Probabilistic password generators
(and fancy curves)

Simon Marechal (bartavelle at openwall.com)
http://www.openwall.com
@Openwall

December 2012
I am no mathematician

Conclusions might be erroneous
  - Bugs!

All conclusions are relative to public leaks, specifically the 2012 Yahoo Contributor Network leak
  - 453491 distinct passwords
  - 342514 unique passwords
  - Unique passwords used, to reduce biases (and introduce new ones, hopefully less problematic)

The training set is the rockyou list
A technique for generating candidate passwords from a statistical model

Notations

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P(x)$</td>
<td>probabilistic distribution of all characters at position $x$</td>
</tr>
<tr>
<td>$p(x, y)$</td>
<td>probability that the character at position $x$ is $y$</td>
</tr>
<tr>
<td>$c(x)$</td>
<td>character at position $x$</td>
</tr>
<tr>
<td>$P'(x)$</td>
<td>$\lceil -K \cdot \log(P(x)) \rceil$</td>
</tr>
<tr>
<td>$p'(x, y)$</td>
<td>$\lceil -K \cdot \log(p(x, y)) \rceil$</td>
</tr>
<tr>
<td>$\Psi(pass)$</td>
<td>probability that a password is chosen</td>
</tr>
</tbody>
</table>
It is common to store log-probabilities instead of raw probabilities. The reason for rounding them will be apparent later. Please note that:

- A likely event will have a $P$ value close to 1, and a $P'$ close to 0
- $P_1 \cdot P_2 \cdot P_3$ will turn onto $P'_1 + P'_2 + P'_3$
- $P'$ is nicer to look at than $P$
Well known cracking methods

Examples

<table>
<thead>
<tr>
<th>$P(x)$ is a function of</th>
<th>Cracking paradigm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nothing (constant)</td>
<td>Naive exhaustive search, standard rainbow tables, frequency optimized search, JtR Markov mode</td>
</tr>
<tr>
<td>$c(x-1)$</td>
<td>JtR Markov mode</td>
</tr>
<tr>
<td>$c(x-2)$, $c(x-1)$, $x$, $l$</td>
<td>JtR incremental mode, for each length $l$</td>
</tr>
<tr>
<td>$c(x-1), x$</td>
<td>Hashcat per position Markov mode</td>
</tr>
</tbody>
</table>

Some distributions have special properties. This talk will focus on distributions that are only functions of the previous characters (i.e. can be modeled as Markov chains). They can be written as:

$$P(x) = f(c(x-1), c(x-2), ..., c(0), x)$$
What for?

- Find a model that fits well with real world password selection
- Compute the parameters that fit a training set
- Generate all candidate passwords that satisfy some condition and use them for cracking
  - Every per-character log-probability of occurrence is less than a given threshold
  - The sum of the log-probabilities of each character in a candidate password is less than a given threshold (we will only consider this case)

Model

- \( \Psi(\text{pass}) = p(0, p) \times p(1, a) \times p(2, s) \times p(3, s) \)
- \( \Psi'(\text{pass}) = p'(0, p) + p'(1, a) + p'(2, s) + p'(3, s) \)
- For a maximum probability \( \psi \), generate and crack all \( \{ p \mid \Psi'(p) < \psi' \} \)

We can think of \( \psi' \) as a budget to spend on individual \( p' \)
These probability distributions have the following nice properties:

▶ It is possible to count the number of words $p$ satisfying $\Psi'(p) < \psi'$ (called $nbparts$)
  ▶ Actually it is possible to enumerate many related values

▶ Once done, it is easy to generate the $n^{th}$ password (this is important for rainbow tables and distributed computing)

▶ It is possible to quickly compute $\Psi'(p)$ for arbitrary passwords provided that we give $\nu$, $\forall (x, y) \in \{p(x, y) = 0\}, p'(x, y) = \nu$
  ▶ We can compute $nbparts$ for every value of $p$, thus estimate how long it would take to crack this password using this model
  ▶ Yes, that means you can fill your reports with curves
Partial results, ran Markov 290 (explains the second drop)
- Multiple humps, typical of frankencurves
- Huge drop after the peak at 250. Are there Markov generated passwords?
### State definition

Let’s use $P(x) = f(c(x - 1))$, ie. JtR Markov mode

- The **reduced** state is the previous character
- The **full** state is the tuple (previous character, remaining budget, remaining length)
- Initial full state could be $(\emptyset, 100, 10)$

### Training set

- abc
- aaa
- bac
- ccab
Take advantage of the state machine structure:

- Build the state transition graph \((\textit{reduced} \text{ state})\)
- Map all \(\textit{full} \text{ states} \) into \(\textit{reduced} \text{ states} \)
- Map all \(\textit{reduced} \text{ states} \) into \(\textit{full} \text{ states} \) that could be derived from it
- Start with the initial \(\textit{full} \text{ state} \)
- From a full set, compute the reduced set, and recursively run this step for all valid derived \(\textit{full} \text{ states} \)
  - When the function finishes, store the \((\textit{full} \text{ state}, \text{password count})\) pair for caching
  - Exploit node collisions (thanks to the rounding)
  - Memory and time usage orders of magnitude lower than password count
### Inner state transition

| $n$ | $c(x-1)$ | $c(x)$ | $p' = \lfloor -10. \ln(p(x, c(x)|c(x-1)) \rfloor$ |
|-----|----------|--------|---------------------------------|
| 0   |          | a      | 6                              |
| 0   |          | b      | 13                             |
| 0   |          | c      | 13                             |
| > 0 | a        | a      | 9                              |
| > 0 | b        | a      | 6                              |
| > 0 | a        | b      | 9                              |
| > 0 | c        | a      | 6                              |
| > 0 | a        | c      | 16                             |
| > 0 | b        | c      | 6                              |
| > 0 | c        | c      | 6                              |
Password generation can be modeled as a state machine:

Figure: The resulting state machine
1. We start with an empty reduced state, $\psi' = 100$, length budget of 10, and $nbparts = \emptyset$. The full state is $(\emptyset, 100, 10)$
2. The list of acceptable next reduced states is $(a, 6), (b, 13), (c, 13)$
3. Start with $(a, 6)$. The next full state is $(a, 94, 9)$. It is not in $nbparts$, so the algorithm keeps going
4. Continue until the length or budget is depleted
5. Store the password count related to this node in $nbparts$

With this training set, 621 nodes will be generated, and the result will be 58314 passwords
Computing nbparts
Exploiting password structure

Known optimization (cf. “mask mode”, Weir thesis)

- Password is made of subsequent characters of the same class (upper, lower, digits, special)
- Can be modeled as a Markov thingy. For example, pass123 can be modeled as:
  - A chain of types [Lower, Digit] – the "no length" model
  - A chain of types with length [Lower 4, Digit 3] – the "part type and length" model
- Each part can be modeled as previously
- \[ \psi'_p(pass123) = B \cdot \psi'([L4, D3]) + \psi'(pass) + \psi'(123) \]
  - \( B \) is a constant that must be tuned
Much harder! Will be written $nbparts_p$ (for patterns)

- Generate the $nbparts$ graph for patterns, \textit{but}:
  - At each node, have intermediate states, one for each point of remaining budget
  - Compute the sub-part $nbparts$ for each of these states
  - And multiply by the $nbparts_p$ of the next nodes
Same procedure as before, but for patterns. Let’s say we pick $U4$, and have a ”budget” of 20

- Generate 18 intermediate states, from 1 to 19
- For each state $i$, ”spend” $i$ on a 4 uppercase letters subpart, and $20 - i$ for the remaining parts
  - let $n_i = nbparts(\Psi' = i, \text{length} = 4)$
  - let $S$ be the state of valid next full states
  - $n_i = \sum_{s \in S} nbparts_p(\Psi' = n - i, s)$
  - $nbparts_{p,i} = (n_i + 1)_{\text{next}_i}$
- $nbparts_p = \sum_{i=1..19} nbparts_{p,i}$
Not so fast!

How to compute $nbparts(P' = i, \text{length} = 4)$? All we can do is $nbparts(P' \leq i, \text{length} \leq 4)$!

- Pretty obvious when written like this. Took me two days to realize ...

- $nbparts(P' = i, \text{length} \leq 4) = nbparts(P' \leq i, \text{length} \leq 4) - nbparts(P' \leq i - 1, \text{length} \leq 4)$

- Same reasoning for fixing the length. Beware of edge cases
Main loop – is there a bug?

```
calcpatternsnbpats' malus !type !stats stt@(LvlState !curlvl !curstate) !curparts !ns !snbpats = let
  !correctstates = filter (\(_,l\) -> l <= (curlvl `div` malus)) $! curstates ns gtype curstate
gennext (mp, c) su =
  let (s,lnomalus) = downgrade gtype su
  l = lnomalus * malus
  remaining = curlvl - l
  (Pattern nextstateType nextstateLen) = getNextState su
  nbpartsmap = snbpats Map.! nextstateType
  gm 0 = 0
  gm _0 = 1
  gm ln v =
    let lo | ln >= 32 = HM.lookup (LvlState 31 v NoState) nbpartsmap
      | otherwise = HM.lookup (LvlState ln v NoState) nbpartsmap
    in case lo of
      Just !x -> x
      Nothing -> error $ "Would not find " ++ show (l, v)
  getnbpats 0 = 0
  getnbpats lv =
    ( gm nextstateLen lv
    - gm (nextstateLen-1) lv )
    - ( gm nextstateLen (lv-1) - gm (nextstateLen-1) (lv-1) )
  levelsToTry = [(lv, getnbpats lv)] lv <- [1..remaining]
  trylevel (tnmp, tnc) (tlvl, tnbpats) = let
    (nnmp, nnc) = calcpatternsnbpats' malus gtype stats (LvlState 0 (remaining - tlvl) s) tnmp ns snbpats
    !res = tnc+(nnc+1)*tnbpats
    in (nnmp, res)
    in (nnmp, c+ncc)
  (nnm, lcount) = foldl' gennext (curparts, 0) correctstates
  in case HM.lookup stt curparts of
    Just x -> (curparts, x)
    Nothing -> (HM.insert stt count nn, count)
```
Frequency optimized exhaustive search

- Search all passwords made with a charset of \( n \) elements
- Start with the shortest passwords and most frequent characters
- What is the best value for \( n \) ?
- For my sample, 36: ae1iorns2lt0m3dc9hu847by56kgpwjfvsxq

Figure: Passwords found per candidates tested, for various charset length
Markov like modes : Model Structure/Subpart/B value

- **Markov mode:**
  - M1 : Markov using the previous item (an item is a character or a part template)
  - M2 : Markov using the two previous items

- **Model type:**
  - No model
  - Model part *type* and length
  - Model part *type* only

- **B value:**
  - As explained previously, the ”score” of a password is the sum of the scores of all subparts, plus B times the score of the structure
  - \( \Psi'_p(pass123) = B.\Psi'([L4, D3]) + \Psi'(pass) + \Psi'(123) \)
So, part/type/length M2/M2/B2 means:

- Each structure item is a (character type, length) pair
- Structure modeled with Markov using the two previous items
- Each part is modeled with Markov using the two previous characters
- Total cost is the sum of the costs of all parts plus twice the cost of the structure
Results – wordlists and mangling rules

- Used two widely used wordlists: wikipedia-sraveau and rockyou
- Used a good and large list of mangling rules (see mangling rules presentation)
- Real world results are better, as word rejection hasn’t been taken into account in the figures
The following figures draw the ratio of passwords found per candidates tested, for various candidate generation methods

The x-axis ticks are labelled with: candidates tested / fast hash / slow hash

- The fast hash time is computed for 5400M c/s (oclHashcat, stock HD7970, 100k MD5 hashes)
- The slow hash time is computed for 1340 c/s (John the Ripper, 2x X5650, 100 BCrypt $2a$08 hashes)

<table>
<thead>
<tr>
<th>Count</th>
<th>MD5</th>
<th>BCrypt $2a$08</th>
</tr>
</thead>
<tbody>
<tr>
<td>1e3</td>
<td>0s</td>
<td>74s</td>
</tr>
<tr>
<td>1e6</td>
<td>0s</td>
<td>20h 43m</td>
</tr>
<tr>
<td>1e9</td>
<td>0s</td>
<td>2y 133d</td>
</tr>
<tr>
<td>1e12</td>
<td>185s</td>
<td>2364y 285d</td>
</tr>
<tr>
<td>1e15</td>
<td>51h 26m</td>
<td>-</td>
</tr>
<tr>
<td>1e18</td>
<td>5y 317d</td>
<td>-</td>
</tr>
</tbody>
</table>
Comparing all values of B

M1/M0

M1/M1

M1/M2

M2/M0

M2/M1

M2/M2
Results – part type only, best B

Figure: Passwords cracked per candidates tested.
Results – part type and length

Comparing all values of B

M1/M0

M1/M1

M1/M2

M2/M0

M2/M1

M2/M2
Figure: Passwords cracked per candidates tested.
Results – JtR incremental mode

Figure: Passwords cracked per candidates tested.
Results – big picture

Figure: Passwords cracked per candidates tested.
What about hard passwords?

A statistical generator is often used after a "wordlist" or "single" run. In order to account for this, the easiest passwords have been removed with the following steps:

- A selection of 754 rules from good sets (see the mangling rules presentation), against rockyou and wikipedia-sraveau
- A quick JtR Markov run (level 250, default shipped statistics)

The password count went from 342514 to 94990 (72% reduction)
Results – hard passwords

Figure: Passwords cracked per candidates tested, no trivial password
Conclusion

- The new model seems better when testing lots of passwords
  - Especially against "hard" passwords
  - Cracks a neglectable amount of passwords with little tests
  - Needs more benchmarks (fractional Bs)

- Guessing game:
  - What about implementation speed?
  - Against Hashcat Bruteforce++?

- Soon:
  - JtR implementation
  - Perhaps a rainbow table implementation
  - More benches
Questions?

http://www.openwall.com