Password Cracking with Time-Memory Trade Offs

By Jason Roth

Hash Tables

Hash	**Password**
AF57D880DED9D80C2165C17D137D7367	God
BED04F914C925E4886647531319D3DBC	Love
…	…
E83845900884B28910500C574A0EB83C	Secret
FCEF869E7B2FC1F7F92F8CA25A145EB3	Sex

Hash Tables

- Pros
  - Constant time password lookups
  - Precomputation equivalent to brute force
  - Massively parallel

- Cons
  - Disk space
  - More disk space

The First Time-Memory Trade-Off

$f(\mathrm{P}_i) = \mathrm{C}_i \qquad \mathrm{R}(\mathrm{C}_i) = \mathrm{P}_{i+1} \qquad \mathrm{X} = \mathrm{R}[f(\mathrm{P}_i)]$

$\mathrm{SP}_0 = \mathrm{X}_{00} \rightarrow \mathrm{X}_{01} \rightarrow \mathrm{X}_{02} \rightarrow \ldots \rightarrow \mathrm{X}_{0t} = \mathrm{EP}_0$

$\mathrm{SP}_1 = \mathrm{X}_{10} \rightarrow \mathrm{X}_{11} \rightarrow \mathrm{X}_{12} \rightarrow \ldots \rightarrow \mathrm{X}_{1t} = \mathrm{EP}_1$

…

$SP_m = \mathrm{X}_{m0} \rightarrow \mathrm{X}_{m1} \rightarrow \mathrm{X}_{m2} \rightarrow \ldots \rightarrow \mathrm{X}_{mt} = \mathrm{EP}_m$

The First Time-Memory Trade-Off

- Pros
  - Reusable data set
  - Cracks passwords in $O(t^2)$ time
- Cons
  - Requires *tm* memory
  - Doesn't really crack passwords in $O(t^2)$ time
  - Chain merges

Chain Merges

“aaaa” → “qohf” → “qmdn” → …. → “owmf”

“bvcx” → “qohf” → “qmdn” → …. → “owmf”

“qpye”→ “bfdsa” → “rofw” → “pold” → “lsde”

“bfdsa” → “rofw” → “pold” → “lsde” → “isln”

Time-Memory Trade-Off using Distinguished Points

$f(P_i) = C_i \qquad R(C_i) = P_{i+1} \qquad X = R[f(P_i)]$

$SP_0 = X_{0,0} \rightarrow X_{0,1} \rightarrow X_{0,2} \rightarrow EP_0$

$SP_1 = X_{1,0} \rightarrow EP_1$

…

$SP_m = X_{m,0} \rightarrow X_{m,1} \rightarrow X_{m,2} \rightarrow \ldots \rightarrow X_{m,329} = EP_m$

Merges and Loops

“aaaa”→“qohf”→“qmdn”→ …. →“owmf”

“rcxb” →“qohf”→“qmdn”→ …. →“owmf”

“qpye”→“bfdsa”→“rofw”→“pold”

“rofw” → “pold”

“asdf”→”qwer”→”asdf”→”qwer”→”asdf”

The First Time-Memory Trade-Off

- Pros
  - Cracks passwords in O(t) time
  - All merges are detectable
- Cons
  - Same amount of chain merges

Rainbow Tables

$f(P_i) = C_i$ $\quad R_{i,l}(C_i) = P_{i+1}$ $\quad X = R_{i,l}[f(P_i)]$

$SP_0 = X_{00} \rightarrow X_{01} \rightarrow X_{02} \rightarrow \ldots \rightarrow X_{0t} = EP_0$

$SP_1 = X_{10} \rightarrow X_{11} \rightarrow X_{12} \rightarrow \ldots \rightarrow X_{1t} = EP_1$

…

$SP_m = X_{m0} \rightarrow X_{m1} \rightarrow X_{m2} \rightarrow \ldots \rightarrow X_{mt} = EP_m$

Merges

"aaaa" → "qohf" → "qmdn" → …. → "owmf"
"bvcx" → "qohf" → "qmdn" → …. → "owmf"

"qpye"→ "bfdsa"→ "rofw"→ "pold"→ "lsde"
"bfdsa"→ "rofw" → "fhgs" → "argv" → "isln"

"poiu" →"kjhg" → "rtuy" → "mncx" → "dfxs"
"lkdf" → "kduy" → "kdos" → "rieq" → "poiu"

Rainbow Tables

- Pros
  - All merges are detectable
  - Merges are reduced by a factor of (t x l)

- Cons
  - Back to O(t^2) search time

Problems with Rainbow Tables

- Doesn’t do anything about merges
- There's a whole lot of merges
  - 50% of chains to be exact
  - Assuming an even distribution of merges, this results in 25% of values being repeated
  - Which is back to the efficiency of distinguished points
  - Without the speed increase

Programming 101

- C++ classes, C file I/O
- 9 different classes when you need about 2
- C++ with goto statements
- Inline assembly to implement multiplication
- Using modulus in the reduction function

High Quality Rainbow Tables*

Hash	Character Set	Length	Cost
LM	Alpha Numeric ASCII	1-14	$400
LM	Alpha Numeric	1-7	$100
MD5	Alpha Numeric	1-7	$100
NTLM	Alpha Numeric	1-7	$100

*while supplies last