Probabilistic password generators

(and fancy curves)

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

December 2012

Warning!

- 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

$P(x)$	probabilistic distribution of all characters at position x
$p(x, y)$	probability that the character at position x is y
$c(x)$	character at position x
$P'(x)$	$\lfloor -K.log(P(x)) \rfloor$
$p'(x, y)$	$\lfloor -K.log(p(x, y)) \rfloor$
$\Psi(pass)$	probability that a password is chosen

A word on log-probabilities

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
- $P_1.P_2.P_3$ will turn onto $P'_1 + P'_2 + P'_3$
- P' is nicer to look at than P

Well known cracking methods

Examples

$P(x)$ is a function of	Cracking paradigm
Nothing (constant)	Naive exhaustive search, standard rainbow tables, frequency optimized search
$c(x-1)$	JtR Markov mode
$c(x-2)$, $c(x-1)$, x, l	JtR incremental mode, for each length l
$c(x-1)$,x	Hashcat per position Markov mode ?

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

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

- 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

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

We can think of ψ' as a *budget* to spend on individual p'

Useful properties

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 v, $\forall(x, y) \in \{p(x, y) = 0\}, p'(x, y) = v$
  - 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

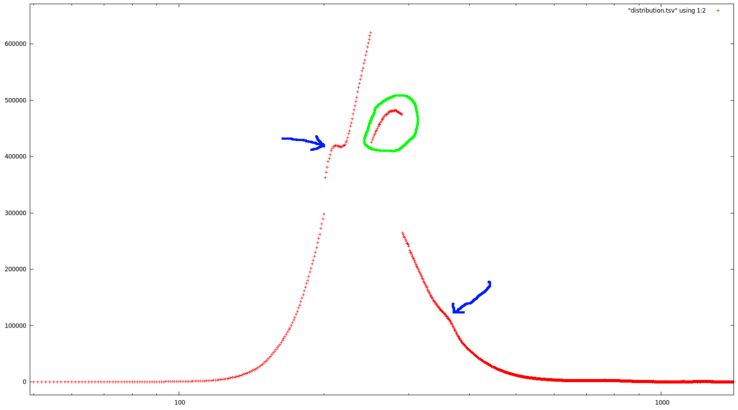


Figure: Passwords found per maximum ψ'

- 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 ?

Computing nbparts – sample 1/3

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

Computing nbparts – algorithm

Take advantage of the state machine structure:

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

Computing nbparts – sample 2/3

Inner state transition

n	$c(x-1)$	$c(x)$	$p' = \lfloor -10.ln(p(x, c(x) \vert c(x-1)) \rfloor$
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:

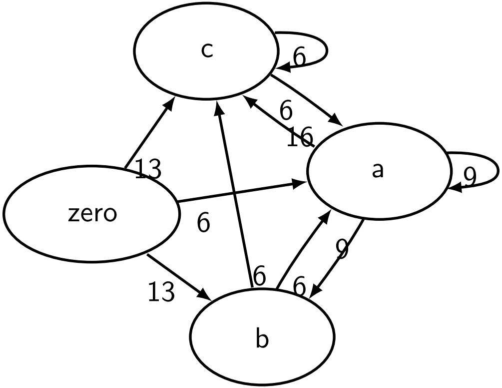
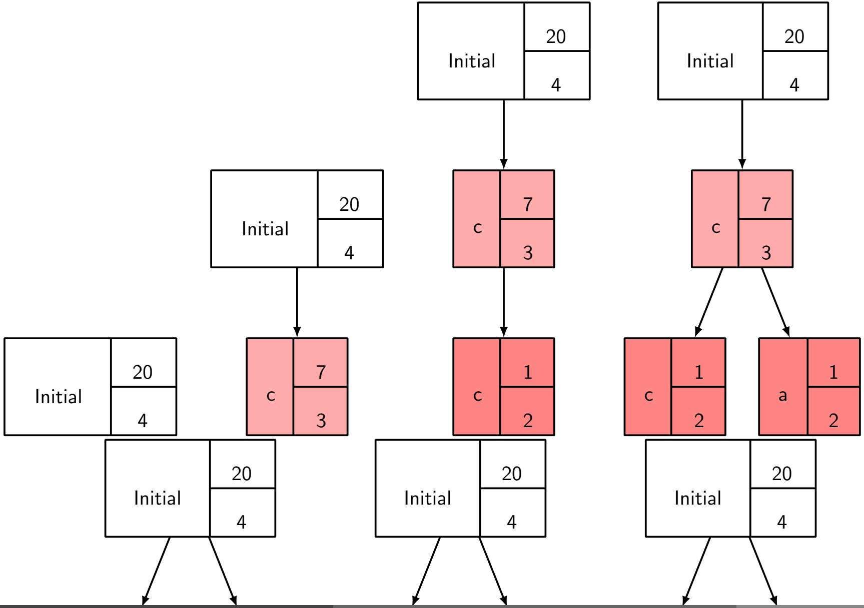


Figure: The resulting state machine

Computing nbparts – sample 3/3

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



Initial
20
4
Initial
20
4
Initial
20
4
c
7
3
c
7
3
Initial
20
4
c
7
3
c
1
2
c
1
2
a
1
2
Initial
20
4
Initial
20
4
Initial
20
4

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
- $\Psi'_p(pass123) = B.\Psi'([L4, D3]) + \Psi'(pass) + \Psi'(123)$
  - B is a constant that must be tuned

Much harder! Will be written $nbparts_p$ (for *patterns*)

- Generate the nbparts graph for patterns, *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 $n_i = nbparts(\Psi' = i, length = 4)$
  - let S be the state of valid next *full* states
  - $n_i = \sum_{s \in S} nbparts_p(\Psi' = n - i, s)$
  - $nbparts_{p,i} = (n_i + 1) next_i$
- $nbparts_p = \sum_{i=1..19} nbparts_{p,i}$

Not so fast !

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

- Pretty obvious when written like this. Took me two days to realize ...
- $nbparts(P' = i, length \leq 4) = nbparts(P' \leq i, length \leq 4) - nbparts(P' \leq i - 1, length \leq 4)$
- Same reasoning for fixing the length. Beware of edge cases

Main loop – is there a bug ?

```
calcpatternsnbparts' malus !gtype !stats stt@(LvlState _ !curlvl !curstate) !curparts !ns !snbparts = 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 = snbparts 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)
            getnbparts 0  = 0
            getnbparts lv = ( gm nextstateLen lv     - gm (nextstateLen-1) lv) -
                            ( gm nextstateLen (lv-1) - gm (nextstateLen-1) (lv-1) )
            levelsToTry = [ (lv, getnbparts lv) | lv <- [1..remaining] ]
            trylevel (tnmp, tnc) (tlvl, tnbparts) = let
                (nnmp, nnc) = calcpatternsnbparts' malus gtype stats (LvlState 0 (remaining - tlvl) s) tnmp ns snbparts
                !res = tnc+(nnc+1)*tnbparts
                in (nnmp, res)
            (nmp, nc) = foldl' trylevel (mp, 0) levelsToTry
        in  (nmp, c+nc)
    (!nm, !count) = foldl' gennext (curparts, 0) correctstates
    in case HM.lookup stt curparts of
        Just x  -> (curparts, x)
        Nothing -> (HM.insert stt count nm, 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: `ae1iorns2lt0m3dc9hu847by56kgpwjfvzxq`

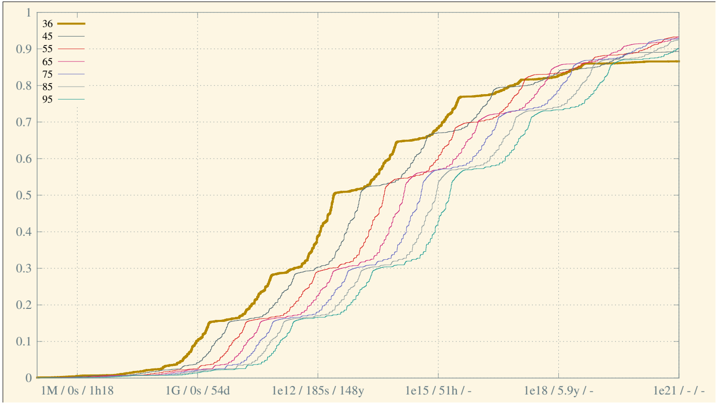


Figure: Passwords found per candidates tested, for various charset length

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
  - $\Psi'_p(pass123) = B.\Psi'([L4, D3]) + \Psi'(pass) + \Psi'(123)$

Results – Markov mode, example

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

Results – candidates tested / time spent

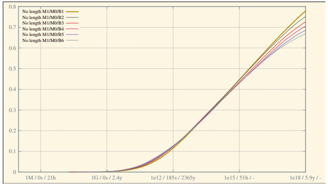
The following figures draw the ratio of passwords found per candidates tested, for various candidate generation methods

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

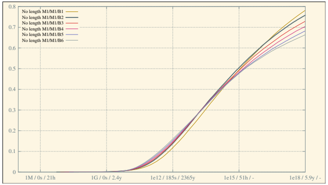
- The fast hash time is computed for 5400M c/s (oclHashcat, stock HD7970, 100k MD5 hashes)
- The slow hash time is computed for 1340 c/s (John the Ripper, 2 x X5650, 100 BCrypt $2a$08 hashes)

Count	MD5	BCrypt $2a$08
1e3	0s	74s
1e6	0s	20h 43m
1e9	0s	2y 133d
1e12	185s	2364y 285d
1e15	51h 26m	-
1e18	5y 317d	-

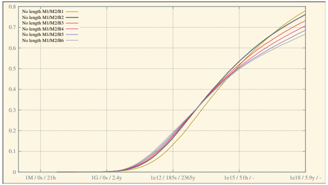
Comparing all values of B



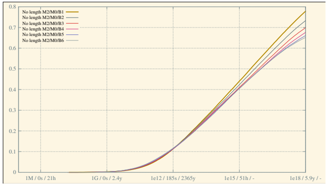
M1/M0



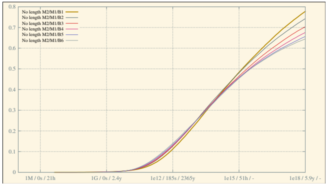
M1/M1



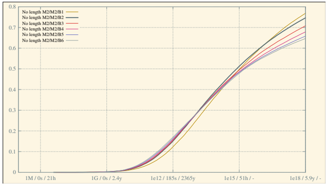
M1/M2



M2/M0



M2/M1



M2/M2

Results – part type only, best B

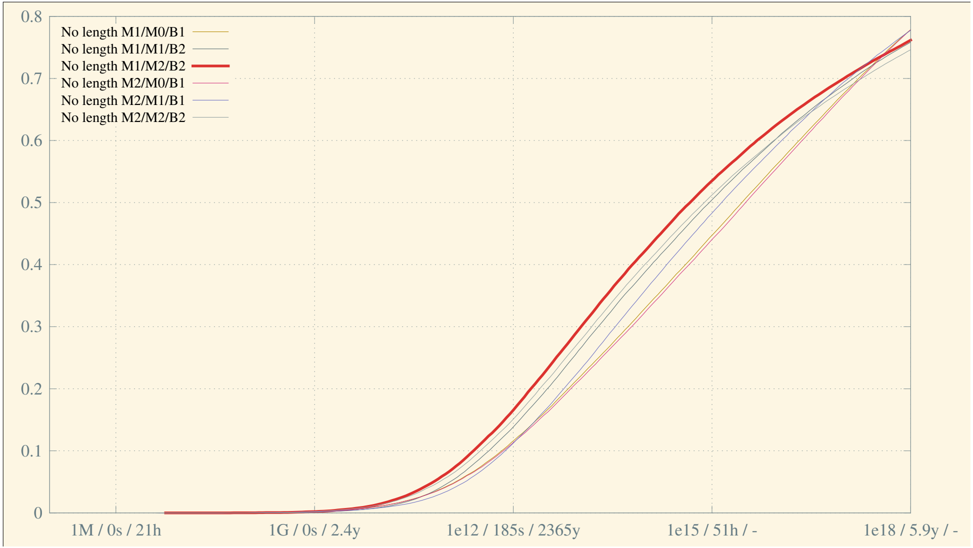
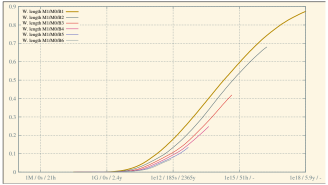
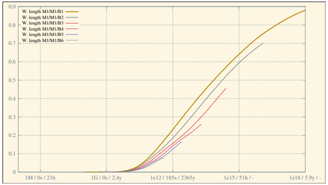


Figure: Passwords cracked per candidates tested.

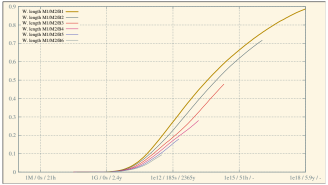
Comparing all values of B



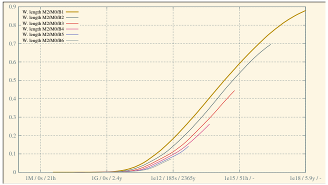
M1/M0



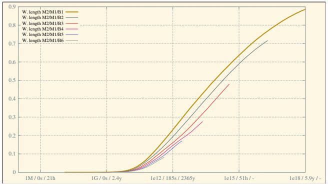
M1/M1



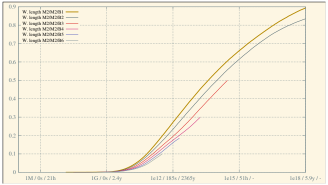
M1/M2



M2/M0



M2/M1



M2/M2

Results – part type and length, best B

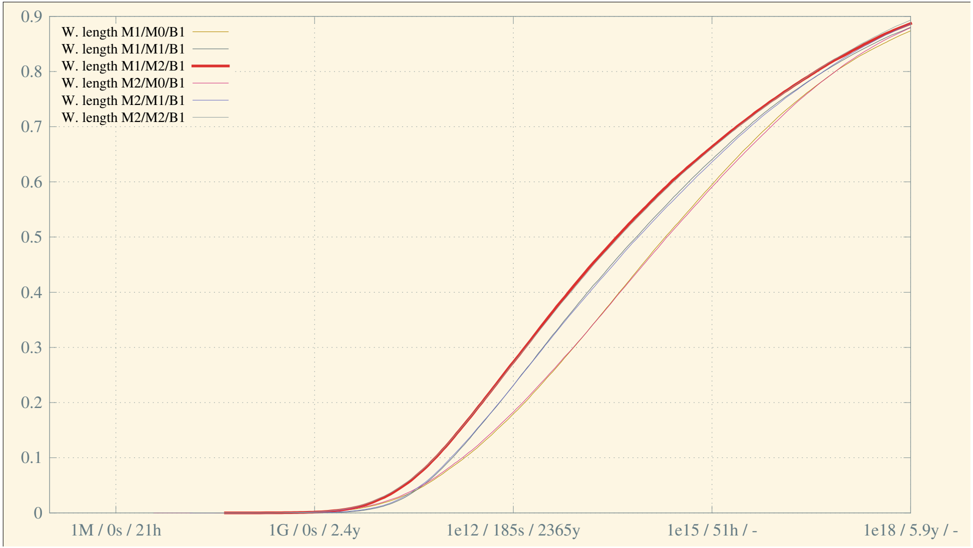


Figure: Passwords cracked per candidates tested.

Results – JtR incremental mode

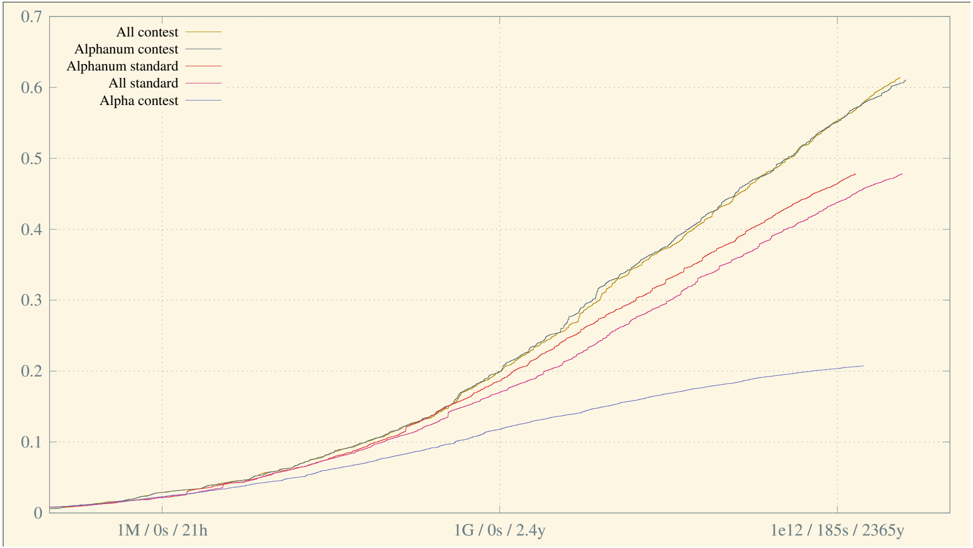
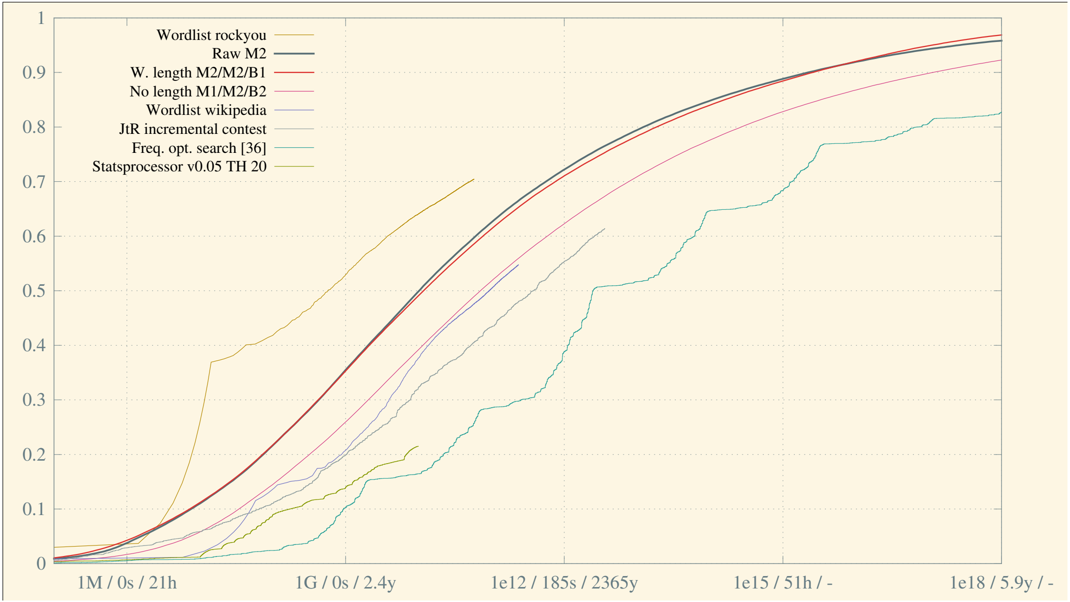


Figure: Passwords cracked per candidates tested.

Results – big picture



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

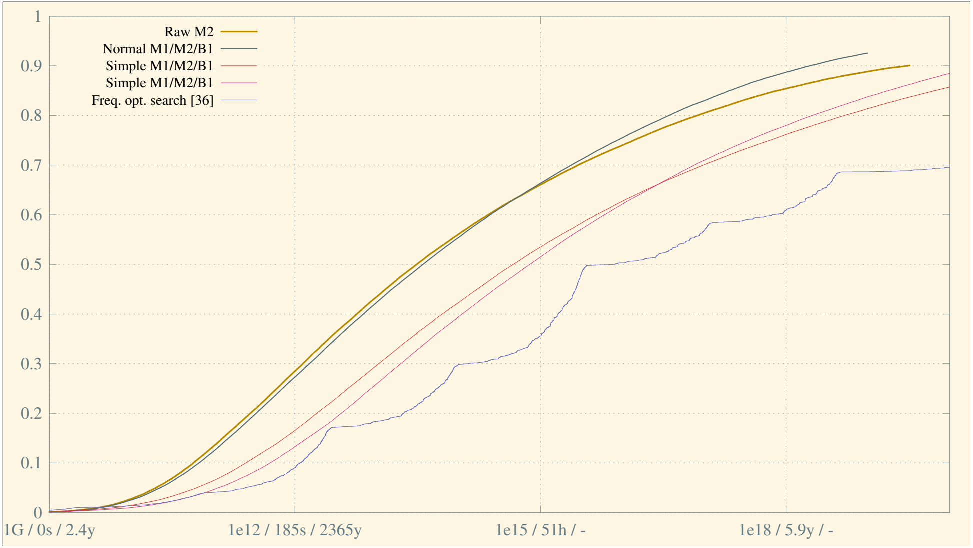


Figure: Passwords cracked per candidates tested, no trivial password

- 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