Les mots de passe Windows à la merci des compromis temps-mémoire

Philippe Oechslin

Laboratoire de Sécurité et de Cryptographie

Ecole Polytechnique Fédérale de Lausanne

philippe.oechslin@epfl.ch

LASEC

Plan de la présentation

- Les compromis temps-mémoire classiques
- La variante développée au Lasec
- Tables « parfaites »
- Configuration optimale
- Performances
- Exemples, démo

Introduction

- Les compromis temps-mémoire s'appliquent à des "fixed plaintext attacks"
  - On a un texte chiffré (ou p.ex. le hash d'un mot de passe)
  - On sait comment le chiffré à été généré
  - On cherche la clé qui a servi au chiffrement
- Attaque par force brute:
  - Nécessite aucune mémoire, mais très lent ($T=N$)
- Attaque par dictionnaire complet:
  - Attaque instantanée, nécessite énormément de mémoire ($M=N$)
- Compromis temps-mémoire
  - Nécessite un peu de mémoire et un peu de temps ($T=M=N^{2/3}$)
  - Temps de préparation N, taux de réussite < 100%

Exemple: les mots de passe Windows

- On stocke les mots de passe sous forme de hash
- Le hash est irréversible $\mathbf{2} \xrightarrow{H} \mathbf{h2}$
- LMHash:
  - On découpe le mot de passe en deux blocs de 7 caractères.
  - On transforme les minuscules en majuscules
  - Pour chaque bloc, un hash de 64 bits est créé en chiffrant un texte prédéfini (KGS!@#$%) avec DES en utilisant le mot de passe de 56 bits comme clé
  - Il y a 80 millards (2^{36}) de hash de mots de passes alphanumériques
    - alors qu'il y a 2^{83} mots de passe alphanumériques!
- NTHash: on calcule le hash MD4 du mot de passe

Compromis temps-mémoire

- Cette méthode à été inventée par Martin Hellman en 1980
- On génère tous les hash, on les organise en chaînes.
- On crée des tables dans lesquelles on ne stocke que le début et la fin de chaque chaîne (gains en mémoire)
- A partir d'un hash quelconque on peut retrouver une fin de chaîne
- A partir du début de chaîne on retrouve le mot de passe

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

Création de chaînes

- On définit un *fonction de réduction R* quelconque qui génère un mot de passe à partir d'un hash

$$0 \xrightarrow{H} h0 \qquad\qquad h0 \xrightarrow{R} 2$$

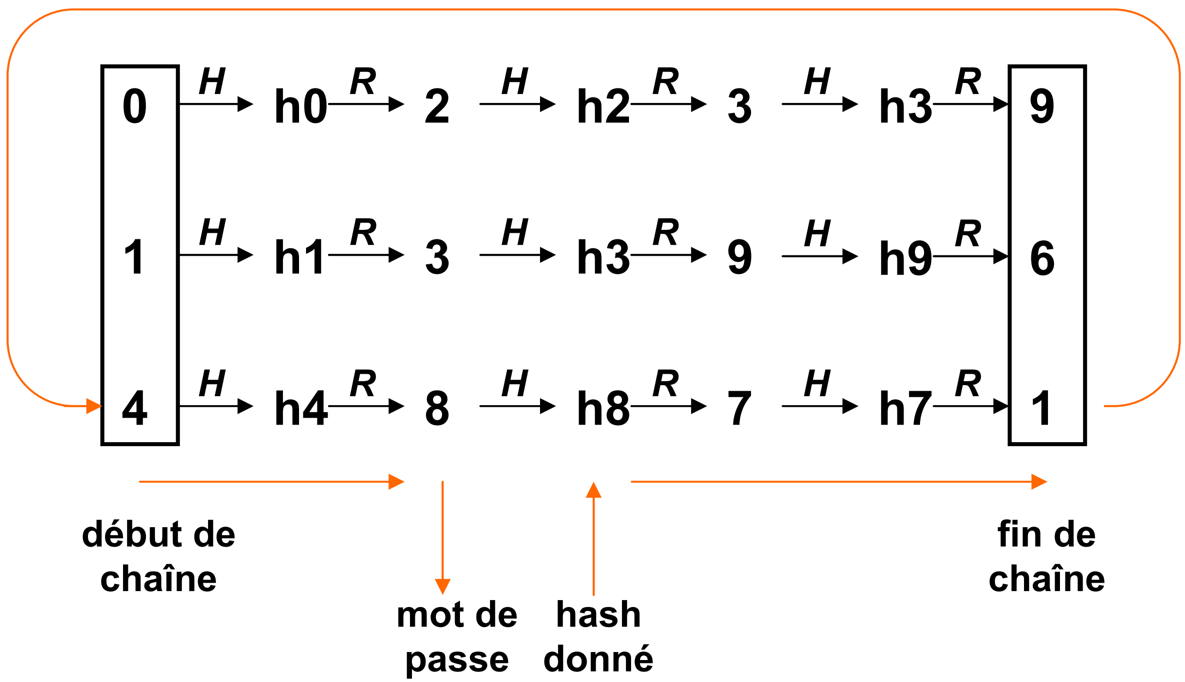
- Grâce à la fonction de réduction, on peut organiser les mots de passe les hashs en chaînes (longueur *t*):

$$0 \xrightarrow{H} h0 \xrightarrow{R} 2 \xrightarrow{H} h2 \xrightarrow{R} 3 \xrightarrow{H} h3 \xrightarrow{R} 9$$

- On génère une quantité de chaînes (*m*) et on ne stocke que le début et la fin dans une table

$$\begin{array}{c}
\boxed{0} \xrightarrow{H} h0 \xrightarrow{R} 2 \xrightarrow{H} h2 \xrightarrow{R} 3 \xrightarrow{H} h3 \xrightarrow{R} \boxed{9} \\
\boxed{1} \xrightarrow{H} h1 \xrightarrow{R} 3 \xrightarrow{H} h3 \xrightarrow{R} 9 \xrightarrow{H} h9 \xrightarrow{R} \boxed{6} \\
\boxed{4} \xrightarrow{H} h4 \xrightarrow{R} 8 \xrightarrow{H} h8 \xrightarrow{R} 7 \xrightarrow{H} h7 \xrightarrow{R} \boxed{1}
\end{array}$$

Recherche dans une table



- Pour retrouver un mot de passe à partir d'un hash, on génère une chaîne jusqu'à ce qu'on trouve une fin de chaîne connue. On recrée la chaîne originale à partir du début

Les fusions

$$0 \xrightarrow{H} h0 \xrightarrow{R} 2 \xrightarrow{H} h2 \xrightarrow{R} 3 \xrightarrow{H} h3 \xrightarrow{R} 9$$

$$1 \xrightarrow{H} h1 \xrightarrow{R} 3 \xrightarrow{H} h3 \xrightarrow{R} 9 \xrightarrow{H} h9 \xrightarrow{R} 6$$

$$4 \xrightarrow{H} h4 \xrightarrow{R} 8 \xrightarrow{H} h8 \xrightarrow{R} 7 \xrightarrow{H} h7 \xrightarrow{R} 1$$

- La fonction de réduction peut générer la même clé à partir de 2 hashs différents -> **fusion**
- On peut trouver une fin de chaîne dans la table sans y trouver le mot de passe recherché -> **fausse alarme**

Tables multiples

- Plus une table est grande, plus grand est la probabilité qu'une chaîne additionnelle fusionne avec une chaîne existante
  - L'efficacité des chaînes additionnelles diminue

- On peut construire plusieurs tables (ℓ) avec des fonctions de réductions différentes
  - Les chaînes de différentes tables peuvent avoir des collisions mais pas fusionner.

- Cette manière de faire est la méthode originale de Hellman

Plus efficaces: les tables rainbow

- Pour éviter des fusions on utilise une fonction de réduction différente pour chaque étape

$$\boxed{\begin{array}{c}8\\2\\4\end{array}}$$

$$8 \xrightarrow{H} h8 \xrightarrow{R_1} 6 \xrightarrow{H} h6 \xrightarrow{R_2} 3 \xrightarrow{H} h3 \xrightarrow{R_3} 0$$

$$2 \xrightarrow{H} h2 \xrightarrow{R_1} 3 \xrightarrow{H} h3 \xrightarrow{R_2} 5 \xrightarrow{H} h5 \xrightarrow{R_3} 9$$

$$4 \xrightarrow{H} h4 \xrightarrow{R_1} 8 \xrightarrow{H} h8 \xrightarrow{R_2} 1 \xrightarrow{H} h1 \xrightarrow{R_3} 7$$

$$\boxed{\begin{array}{c}0\\9\\7\end{array}}$$

- Deux chaînes rainbow ne peuvent fusionner que si elles ont une collision à la même position
  - Les autres collisions ne provoquent pas de fusion

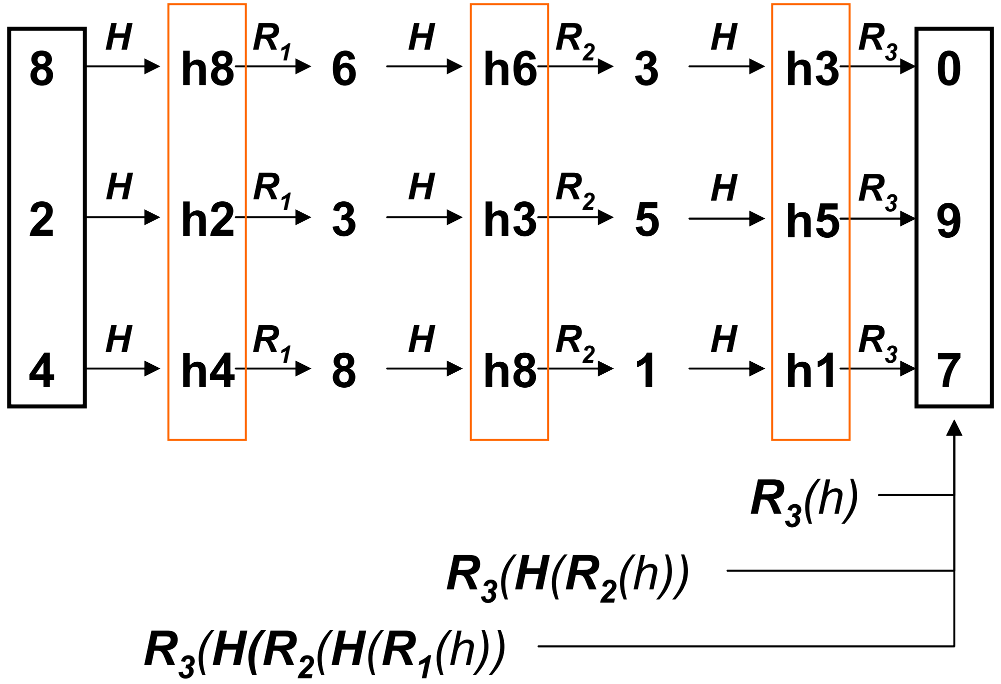
Comparaison de la probabilité de succès

$$\begin{array}{cc} & \overrightarrow{\quad t \quad} \\ m \downarrow & \begin{bmatrix} k^1_{1,1} \xrightarrow{f_1} \xrightarrow{f_1} \cdots \xrightarrow{f_1} k^1_{1,t} \\ \\ k^1_{m,1} \xrightarrow{f_1} \xrightarrow{f_1} \cdots \xrightarrow{f_1} k^1_{m,t} \end{bmatrix} \\ & \overrightarrow{\quad t \quad} \\ m \downarrow & \begin{bmatrix} k^2_{1,1} \xrightarrow{f_2} \xrightarrow{f_2} \cdots \xrightarrow{f_2} k^2_{1,t} \\ \\ k^2_{m,1} \xrightarrow{f_2} \xrightarrow{f_2} \cdots \xrightarrow{f_2} k^2_{m,t} \end{bmatrix} \\ & \vdots \qquad\qquad \vdots \\ & \overrightarrow{\quad t \quad} \\ m \downarrow & \begin{bmatrix} k^{t-1}_{1,1} \xrightarrow{f_{t-1}} \xrightarrow{f_{t-1}} \cdots \xrightarrow{f_{t-1}} k^{t-1}_{1,t} \\ \\ k^{t-1}_{m,1} \xrightarrow{f_{t-1}} \xrightarrow{f_{t-1}} \cdots \xrightarrow{f_{t-1}} k^{t-1}_{m,t} \end{bmatrix} \\ & \overrightarrow{\quad t \quad} \\ m \downarrow & \begin{bmatrix} k^t_{1,1} \xrightarrow{f_t} \xrightarrow{f_t} \cdots \xrightarrow{f_t} k^t_{1,t} \\ \\ k^t_{m,1} \xrightarrow{f_t} \xrightarrow{f_t} \cdots \xrightarrow{f_t} k^t_{m,t} \end{bmatrix} \end{array}$$

$$\begin{array}{cc} & \overrightarrow{\qquad t \qquad} \\ m \times t \downarrow & \begin{bmatrix} k_{1,1} \xrightarrow{f_1} \xrightarrow{f_2} \qquad \xrightarrow{f_{t-1}} k_{1,t} \\ \\ \\ k_{mt,1} \xrightarrow{f_1} \xrightarrow{f_2} \qquad \xrightarrow{f_{t-1}} k_{mt,t} \end{bmatrix} \end{array}$$

Recherche dans une table rainbow

- Lorsqu'on qu'on doit rechercher un hash *h* dans une table rainbow, on ne sait pas par quelle fonction de réduction commencer.
- On essaie toutes les possibilités, en commençant par la fin (là ou c'est le plus court)



Qualités des tables rainbow

- Les tables rainbow peuvent être *t* fois plus grandes que les tables classiques
- Il faut deux fois moins d'opérations pour chercher dans une table rainbow que dans *t* tables classiques (cas le pire, bien mieux en moyenne, p.ex. 14 fois)
- Les tables rainbow peuvent être générées plus efficacement
- Les tables rainbow peuvent facilement être analysées mathématiquement

Les tables parfaites

- Pour plus d'efficacité on préfère avoir des tables sans aucune fusion: « tables parfaites » (Borst & Preneel '98)
- Les chaînes fusionnées se détectent par leur fin de chaîne identique
- Il y a une limite au nombre maximal de chaînes m_{max} sans fusion que l'on peut générer:

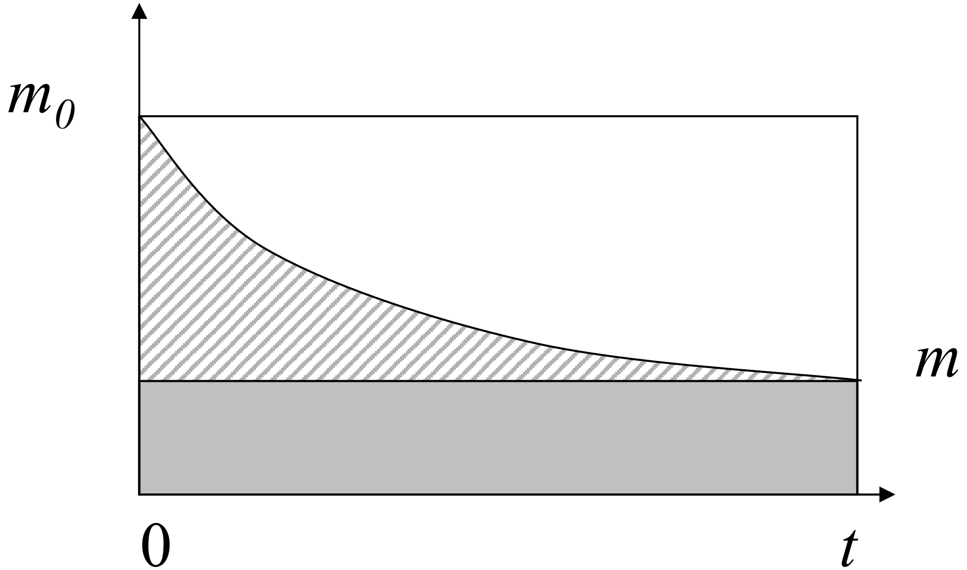
$$m_{max}(t) \approx \frac{2N}{t+2}$$

- Le taux de réussite d'une table qui a le nombre maximal de chaînes sans fusions et de:

$$\hat{P}_{table} = 1 - (1 - \frac{m_{max}}{N})^t \approx 1 - e^{-t\frac{m_{max}}{N}} \approx 1 - e^{-2} = 86\%$$

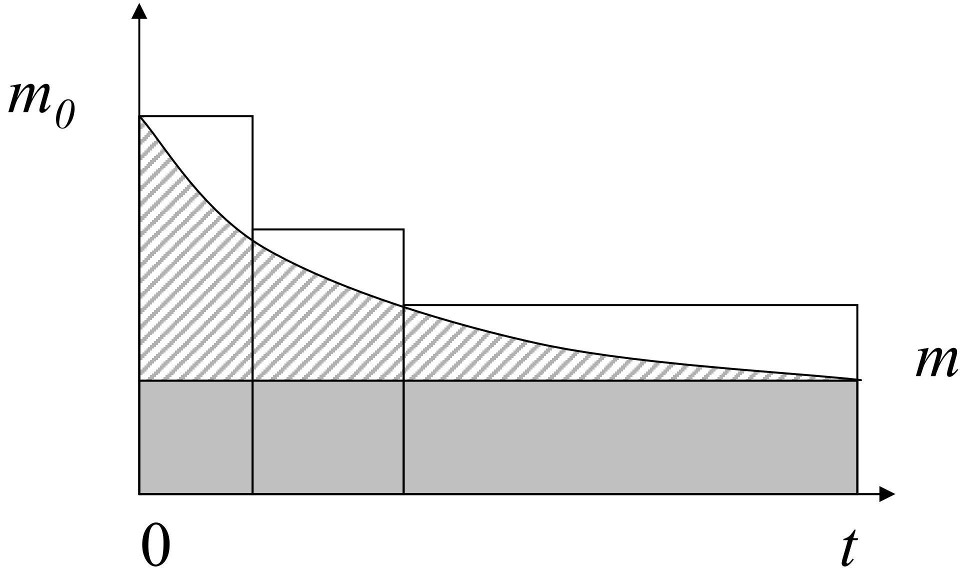
Génération de tables parfaites

- Pour générer une table parfaite
  - On génère trop de chaînes
  - On retire les fusions



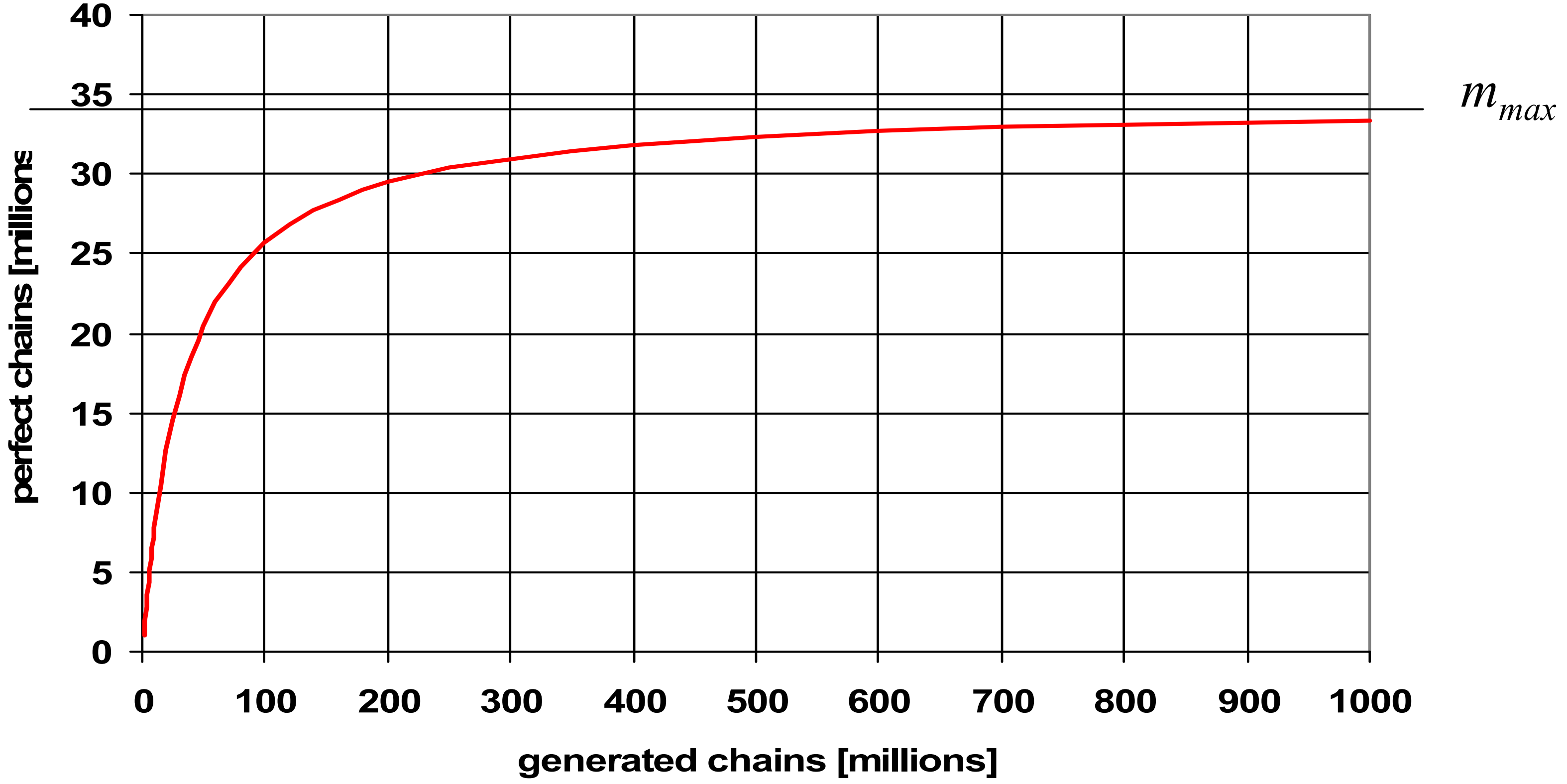
Génération efficace de tables parfaites

- On peut considérablement réduire l'effort de génération des tables en éliminant les fusions par étapes successives



Nombre maximal de chaînes parfaites

- Il est très cher d'aller jusqu'au nombre maximum de chaînes (*t=4666)*



Paramètres optimaux (*t*, *m*, *ℓ*)

- Etant donné une mémoire pouvant contenir *M* chaînes et un taux de réussite désiré *P*, la configuration optimale est atteinte en minimisant le nombre de tables:

- Nombre de tables

$$\ell = \left\lceil \frac{-\ln(1-P)}{2} \right\rceil$$

- Nombre de chaînes parfaites par table

$$m = \frac{M}{\ell}$$

- Longueur des chaînes

$$t \approx \frac{-N}{M} \ln(1-P)$$

Performance

- Le nombre moyen d'opérations pour trouver un hash est composé
  - des opérations nécessaires à générer les chaînes partielles
  - des opérations dues à la vérification de fausses alarmes

- Quand on recherche un hash dans ℓ tables de *m* chaînes de longueur *t*, on cherche d'abord dans la dernière colonne de chaque table, puis l dans l'avant-dernière, etc

- Il y a $\ell\, t$ recherches, à la *k*-ième recherche on se trouve à la colonne

$$c = t - \lfloor \frac{k}{\ell} \rfloor$$

Espérance de *T*

- La probabilité de trouver un mot de passe à la *k*-ème recherche est

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

- La probabilité d'avoir une fausse alarme à la colonne *c* est

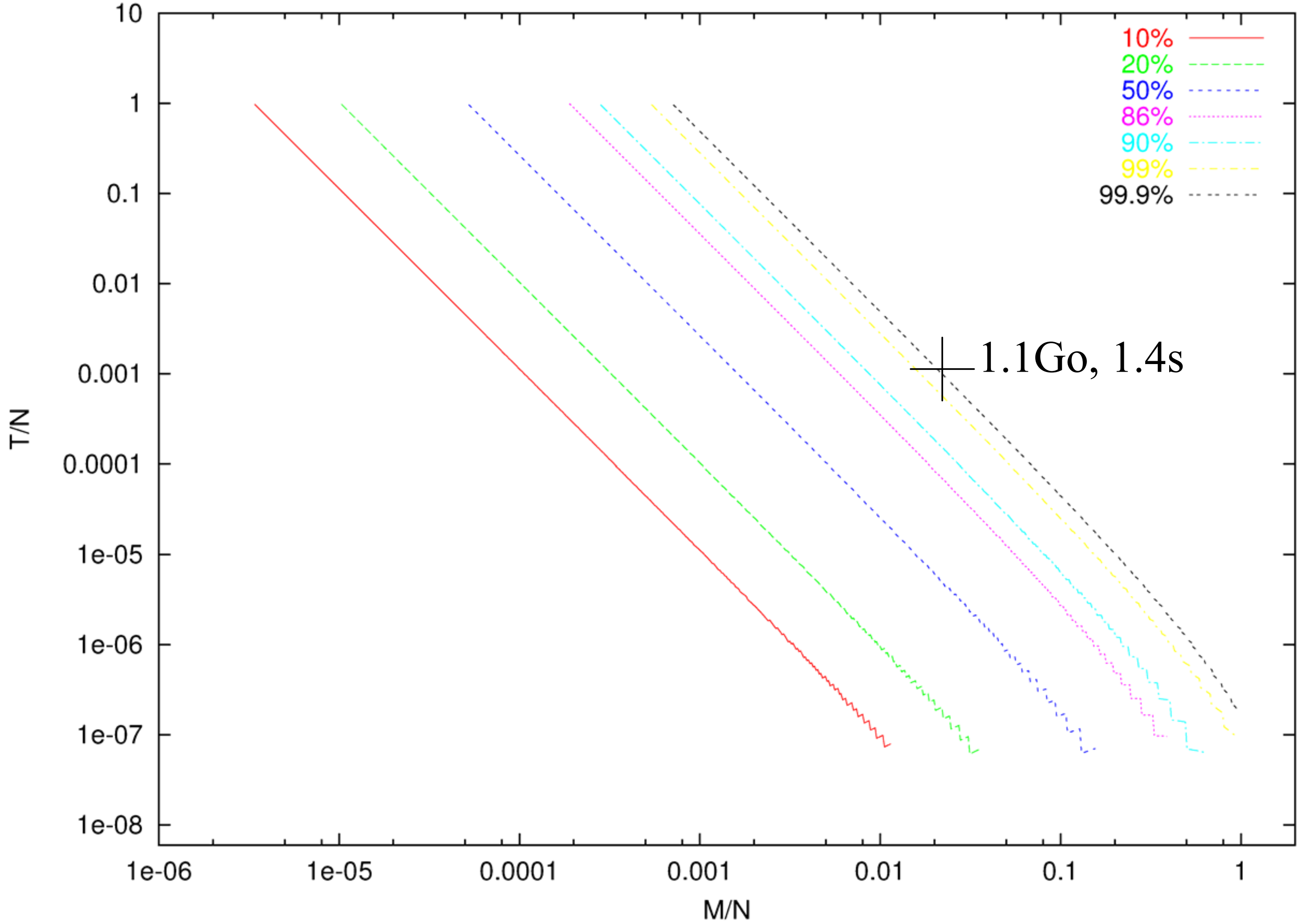
$$q_c = 1 - \prod_{i=c}^{i=t}(1 - \frac{m_i}{N}) - \frac{m}{N}$$

- On trouve:

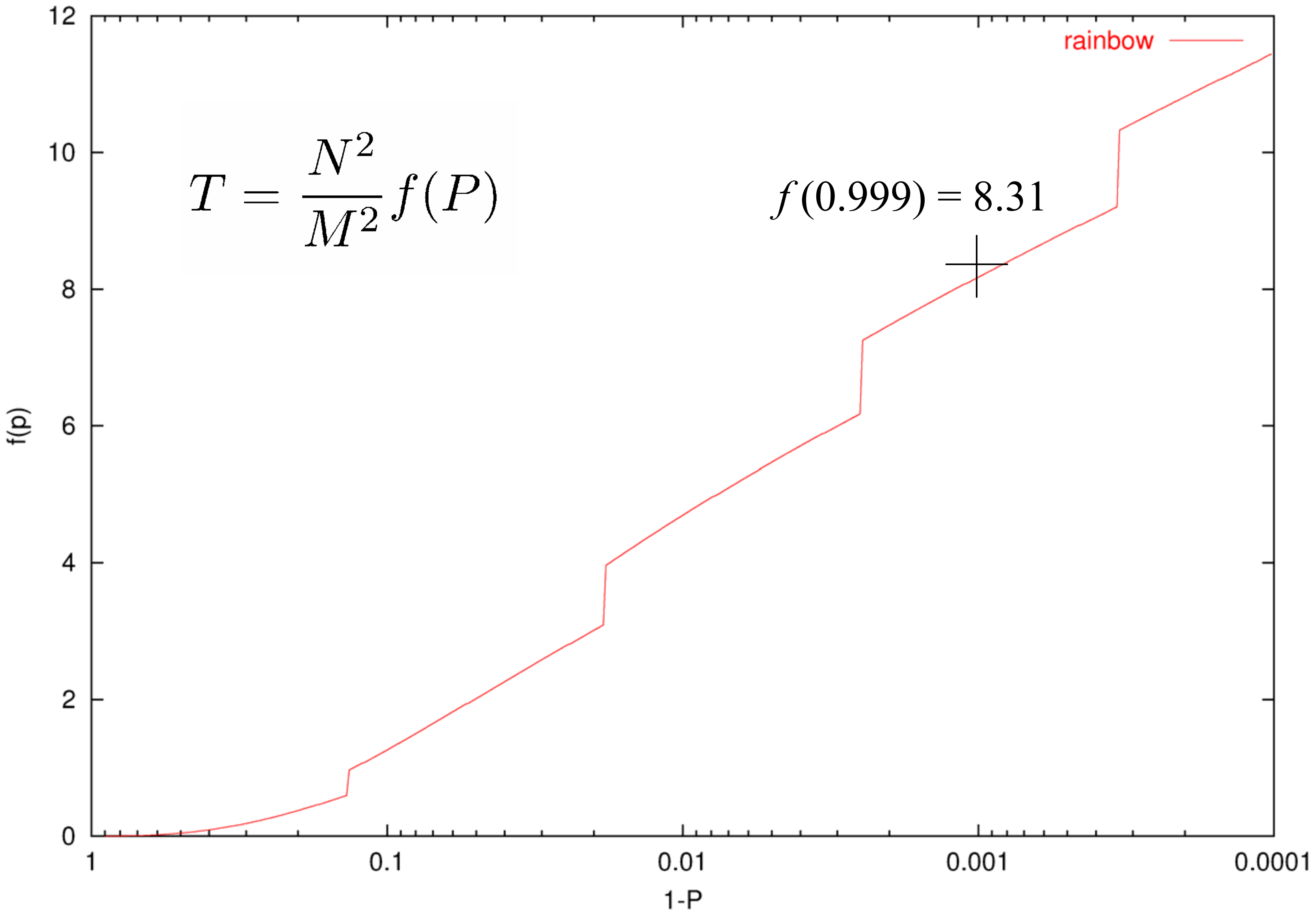
$$T = \sum_{\substack{k=1}}^{\substack{k=\ell t \\ c=t-\lfloor \frac{k}{\ell} \rfloor}} p_k \left(\frac{(t-c)(t-c-1)}{2} + \sum_{i=c}^{i=t} q_i i \right) \ell + (1 - \frac{m}{N})^{\ell t} \left(\frac{t(t-1)}{2} + \sum_{i=1}^{i=t} q_i i \right) \ell$$

LASEC

Performance



Caractéristique *f(P)* du compromis



Implémentation efficace

- Calcul des hash:
  - Implémentation optimisée de DES
    - Bitslice DES?
- Stockage des chaînes
  - La performance du compromis dépend de la mémoire au carré
  - On pourrait stocker les début et les fins de chaînes comme hash (64 bits) ou mot de passe (56 bits)
  - Les mots de passe alphanumériques n'utilisent pas toutes les 2^{56} possibilités
  - Les fin de chaînes sont triées pour faciliter la recherche

Représentation compacte des chaînes

- Il n’y a que $2^{36.23}$ mots de passe alphanumériques, on peut donc les coder sur 37 bits au lieu de 7 bytes ascii.
- Les fins de chaînes ont un préfix qui change rarement
  - En créant un index, on peut retirer les préfix, p.ex:
  - Préfix 1 car et 6 car sur 32 bits
  - Préfix 4 car et 3 car sur 16 bits
- Il n’y a que m_0 débuts de chaîne possibles
  - On peut les coder sur $\log_2(m_0)$ bits

```
 Début: Fin
35351N:00001UJ
2YTE99:000049R
4FITAB:0000738
4CG3QK:000082X
4N7J4B:0000987
......:.......
1XH5G8:000ZYCQ
01ZN8N:00101K7
1IJ6A3:00102B7
```

Exemples

- Démo été 2003:
  - P = 99.9% ,
  - M = 116 mio de chaînes, 910Mo (préfix 1 car),
  - ℓ = 5, m = 23.3 mio (m_0 = 90 mio), t = 4666
  - T = 4.11 mio, 5.26s (AthlonXP 2500+, 1.8Ghz)
- Démo actuelle:
  - P = 99.9% ,
  - M = 186 mio de chaînes, 1.07Go (préfix 4 car),
  - ℓ = 4, m = 46.5 mio (m_0 = 350 mi0), t = 3000
  - T = 1.56 mio, 1.5s (AthlonXP 2500+, 1.8Ghz)

Pourquoi ça marche si bien?

- Les compromis sont applicable quand:
  - Il n'y a pas de partie aléatoire dans le calcul des hash (pas de sel, pas de vecteur d'initialisation)
    - ni dans le LMHash, ni dans le NTHash
  - Le problème n'est pas trop complexe (2^{40}), ni trop simple (2^{30})
- Il n'y a pas d'autre système d'exploitation à notre connaissance qui n'utilise pas du sel dans les hash de mot de passe
- Les systèmes de chiffrement utilisent des vecteurs d'initialisation:
  - protections de fichiers ZIP, MS-Office, Acrobat,
  - IPSec, SSL, Ms-Kerberos
- Certains drivers WLAN/WEP génèrent des vecteurs d'initialisation prédictibles

Conclusions

- Il faut toujours ajouter un facteur aléatoire dans tous les systèmes de chiffrement, pour éviter la précalculation
- L’efficacité des compromis temps-mémoire augmente avec les progrès des processeurs ET de la mémoire
  - C’est la loi de Moore’s au carré (ou au cube)!

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

- Espérons que Microsoft va bientôt ajouter du sel à ses mots de passe!

Questions ?