunto α for encontrado então a busca recomeça, somando todos os elementos do conjunto β ao ultimo elemento do conjunto α encontrado. Desta maneira, todos os elementos do conjunto α serão encontrados e o conjunto será completamente recuperado.

Contudo, é necessário que exista alguma forma de validar se algum elemento do conjunto α foi encontrado ou não. Evidentemente tal método não deve se basear nos elementos do conjunto original mas sim em alguma heurística.

Uma maneira de resolver este problema é através da criação de uma função $m(x)$ capaz de determinar se um elemento pertence ao conjunto α ou não. A Figura 5 detalha o funcionamento do algoritmo através de tal função $m(x)$, seguindo os preceitos do Teorema 1. Abaixo desse Teorema, está sua prova, confirmando que aplicado ao método descrito acima é possível recuperar o conjunto α integralmente.

Teorema 1 *Dado um conjunto α de elementos ordenados e aplicando-se as operações de redução $D()$, ordenação $S()$, remoção e redução $D()$, nessa ordem, é possível recuperar todo o conjunto original α se após a última operação o conjunto β_d gerado contiver, além dos seus elementos, o primeiro elemento do conjunto original α.*

Proposição 1 *Aplicando as funções de ordenação e remoção a um conjunto qualquer α_d, o conjunto resultante β mantém um representante de cada elemento do conjunto α_d além do primeiro elemento do conjunto α.*

Proposição 2 *Dado o primeiro elemento do conjunto β, os outros elementos deste conjunto são recuperáveis a partir de β_d de forma simples e óbvia.*

Prova.

1. Base: O elemento α_1 está no conjunto β.
2. Passo: Com base nas proposições 1 e 2, dado que α_n é conhecido, $\alpha_{n+1} = \alpha_n + k$, sendo $\alpha_n + k$ o menor elemento resultante de $m(\alpha_n + k) = verdadeiro$, para todo $k \in \beta$.
3. Restrição: O conjunto α é ordenado e não possui elementos duplicados.

Entretanto, criar uma função $m(x)$ capaz de determinar se um elemento qualquer faz parte do conjunto α ou não pode ser muito difícil. Outra forma de solucionar este problema é separando os elementos do conjunto original α em conjuntos menores através de algum método aleatório, como por exemplo uma função que mapeie cada elemento a algum subconjunto aleatoriamente. Tal função $m(x)$ deve sempre mapear um elemento x para um subconjunto α_a, nunca para outro.

Considerando que a função $m(x)$ mapeie os elementos do conjunto α para 2 subconjuntos $\alpha_a = \{3, 12, 21\}$ e $\alpha_b = \{20, 30\}$. Aplicando o algoritmo descrito anteriormente no subconjunto a obtém-se $\alpha_{ad} = \{3, 9, 9\}$, $\alpha_{as} = \{3, 9, 9\}$, $\beta_a = \{3, 9\}$ e finalmente $\beta_{ad} = \{3, 9\}$.

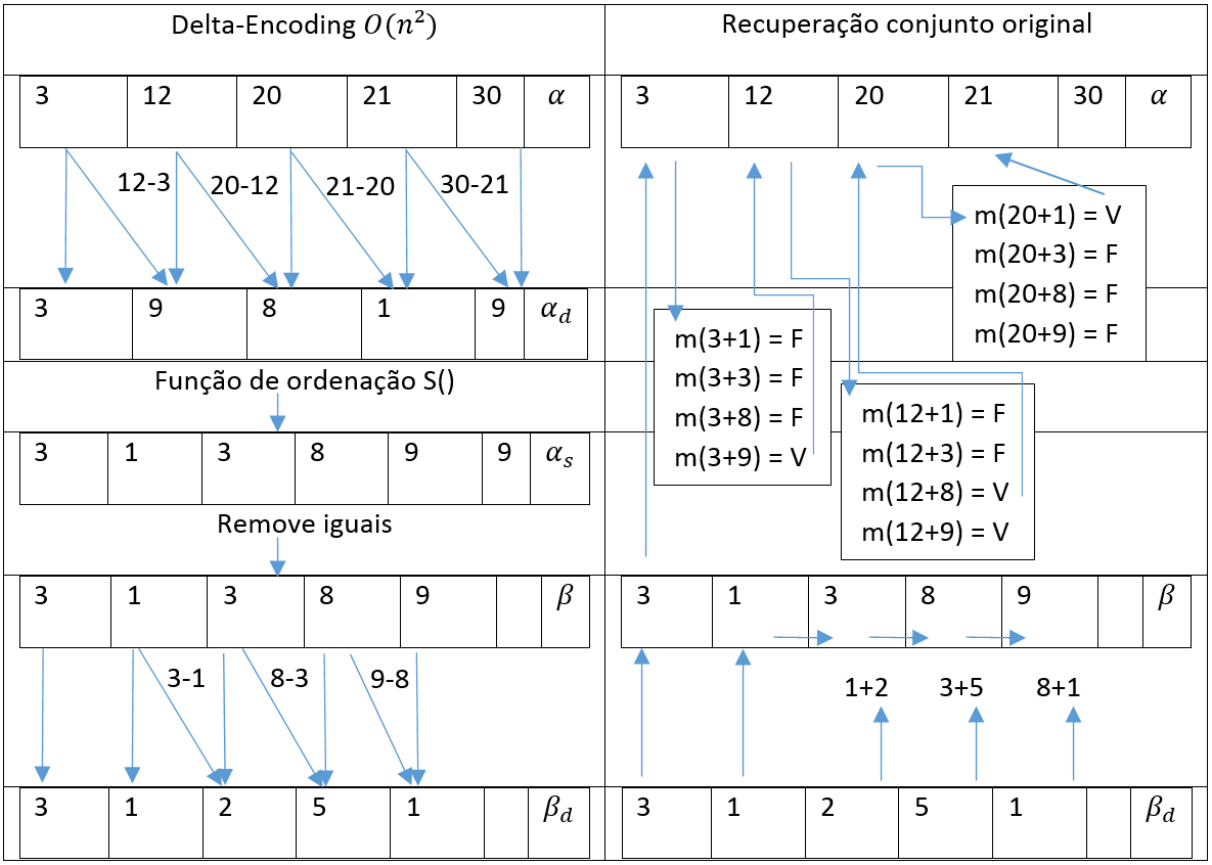
Como visto anteriormente recuperar β_a é trivial porém recuperar α_a exige alguma heurística definida na função $m(x)$. O primeiro elemento de β_a sempre será o primeiro elemento de α_a, soma-se então o ultimo elemento do conjunto original encontrado (que na primeira rodada é $\alpha[0]$) com o primeiro da lista, 3, obtendo 6. Aplica-se $m(6)$ e descobre-se que 6 não faz parte do conjunto α_a. Soma-se então o ultimo elemento do conjunto original com o segundo elemento do conjunto β_a, ou seja, 9, obtendo 12. Aplica-se então $m(12)$ que retornará o conjunto a, indicando que 12 pertence ao conjunto original α_a. Como um novo elemento do conjunto original foi encontrado, todos os elementos de β_a devem ser somados novamente com o último elemento do conjunto α_a encontrado: 12, para que o terceiro elemento do conjunto seja encontrado. Este procedimento deve ser repetido até que todos os elementos do conjunto β_a sejam usados diretamente para encontrar algum elemento do conjunto original.

Desconsiderando a complexidade da função $m(x)$ é evidente que a complexidade do algoritmo de recuperação dos subconjuntos criados a partir do resultado de $m(x)$ é de $O(n^2)$. A complexidade da geração dos subconjuntos é negligenciável porque são gerados apenas uma vez enquanto que o a recuperação é uma tarefa que tende a se repetir inúmeras vezes.

8.1.1 Considerações sobre o algoritmo Delta Encoding de complexidade $O(n^2)$

A parte mais importante do algoritmo de delta encoding de complexidade $O(n^2)$ está na função heurística que determina a distribuição dos elementos de um conjunto em subconjuntos menores. Esta função deve ser o mais próximo possível de uma função aleatória.

Figura 5 – Criação de β_d e recuperação de $\beta_d \rightarrow \alpha$



Fonte: Elaborado pelo autor.

Apenas conjuntos ordenados podem ser aplicados neste algoritmo. Outra limitação é que ele não é capaz de recuperar conjuntos com números duplicados pois durante a recuperação não é possível determinar quantas vezes um elemento existe no conjunto original, desta forma, sempre assume-se que não existam elementos duplicados.

É interessante adicionar um elemento extra no começo ou no fim de β_x e consequentemente de β_{dx} para indicar o total de elementos original do subconjunto α_x pois criar uma função $m(x)$ tal que conheça o total de elementos de α_x é inviável.

8.2 *Trade-off* criptoanalítico não probabilístico

Utilizando o algoritmo de delta encoding de complexidade $O(n^2)$ abordado na sessão anterior, é possível representar conjuntos numéricos com muitos elementos de uma forma compacta, ao custo de utilizar mais processamento. Entretanto, apenas o algoritmo proposto não pode ser considerado como um *trade-off* pois não é possível variar entre memória e processamento.

Uma forma de torná-lo um *trade-off* é alterar a função $m(x)$ de tal maneira que para um mesmo conjunto de elementos seja possível dividi-lo em subconjuntos de tamanhos diferentes. Desta maneira, muitos subconjuntos resultarão em subconjuntos com poucos elementos e consequentemente pouco processamento e muita memória necessária para armazena-los ($M = N$ e $T = 1$). Contudo, poucos subconjuntos resultarão em subconjuntos com muitos elementos, ou seja, muito processamento e pouca memória ($T = N$ e $M = 1$). Poucos subconjuntos resultarão em pouca memória pois ao aplicar o primeiro passo do algoritmo descrito anteriormente muitos deltas serão duplicados e serão posteriormente removidos enquanto que muitos subconjuntos com poucos elementos resultarão em subconjuntos esparsos e por isso haverão menos deltas duplicados, logo, o espaço total das tabelas será maior.

É possível associar os subconjuntos deste método com as múltiplas tabelas de Hellman.

Como o objetivo deste algoritmo é armazenar um conjunto de senhas e seus hashes, a ordem dos elementos é irrelevante. Deste modo tornando possível a utilização do delta encoding de complexidade $O(n^2)$.

Considere α como um conjunto de senhas a serem armazenadas. Este conjunto deve ser primeiramente ordenado e então submetido ao algoritmo de delta encoding de complexidade $O(n^2)$, a função $m(x)$ escolhida deve ser a função de hash $H(x)$ a ser invertida.

É importante reduzir a saída da função $m(x)$ através de uma função de redução qualquer $r(x)$ tal que produza uma quantidade m_{alpha} de subconjuntos que satisfaça os parâmetros do *trade-off* desejados. Por exemplo, uma função de hash que tenha sua saída fixa em 128 bits resultará em 2^{128} subconjuntos se sua saída não for reduzida. Caso a quantidade de elementos no conjunto α seja menor que 2^{128} então o *trade-off* seria de $M = N$ e $T = 1$.

Os subconjuntos de α devem ser armazenados em memória juntamente com a identificação de qual algoritmo de hash foi utilizado como função $m(x)$ e como a redução de sua saída deve ser reproduzida.

Após todos os subconjuntos serem gerados e armazenados em memória, utilizá-los para recuperar alguma senha fornecendo apenas o seu hash é trivial, considerando que ela esteja no conjunto α inicial. Para inverter um hash qualquer h basta primeiro reduzi-lo, encontrando o subconjunto onde o texto plano de h está armazenado. Depois de identificar o subconjunto basta aplicar o algoritmo de recuperação proposto na sessão anterior utilizando a função $r(H(x))$ como $m(x)$. Quando algum elemento qualquer i pertencente ao subconjunto $m_{r(h)}$ for encontrado basta comparar $H(i) = h$, caso for verdadeiro então i é o texto plano do hash h produzido pela função $H(x)$. Se o texto plano i fizer parte do conjunto α então é garantido que será recuperado.

O algoritmo descrito é exemplificado pela Tabela 4. A coluna Texto Plano representa o conjunto α a ser armazenado, a função $H(x)$ é o MD5 e a função de redução $r(x)$ reduz a saída de $H(x)$ para 4 bits, ou 1 caractere hexadecimal como representado na terceira coluna.

Tabela 4 – Exemplo de um conjunto α utilizando MD5 como $m(x)$

Texto Plano	$H(x)$	$r(H(x))$
1	**c4ca4238a0b923820dcc509a6f75849b**	**c**
2	**c81e728d9d4c2f636f067f89cc14862c**	**c**
3	eccbc87e4b5ce2fe28308fd9f2a7baf3	e
4	a87ff679a2f3e71d9181a67b7542122c	a
5	e4da3b7fbbce2345d7772b0674a318d5	e
6	1679091c5a880faf6fb5e6087eb1b2dc	1
7	8f14e45fceea167a5a36dedd4bea2543	8
8	**c9f0f895fb98ab9159f51fd0297e236d**	**c**
9	45c48cce2e2d7fbdea1afc51c7c6ad26	4
10	d3d9446802a44259755d38e6d163e820	d
11	6512bd43d9caa6e02c990b0a82652dca	6
12	**c20ad4d76fe97759aa27a0c99bff6710**	**c**

Fonte: Elaborado pelo autor.

Neste exemplo dos 12 elementos do conjunto α, 4 foram reduzidos para o endereço c: $1, 2, 8, 12$ que após serem submetidos ao algoritmo de delta encoding de complexidade $O(n^2)$ resultam em: $1, 4, 6$. O endereço c será composto pelos elementos $1, 4, 6$.

Como resultado o algoritmo gera uma tabela de consultas relacionando cada $r(H(x))$ com sua respectiva lista de elementos. No exemplo anterior da Tabela 4, $r(H(x)) = c$ está ligada a lista $1, 4, 6$. Essa relação ajuda a recuperar os textos planos 1, 2, 8 e 12. Partindo dessa tabela e de um hash y interceptado, um criptoanalista consegue recuperar o texto plano relacionado a y aplicando $r(y)$ e acessando a lista de valores codificados com $DE2$. Dessa lista extrai-se o texto plano original aplicando a função de hash $H(y)$ aos elementos 1, 2, 8 e 12 até que algum dos hashes seja igual ao hash y interceptado.

Como $m(x)$ é uma função de hash utilizada para separar o conjunto α em subconjuntos então os elementos devem obrigatoriamente fazer parte de uma progressão aritmética. Caso a distância entre os elementos não seja constante a fase de recuperação poderá erroneamente gerar

elementos que a função de hash irá considerar como parte do subconjunto sendo recuperando. Apesar desta limitação, funções de hash satisfazem as condições impostas sobre como a função $m(x)$ deve funcionar. Contudo, esta limitação não é problemática para este tipo de aplicação pois normalmente um atacante irá utilizá-la com grandes conjuntos de dados, frequentemente englobando todas as possibilidades, ou seja, tal conjunto seria uma progressão aritmética com intervalo de 1 bit entre cada elemento.

Considerando que anteriormente a segunda rodada de delta encoding os elementos são únicos e ordenados, outras técnicas de compressão podem ser utilizadas ao invés de uma rodada de delta encoding. Uma técnica simples e eficiente quando os elementos não são distantes entre si e nem repetidos é o bitmap (LEMIRE; KASER; AOUICHE, 2010). Em um bitmap os números são salvos nos seus respectivos bits, por exemplo a sequência $1, 4, 7, 8$ seria salva como 010010011, onde cada bit setado como 1 representa um número da sequência. Neste caso, o primeiro bit, que representa o número 0 está setado como 0, ou seja, o número 0 não faz parte da sequência. Entretanto, o segundo bit, que representa o número 1 está setado, ou seja, 1 faz parte da sequência. O mesmo vale para os bits seguinte, como apenas o quinto, oitavo e nono bit estão setados então somente estes números fazem parte da sequência. Utilizando esta técnica foram necessários apenas 9 bits para representar uma sequência de 4 elementos. Caso a sequência terminasse com um número muito grande, por exemplo, 1500 ao invés de 8 então seriam necessários 1501 bits para representá-la, neste caso bitmap claramente não é vantajoso.

Nesta implementação tanto o delta encoding como o bitmap foram utilizados após a rodada de delta encoding de complexidade $O(n^2)$. Quando poucos subconjuntos foram utilizados a técnica de bitmap se mostrou mais eficiente.

8.3 Análise Teórica

Como já explicado anteriormente, o algoritmo de delta encoding para recuperar α a partir de β proposto possui complexidade temporal $O(n^2)$. O pseudo-algoritmo 1 aprofunda, ainda mais, como a recuperação de α a partir de β funciona.

Conforme apontado por Oechslin, o número médio de iterações para percorrer uma *rainbow table* é de $t(t-1)/2$, enquanto que t^2 operações são necessárias para buscar em t tabelas de Hellman de tamanho m linhas x t colunas (correspondem a 1 *rainbow table*) (OECHSLIN, 2003), ou seja, ambos possuem complexidade temporal de $O(n^2)$.

Desconsiderando a complexidade da função $m(x)$ é evidente que a complexidade do pseudo-algorítmo 1 é de $O(n^2)$. A complexidade da função $m(x)$ é desconsiderada nessa análise pois pode ser implementada de diversas maneiras. Caso seja utilizada uma função de hash como implementado neste estudo, então a sua complexidade ($O(1)$) é desprezível (SABHARWAL; BRATIA, 1997).

Algorithm 1 Pseudo-algoritmo para recuperar $alpha$ a partir de $beta$

```
1:  procedure RECUPERA α(β)
2:      α[0] ← β[0]
3:      endereco ← m(α[0])
4:      i ← 1
5:      for all a in β do
6:          for all b in β do
7:              if endereco = m(a + b)  &  (a + b) > α[i − 1] then
8:                  α[i] ← a + b
9:                  i ← i + 1
10:                 break
11:             end if
12:         end for
13:     end for
14:     return α
15: end procedure
```

9 RESULTADOS EXPERIMENTAIS E COMPARAÇÃO

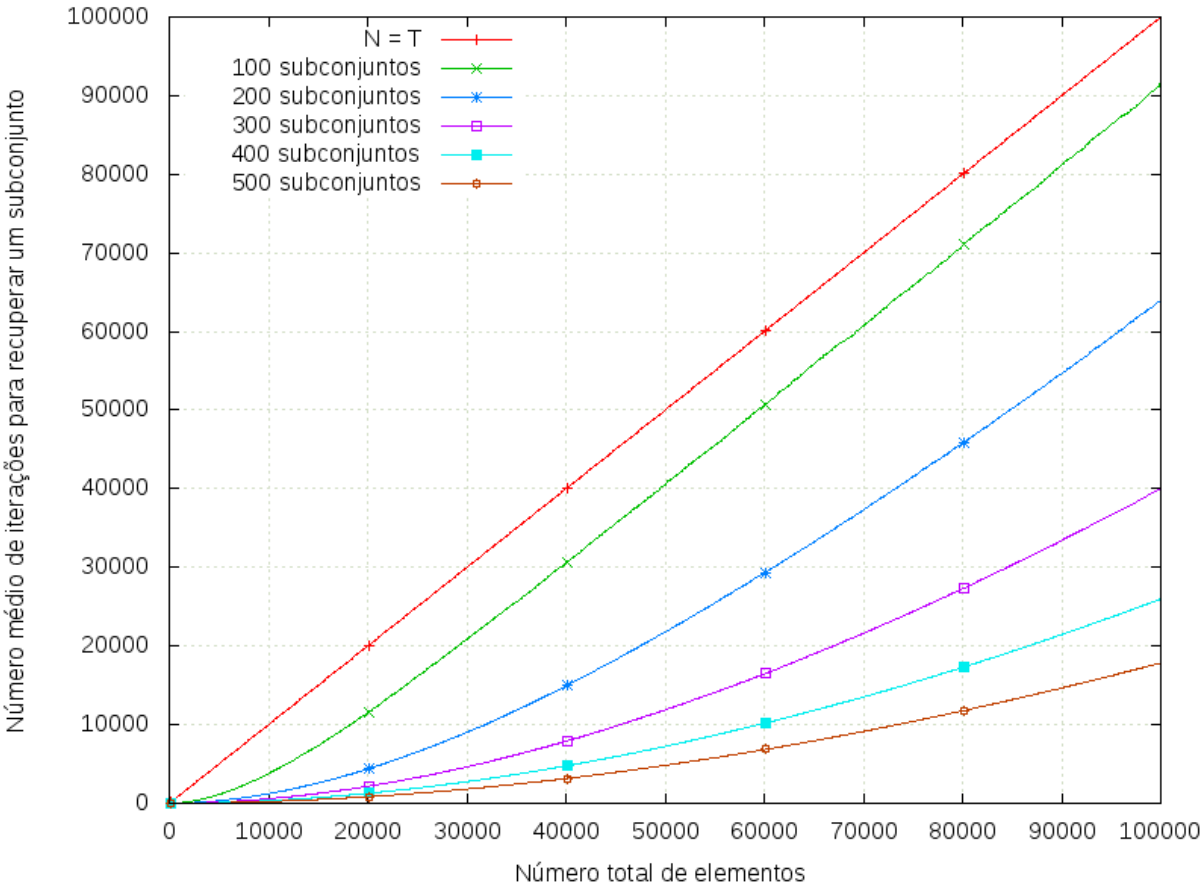
O objetivo da sessão de resultados experimentais é verificar a veracidade e a acurácia da análise teórica através da análise dos resultados da implementação do *trade-off* criptoanalítico não probabilístico. Nesta sessão o algoritmo proposto também será comparado, de forma prática, com o método de Hellman e o de Oechslin.

Para analisar o comportamento temporal e espacial do algoritmo, conjuntos numéricos variando entre 1000 e 100000 elementos foram gerados e aplicados ao *trade-off* criptoanalítico não probabilístico utilizando múltiplos subconjuntos (100, 200, 300, 400 e 500). Os gráficos das Figuras 6 e 7 tem como objetivo demonstrar o comportamento temporal e espacial do algoritmo, respectivamente. Os dois gráficos representam os mesmos conjuntos de dados e os mesmos subconjuntos. O eixo X dos dois gráficos representa o número total de elementos enquanto que as linhas representam cada subconjunto. As linhas $N = T$ e $N = M$, entretanto, não representam nenhum subconjunto mas sim os extremos do *trade-off* apenas para fins informativos. O eixo Y do gráfico da Figura 6 representa o número médio de iterações para recuperar um subconjunto alpha enquanto que o eixo Y do gráfico da Figura 7 representa o número médio de deltas salvos em disco de todos os subconjuntos.

No gráfico da Figura 6 é possível observar como dividindo os elementos em muitos subconjuntos (300, 400 e 500) reduz-se drasticamente o número de operações de hash necessárias (eixo Y) para recuperar os elementos de um subconjunto. Quanto menos subconjuntos forem utilizados, ou seja, quanto mais elementos existirem em um subconjunto, mais operações de hash serão necessárias para recuperá-lo, aproximando o *trade-off* a $N = T$.

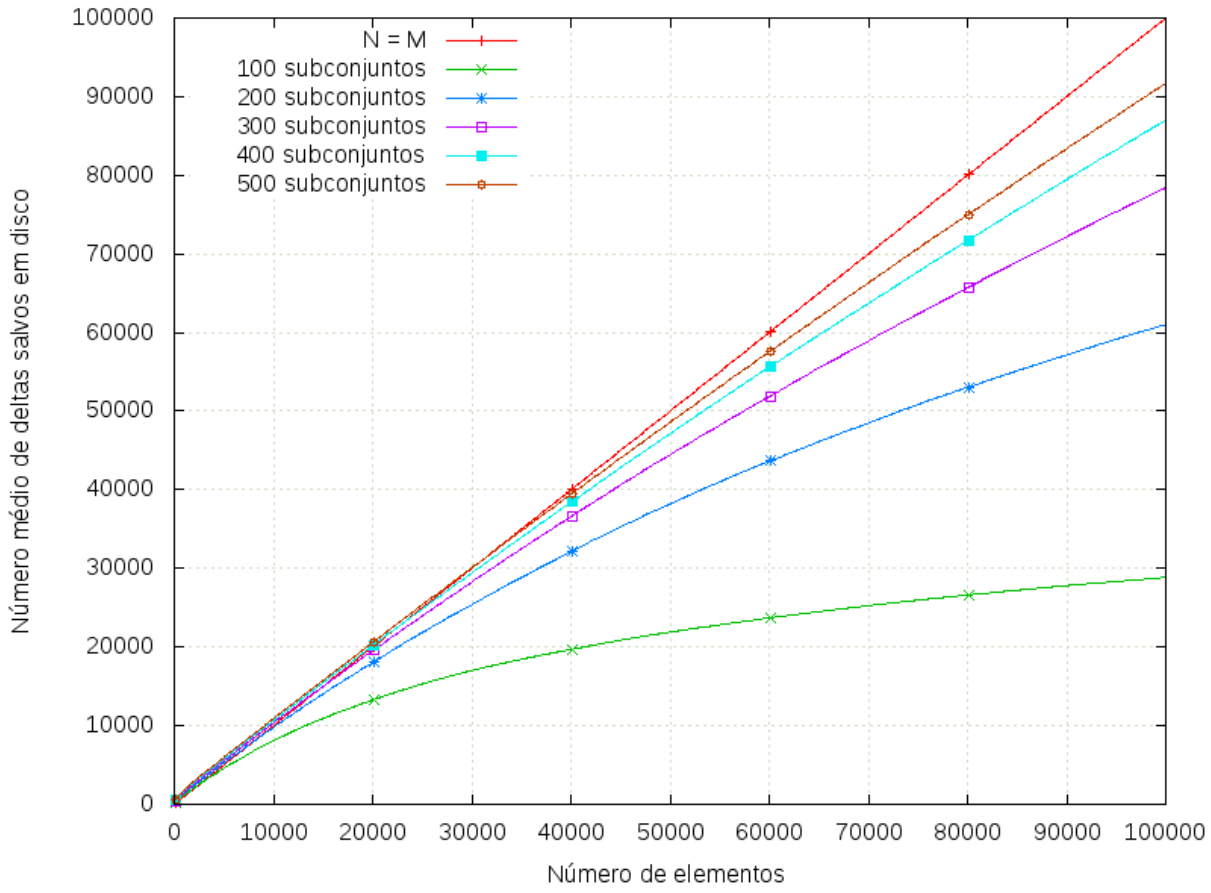
Em contrapartida, quanto menos subconjuntos forem utilizados, maior será o número de elementos destinados a cada subconjunto pela função $m(x)$. Consequentemente, a primeira rodada de delta encoding resultará em mais elementos duplicados do que se fossem utilizados

Figura 6 – Número médio de iterações para recuperar um subconjunto



Fonte: Elaborado pelo autor.

Figura 7 – Número médio de deltas salvos em disco



Fonte: Elaborado pelo autor.

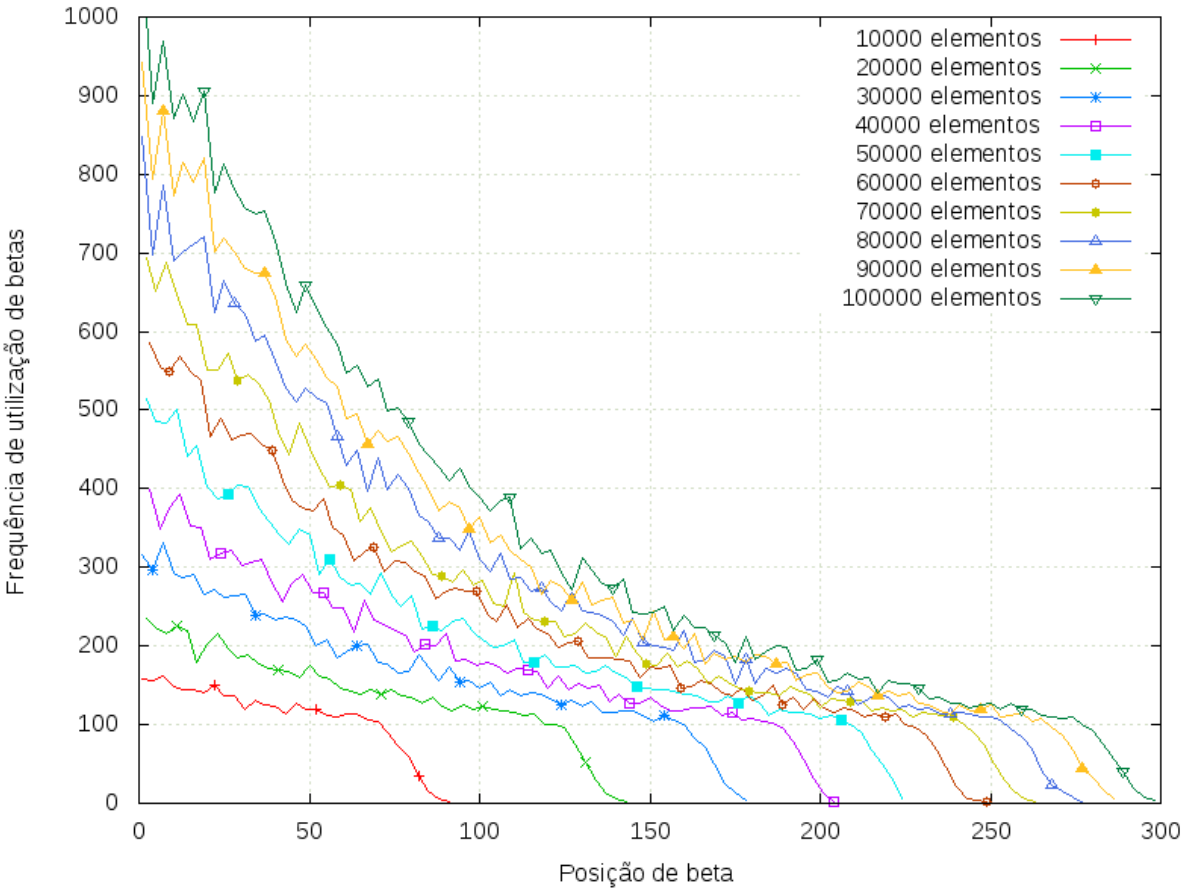
mais subconjuntos (e consequentemente menos elementos por subconjunto). Como os deltas duplicados serão posteriormente removidos, menos espaço em disco será necessário para salvar todos os subconjuntos, o que se comprova no gráfico da Figura 7. Todavia, se muitos subconjuntos forem utilizados, poucos deltas duplicados serão gerados, fazendo com que mais deltas sejam gravados no disco, aproximando o *trade-off* a $N = M$.

O gráfico da Figura 8 consiste em um histograma dos conjuntos anteriores aplicados a 100 subconjuntos onde cada linha representa um conjunto de dados diferente. Neste gráfico o eixo X representa as posições dos subconjuntos de betas e o eixo Y o número de vezes que cada posição do eixo X foi utilizada. Este gráfico foi criado para determinar a frequência de cada delta efetivamente utilizado para recuperar os elementos de alpha.

Como grande parte dos elementos originais foram encontrados nas primeiras posições do subconjunto, o segundo laço dificilmente passará das posições iniciais, o que o torna praticamente inexistente, isso significa que na pratica existe apenas um laço para recuperar os elementos. Isto é, para um conjunto N são necessários em média N operações para recuperar um subconjunto.

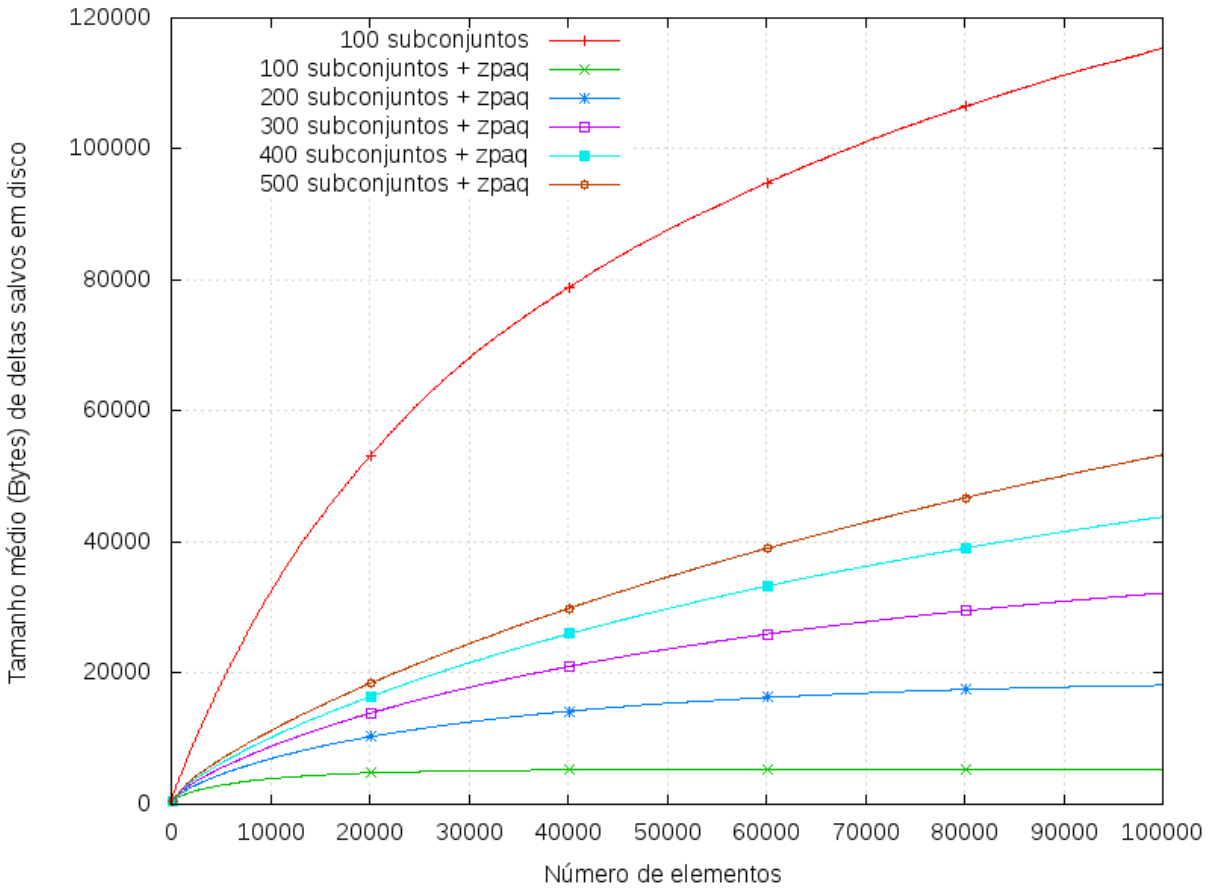
É possível, ainda, diminuir o tamanho dos deltas salvos em memória através de algum al-

Figura 8 – Histograma de utilização de $beta$



Fonte: Elaborado pelo autor.

Figura 9 – Tamanho médio dos subconjuntos após serem comprimidos com ZPAQ



Fonte: Elaborado pelo autor.

goritmo de compressão de dados. O algoritmo ZPAQ (MAHONEY, 2013) foi capaz de reduzir o tamanho dos subconjuntos de modo que os deltas comprimidos ocupem menos espaço que quando dispostos em 100 subconjuntos sem compressão conforme mostrado no gráfico da Figura 9.

Oechslin criou 23330 tabelas de Hellman com dimensões de $m = 7501$ linhas e $t = 4666$ colunas e 5 *rainbow tables* com dimensões $t_r = 4666$ linhas e $m = 35000000$ colunas ambas usando o mesmo conjunto N de 2^{37} hashes diferentes de 8 bytes cada. A probabilidade de encontrar algum hash deste conjunto foi estimada em 99.9% tanto na *rainbow table* quanto nas tabelas de Hellman. Oechslin recuperou 500 elementos aleatórios na *rainbow table* e constatou que foram necessários em média 7.4 milhões de operações de hash contra 90.3 milhões usando as tabelas originais para recuperar cada elemento. Os dois conjuntos de tabelas foram projetados para ocuparem 1.4 GB cada.

Seguindo o critério de utilizar no máximo 1.4 GB, tabelas do algoritmo proposto foram geradas para cobrir o mesmo espaço de 2^{37} hashes diferentes de 8 bytes cada. Foram criados 27 mil subconjuntos e 500 elementos foram recuperados aleatoriamente, utilizando, em média, 4 bilhões de operações de hash para recuperar cada elemento. Foi observado que o número

de operações de hash para recuperar cada elemento sempre foi igual a 99.57% do número de operações de hash utilizando a técnica de força bruta. Caso a compactação esteja disponível, como discutido anteriormente, o número de subconjuntos necessários para ocupar 1.4 GB seria de 60 mil subconjuntos. Nesse cenário cada um dos 500 números aleatórios necessitaria, em média, $3,5$ bilhões de operações de hash, sendo que recuperar cada elemento equivaleria a 97.32% do número de operações de hash utilizando a técnica de força bruta.

Ao contrário dos métodos probabilísticos onde os elementos são distribuidos aleatóriamente entre todas as linhas de todas as tabelas, o algoritmo proposto distribui os elementos aleatóriamente entre as tabelas de forma progressiva e constante, começando do primeiro elemento até o último. Resultando que cada tabela é composta por elementos aleatórios mas dispostos em ordem crescente, consequentemente recuperar um elemento vai depender da sua posição na tabela. Caso um elemento esteja no começo de uma tabela, poucas operações de hash serão necessárias para recuperá-lo, mas se ele estiver no final da tabela muitas operações de hash serão necessárias. Desta forma, o que se mantém constante entre o número de operações de hash necessárias para recuperar qualquer elemento de uma tabela é a sua relação com o número de operações de hash necessárias para recuperá-lo utilizando a técnica de força bruta.

No cenário de 27 mil tabelas, o número de elementos salvos em cada subconjunto equivale, em média, a 3.09% do total de elementos antes da aplicação do algoritmo de delta encoding de complexidade $O(n^2)$, enquanto que no cenário de 60 mil tabelas equivalem em média a 11.03%. Logo, com menos elementos duplicados, menos operações de hash foram necessárias para recuperar cada elemento.

Em ambos os casos de 27 e 60 mil tabelas a última operação de delta encoding foi substituída por bitmap pois, em média, todos os conjuntos possuíam elementos próximos o bastante e em quantidade suficiente para a técnica ser mais vantajosa que a segunda rodada de delta encoding. Apenas utilizando mais de cerca de 550 mil subconjuntos o uso do segundo delta encoding se mostrou mais eficiente que o bitmap.

A Tabela 5 sumariza esta comparação, mostrando que apesar de o número médio de operações de hash necessárias ser muito superior aos métodos de Hellman e de Oechslin para recuperar um elemento, o algoritmo proposto necessita fazer poucas operações de leitura em disco para recuperar um elemento pois apenas o subconjunto a qual ele pertence precisa ser carregado do disco para a memória RAM no início da execução. Enquanto isso, os métodos de Hellman e de Oechslin precisam ou carregar todas as tabelas de uma só vez ou fazer buscas aleatórias em disco pois ao aplicar a função de redução ao hash sendo recuperado, é necessário buscar em todas as linhas de todas as tabelas pelo resultado desta redução. Dependendo da arquitetura em que os ataques são conduzidos e das configurações escolhidas para o *trade-off*, o tempo médio de execução dos métodos probabilísticos citados podem ser maiores que o método proposto.

Tabela 5 – Comparação entre o algoritmo proposto com os métodos de Hellman e de Oechslin

	Hellman	***Rainbow Table***	**Alg. Proposto**	**Alg. Proposto com Compressão**
Informações	2^{37}	2^{37}	2^{37}	2^{37}
Tamanho	1.4 GB	1.4 GB	1.4 GB	1.4 GB
Operações de hash	90.3 M	7.4 M	4 B	3.5 B
Vantagens	• Poucas operações de hash		• Pouco acesso ao disco • Acesso ao disco é serial • Sucesso é garantido • Facilmente paralelizável	

Fonte: Elaborado pelo autor.

10 CONCLUSÃO

Funções de hash são amplamente utilizadas para armazenar senhas de forma segura, porém a simples aplicação de uma função de hash pode não ser suficiente para protegê-las. Diferentes ataques podem ser aplicados em hashes, tais como força bruta onde senhas são geradas, seus hashes calculados e comparados com os hashes sendo quebrados e caso algum seja igual, a senha que originou tal hash é retornada. Outra possibilidade é pré-computar um intervalo de senhas e salvá-las em uma estrutura chamada dicionário, juntamente com seus hashes em disco para então buscar neste banco de dados os hashes a serem quebrados. No primeiro ataque, nada é salvo e todos os hashes são re-calculados cada vez que o ataque é executado, enquanto que no segundo ataque o intervalo de senhas é calculado apenas uma vez e é integralmente salvo em disco, não sendo necessário executar qualquer operação de hash quando o ataque for aplicado. O ataque de força bruta representa um extremo do *trade-off*: $M = 1$ e $N = T$ enquanto que o ataque de dicionário o outro: $M = T$ e $N = 1$.

Hellman propôs um método probabilístico capaz de recuperar N elementos em $N^{2/3}$ operações utilizando $N^{2/3}$ palavras de memória. Oechslin alterou o método de Hellman de forma a utilizar uma função de redução diferente para cada coluna da tabela, desta forma obteve resultados práticos de cerca de 12 vezes menos operações de hash para recuperar um elemento em comparação com o método de Hellman.

Diferentemente dos métodos de Hellman e de Oechslin o algoritmo proposto neste trabalho não é probabilístico, garantindo que qualquer elemento do intervalo utilizado será encontrado. Este algoritmo utiliza a técnica de delta encoding de complexidade $O(n^2)$, também proposta e analisada neste trabalho. Apesar de uma segunda rodada de delta encoding ter sido utilizada para reduzir ainda mais o tamanho de cada subconjunto, outras técnicas podem ser utilizadas tais como bitmap.

Para comparar o desempenho do algoritmo proposto com os métodos de Hellman e Oechslin ele foi aplicado ao mesmo conjunto de dados 2^{37} utilizando uma função de hash que produz uma saída de 8 bytes. A limitação de tamanho de 1.4 GB também foi considerada de modo que quando dividido em 27 mil subconjuntos foi capaz de recuperar 500 elementos utilizando em média 4 bilhões de operações de hash para cada elemento. Caso compactação esteja disponível, com 60 mil subconjuntos $3,5$ bilhões de operações de hash foram necessárias, em média, para recuperar cada um dos 500 elementos.

Apesar do número médio de operações de hash necessárias para recuperar cada elemento do algoritmo proposto ser muito maior que os resultados de Hellman e Ochslin, o método proposto necessita consultar em disco apenas o subconjunto ao qual o hash sendo buscado pertence enquanto que os métodos de Hellman e Oechslin necessitam fazer leituras aleatórias as suas tabelas. Na prática, a grande quantidade de operações de hash exigida pelo método proposto é, de certa forma, compensada pela baixa quantidade de acessos a memória. O ganho obtido com o baixo acesso a memória depende muito do tipo de hardware onde o algoritmo é executado, portanto trabalhos futuros são necessários para avaliar se na prática o algoritmo proposto é mais eficiente em termos de tempo de execução que os atuais e em quais arquiteturas de hardware ele pode obter vantagens de velocidade. Concluindo, o algoritmo proposto possui vantagens em relação ao atual, mas também possui desvantagens que podem ser contornadas com futuras melhorias, demonstrando que ele tem potencial de uso se aprimorado.

PROPOSAL OF A CRYPTANALYTIC NON-PROBABILISTIC TRADE-OFF

Abstract: This work proposes a new cryptanalytic non-probabilistic trade-off. It presents the main cryptanalytic trade-offs, making a comparison with the proposed method. Although the number of hash operations to recover an element is high in comparison with the traditional methods, the new method has the advantage of guaranteed success on the recovery of hashes and the minimal and sequential disk read operations, unlike the existing probabilistic trade-offs.

Keywords: Trade-off. Hash. Cryptanalysis

REFERÊNCIAS

ÅGREN, M.; JOHANSSON, T.; HELL, M. Improving the rainbow attack by reusing colours. In: **Cryptology and network security**. [S.l.]: Springer, 2009. p. 362–378.

BELLARE, M.; CANETTI, R.; KRAWCZYK, H. Keying hash functions for message authentication. In: ADVANCES IN CRYPTOLOGY—CRYPTO'96, 1996. **Anais...** [S.l.: s.n.], 1996. p. 1–15.

BLACK, J. et al. Umac: fast and secure message authentication. In: ADVANCES IN CRYPTOLOGY—CRYPTO'99, 1999. **Anais...** [S.l.: s.n.], 1999. p. 216–233.

BURNS, R. C.; LONG, D. D. Efficient distributed backup with delta compression. In: I/O IN PARALLEL AND DISTRIBUTED SYSTEMS, 1997. **Proceedings...** [S.l.: s.n.], 1997. p. 27–36.

DIFFIE, W.; HELLMAN, M. E. New directions in cryptography. **Information Theory, IEEE Transactions on**, [S.l.], v. 22, n. 6, p. 644–654, 1976.

FREDMAN, M. L.; KOMLÓS, J.; SZEMERÉDI, E. Storing a sparse table with 0 (1) worst case access time. **Journal of the ACM (JACM)**, [S.l.], v. 31, n. 3, p. 538–544, 1984.

GAURAVARAM, P. Security analysis of salt|| password hashes. In: ADVANCED COMPUTER SCIENCE APPLICATIONS AND TECHNOLOGIES (ACSAT), 2012 INTERNATIONAL CONFERENCE ON, 2012. **Anais...** [S.l.: s.n.], 2012. p. 25–30.

HELLMAN, M. E. A cryptanalytic time-memory trade-off. **Information Theory, IEEE Transactions on**, [S.l.], v. 26, n. 4, p. 401–406, 1980.

IMPAGLIAZZO, R.; LEVIN, L. A.; LUBY, M. Pseudo-random generation from one-way functions. In: ACM SYMPOSIUM ON THEORY OF COMPUTING, 1989. **Proceedings...** [S.l.: s.n.], 1989. p. 12–24.

KIM, B.-I.; HONG, J. Analysis of the non-perfect table fuzzy rainbow tradeoff. In: INFORMATION SECURITY AND PRIVACY, 2013. **Anais...** [S.l.: s.n.], 2013. p. 347–362.

KRAWCZYK, H.; CANETTI, R.; BELLARE, M. **Hmac**: keyed-hashing for message authentication. [S.l.: s.n.], 1997. RFC.

LEMIRE, D.; KASER, O.; AOUICHE, K. Sorting improves word-aligned bitmap indexes. **Data & Knowledge Engineering**, [S.l.], v. 69, n. 1, p. 3–28, 2010.

MAHONEY, M. **The zpaq open standard format for highly compressed data - level 2**. 2013.

MOGUL, J. C. et al. Potential benefits of delta encoding and data compression for http. In: ACM SIGCOMM COMPUTER COMMUNICATION REVIEW, 1997. **Anais...** [S.l.: s.n.], 1997. v. 27, n. 4, p. 181–194.

NARAYANAN, A.; SHMATIKOV, V. Fast dictionary attacks on passwords using time-space tradeoff. In: ACM CONFERENCE ON COMPUTER AND COMMUNICATIONS SECURITY, 12., 2005. **Proceedings...** [S.l.: s.n.], 2005. p. 364–372.

OECHSLIN, P. Making a faster cryptanalytic time-memory trade-off. In: **Advances in cryptology-crypto 2003**. [S.l.]: Springer, 2003. p. 617–630.

PARK, J. S.; CHEN, M.-S.; YU, P. S. **An effective hash-based algorithm for mining association rules**. [S.l.]: ACM, 1995. v. 24, n. 2.

PAULO, J.; PEREIRA, J. A survey and classification of storage deduplication systems. **ACM Computing Surveys (CSUR)**, [S.l.], v. 47, n. 1, p. 11, 2014.

PSOUNIS, K. Class-based delta-encoding: a scalable scheme for caching dynamic web content. In: DISTRIBUTED COMPUTING SYSTEMS WORKSHOPS, 2002. PROCEEDINGS. 22ND INTERNATIONAL CONFERENCE ON, 2002. **Anais...** [S.l.: s.n.], 2002. p. 799–805.

PUB, N. F. 180-2. **Secure Hash Standard, National Institute of Standards and Technology, US Department of Commerce, DRAFT**, [S.l.], 2004.

PUGH, W.; HENZINGER, M. H. **Detecting duplicate and near-duplicate files**. US Patent 6,658,423.

RIJMEN, V.; OSWALD, E. Update on sha-1. In: **Topics in cryptology–ct-rsa 2005**. [S.l.]: Springer, 2005. p. 58–71.

RIVEST, R. **The md5 message-digest algorithm**. [S.l.]: RFC 1321, April, 1992.

SABHARWAL, C. L.; BRATIA, S. K. Image databases and near-perfect hash table. **Pattern Recognition**, [S.l.], v. 30, n. 11, p. 1867–1876, 1997.

SARAN, N.; DOGANAKSOY, A. Choosing parameters to achieve a higher success rate for hellman time memory trade off attack. In: AVAILABILITY, RELIABILITY AND SECURITY, 2009. ARES'09. INTERNATIONAL CONFERENCE ON, 2009. **Anais...** [S.l.: s.n.], 2009. p. 504–509.

SASAKI, Y. et al. New message difference for md4. In: FAST SOFTWARE ENCRYPTION, 2007. **Anais...** [S.l.: s.n.], 2007. p. 329–348.

SLEATOR, D. D.; TARJAN, R. E. Self-adjusting binary search trees. **Journal of the ACM (JACM)**, [S.l.], v. 32, n. 3, p. 652–686, 1985.

STALINGS, W. **Cryptography and network security**. [S.l.]: Upper Saddle River, NJ: Prentice Hall, 2005.

STALLINGS, W. **Network security essentials**: applications and standards. [S.l.]: Pearson Education India, 2007.

STANDARD, D. E. Fips pub 46. **Appendix A, Federal Information Processing Standards Publication**, [S.l.], 1977.

STANDARD, S. H. Department of commerce. **NIST, National Technical Information Service, Springfield, VA, USA**, [S.l.], 1995.

SUEL, T.; MEMON, N. **Algorithms for delta compression and remote file synchronization**. [S.l.]: Lossless Compression Handbook. Academic Press, 2002.

SUEL, Z. O. N. M. T.; TRENDAFILOV, D. Cluster-based delta compression of a collection of files. In: INTERNATIONAL CONFERENCE ON WEB INFORMATION SYSTEMS ENGINEERING, 2002. **Proceedings...** [S.l.: s.n.], 2002. p. 257.

WANG, X.; YIN, Y. L.; YU, H. Finding collisions in the full sha-1. In: ADVANCES IN CRYPTOLOGY–CRYPTO 2005, 2005. **Anais...** [S.l.: s.n.], 2005. p. 17–36.

WANG, X.; YU, H. How to break md5 and other hash functions. In: **Advances in cryptology–eurocrypt 2005**. [S.l.]: Springer, 2005. p. 19–35.

YING, H.-M.; THING, V. L. A novel rainbow table sorting method. In: CYBERLAWS 2011, THE SECOND INTERNATIONAL CONFERENCE ON TECHNICAL AND LEGAL ASPECTS OF THE E-SOCIETY, 2011. **Anais...** [S.l.: s.n.], 2011. p. 35–40.

ZHANG, W. et al. A new time-memory-resource trade-off method for password recovery. In: COMMUNICATIONS AND INTELLIGENCE INFORMATION SECURITY (ICCIIS), 2010 INTERNATIONAL CONFERENCE ON, 2010. **Anais...** [S.l.: s.n.], 2010. p. 75–79.