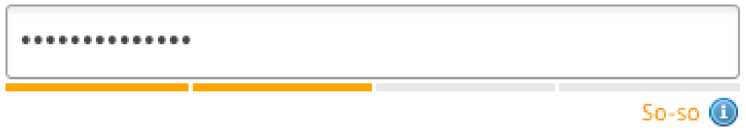
zxcvbn: realistic password strength estimation

Posted by Dan Wheeler on April 10, 2012

Over the last few months, I've seen a password strength meter on almost every signup form I've encountered. Password strength meters are on fire.



Here's a question: does a meter actually help people secure their accounts? It's less important than other areas of web security, a short sample of which include:

- Preventing online cracking with throttling or CAPTCHAs.
- Preventing offline cracking by selecting a suitably slow hash function with user-unique salts.
- Securing said password hashes.

With that disclaimer — yes. I'm convinced these meters *have the potential* to help. According to Mark Burnett's 2006 book, *Perfect Passwords: Selection, Protection, Authentication*, which counted frequencies from a few million passwords over a variety of leaks, one in nine people had a password in this top 500 list. These passwords include some real stumpers: `password1`, `compaq`, `7777777`, `merlin`, `rosebud`. Burnett ran a more recent studylast year, looking at 6 million passwords, and found an insane 99.8% occur in the top 10,000 list, with 91% in the top 1,000. The methodology and bias is an important qualifier — for example, since these passwords mostly come from cracked hashes, the list is biased towards crackable passwords to begin with.

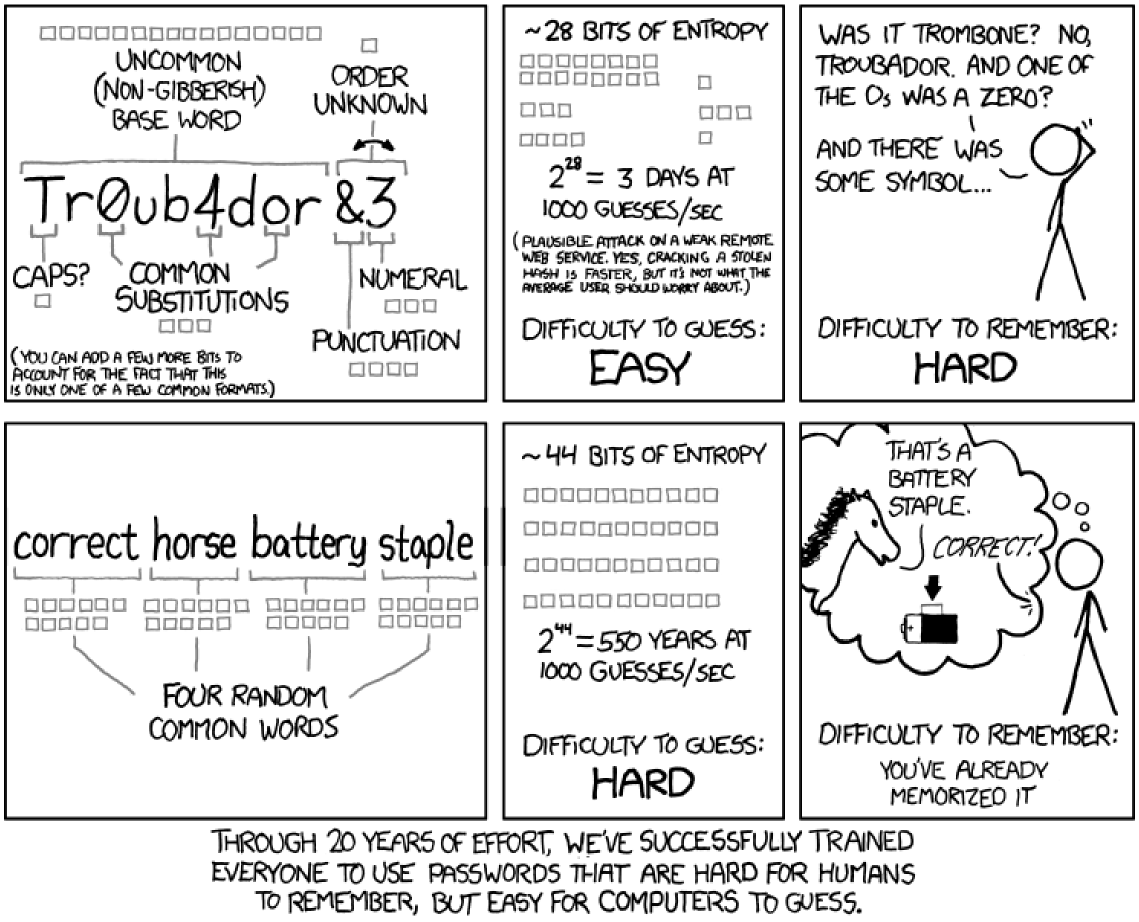
These are only the really easy-to-guess passwords. For the rest, I'd wager a large percentage are still predictable enough to be susceptible to a modest online attack. So I do think these meters could help, by encouraging stronger password decisions through direct feedback. But right now, with a few closed-source exceptions, I believe they mostly hurt. Here's why.

Strength is best measured as entropy, in bits: it's the number of times a space of possible passwords can be cut in half. A naive strength estimation goes like this:

```
# n: password length
# c: password cardinality: the size of the symbol space
#    (26 for lowercase letters only, 62 for a mix of lower+upper+numbers)
entropy = n * lg(c) # base 2 log
```

This brute-force analysis is accurate for people who choose random sequences of letters, numbers and symbols. But with few exceptions (shoutout to 1Password / KeePass), people of course choose *patterns* — dictionary words, spatial patterns like `qwerty`, `asdf` or `zxcvbn`, repeats like `aaaaaaa`, sequences like `abcdef` or `654321`, or some combination of the above. For passwords with uppercase letters, odds are it's the first letter that's uppercase. Numbers and symbols are often predictable as well: l33t speak (3 for e, 0 for o, @ or 4 for a), years, dates, zip codes, and so on.

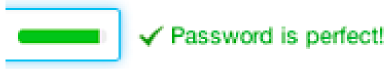
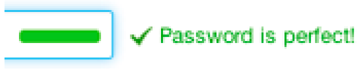
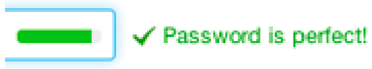
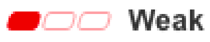
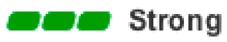
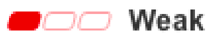
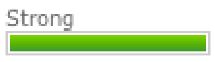
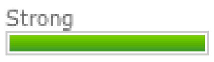
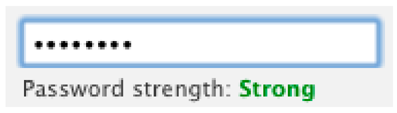
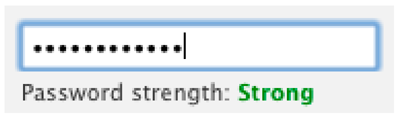
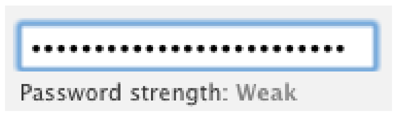
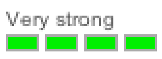
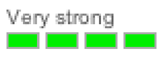
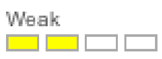
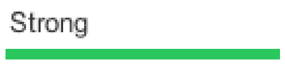
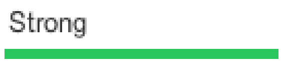
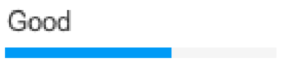
As a result, simplistic strength estimation gives bad advice. Without checking for common patterns, the practice of encouraging numbers and symbols means encouraging passwords that might only be slightly harder for a computer to crack, and yet frustratingly harder for a human to remember. xkcd nailed it:


UNCOMMON (NON-GIBBERISH) BASE WORD
ORDER UNKNOWN
Tr0ub4dor&3
CAPS?
COMMON SUBSTITUTIONS
NUMERAL
PUNCTUATION
(YOU CAN ADD A FEW MORE BITS TO ACCOUNT FOR THE FACT THAT THIS IS ONLY ONE OF A FEW COMMON FORMATS.)
~28 BITS OF ENTROPY
2^28 = 3 DAYS AT 1000 GUESSES/SEC
(PLAUSIBLE ATTACK ON A WEAK REMOTE WEB SERVICE. YES, CRACKING A STOLEN HASH IS FASTER, BUT IT'S NOT WHAT THE AVERAGE USER SHOULD WORRY ABOUT.)
DIFFICULTY TO GUESS: EASY
WAS IT TROMBONE? NO, TROUBADOR. AND ONE OF THE Os WAS A ZERO?
AND THERE WAS SOME SYMBOL...
DIFFICULTY TO REMEMBER: HARD
correct horse battery staple
FOUR RANDOM COMMON WORDS
~44 BITS OF ENTROPY
2^44 = 550 YEARS AT 1000 GUESSES/SEC
DIFFICULTY TO GUESS: HARD
THAT'S A BATTERY STAPLE.
CORRECT!
DIFFICULTY TO REMEMBER: YOU'VE ALREADY MEMORIZED IT
THROUGH 20 YEARS OF EFFORT, WE'VE SUCCESSFULLY TRAINED EVERYONE TO USE PASSWORDS THAT ARE HARD FOR HUMANS TO REMEMBER, BUT EASY FOR COMPUTERS TO GUESS.

As an independent Dropbox hackweek project, I thought it'd be fun to build an open source estimator that catches common patterns, and as a corollary, doesn't penalize sufficiently complex passphrases like`correcthorsebatterystaple`. It's now live on dropbox.com/register and available for use on github. Try the demo to experiment and see several example estimations.

The table below compares zxcvbn to other meters. The point isn't to dismiss the others — password policy is highly subjective — rather, it's to give a better picture of how zxcvbn is different.

`qwER43@!`	`Tr0ub4dour&3`	`correcthorsebatterystaple`
Weak	So-so	Great!
Great!	Great!	So-so
Medium	Strong	1 number required

(password not allowed)	(password not allowed)	(password not allowed)
 ✓ Password is perfect!	 ✓ Password is perfect!	 ✓ Password is perfect!
 Weak	 Strong	 Weak
 Strong	 Strong	(password not allowed)
 •••••••• Password strength: Strong	 •••••••••••• Password strength: Strong	 •••••••••••••••••••••••••• Password strength: Weak
 Very strong	 Very strong	 Weak
 Strong	 Strong	 Good

A few notes:

- I took these screenshots on April 3rd, 2012. I needed to crop the bar from the gmail signup form to make it fit in the table, making the difference in relative width more pronounced than on the form itself.
- zxcvbn considers `correcthorsebatterystaple` the strongest password of the 3. The rest either consider it the weakest or disallow it. (Twitter gives about the same score for each, but if you squint, the scores are slightly different.)
- zxcvbn considers `qwER43@!` weak because it's a short QWERTY pattern. It adds extra entropy for each turn and shifted character.
- The PayPal meter considers `qwER43@!` weak but `aaAA11!!` strong. Speculation, but that might be because it detects spatial patterns too.
- Bank of America doesn't allow passwords over 20 characters, disallowing `correcthorsebatterystaple`. Passwords can contain some symbols, but not `&` or `!`, disallowing the other two passwords. eBay doesn't allow passwords over 20 characters either.
- Few of these meters appear to use the naive estimation I opened with; otherwise`correcthorsebatterystaple` would have a high rating from its long length. Dropbox used to add points for each unique lowercase letter, uppercase letter, number, and symbol, up to a certain cap for each group. This mostly has the same only-works-for-brute-force problem, although it also checked against a common passwords dictionary. I don't know the details behind the other meters, but a scoring checklistis another

common approach (which also doesn't check for many patterns).
- I picked *Troubadour* to be the base word of the second column, not *Troubador* as occurs in xkcd, which is an uncommon spelling.

Installation

zxcvbn has no dependencies and works on ie7+/opera/ff/safari/chrome. The best way to add it to your registration page is:

```
<script type="text/javascript" src="zxcvbn-async.js">
</script>
```

`zxcvbn-async.js` is a measly 350 bytes. On `window.load`, after your page loads and renders, it'll load`zxcvbn.js`, a fat 680k (320k gzipped), most of which is a dictionary. I haven't found the script size to be an issue; since a password is usually not the first thing a user enters on a signup form, there's plenty of time to load. Here's a comprehensive rundown of crossbrowser asynchronous script loading.

zxcvbn adds a single function to the global namespace:

```
zxcvbn(password, user_inputs)
```

It takes one required argument, a password, and returns a result object. The result includes a few properties:

```
result.entropy             # bits
result.crack_time          # estimation of actual crack time, in seconds.
result.crack_time_display  # same crack time, as a friendlier string:
                           # "instant", "6 minutes", "centuries", etc.
result.score               # 0, 1, 2, 3 or 4 if crack time is less than
                           # 10**2, 10**4, 10**6, 10**8, Infinity.
                           # (helpful for implementing a strength bar.)
result.match_sequence      # the detected patterns used to calculate entropy.
result.calculation_time    # how long it took to calculate an answer,
                           # in milliseconds. usually only a few ms.
```

The optional `user_inputs` argument is an array of strings that zxcvbn will add to its internal dictionary. This can be whatever list of strings you like, but it's meant for user inputs from other fields of the form, like name and email. That way a password that includes the user's personal info can be heavily penalized. This list is also good for site-specific vocabulary. For example, ours includes `dropbox`.

zxcvbn is written in CoffeeScript. `zxcvbn.js` and `zxcvbn-async.js` are unreadably closure-compiled, but if you'd like to extend zxcvbn and send me a pull request, the README has development setup info.

The rest of this post details zxcvbn's design.

The model

zxcvbn consists of three stages: match, score, then search.

- **match** enumerates all the (possibly overlapping) patterns it can detect. Currently zxcvbn matches against several dictionaries (English words, names and surnames, Burnett's 10,000 common passwords), spatial keyboard patterns (QWERTY, Dvorak, and keypad patterns), repeats (`aaa`), sequences (`123`,`gfedcba`), years from 1900 to 2019, and dates (`3-13-1997`, `13.3.1997`, `1331997`). For all dictionaries, match recognizes uppercasing and common l33t substitutions.

- **score** calculates an entropy for each matched pattern, independent of the rest of the password, assuming the attacker knows the pattern. A simple example: `rrrrr`. In this case, the attacker needs to iterate over all repeats from length 1 to 5 that start with a lowercase letter:

  ```
  entropy = lg(26*5) # about 7 bits
  ```

- **search** is where Occam's razor comes in. Given the full set of possibly overlapping matches, search finds the simplest (lowest entropy) non-overlapping sequence. For example, if the password is `damnation`, that could be analyzed as two words, `dam` and `nation`, or as one. It's important that it be analyzed as one, because an attacker trying dictionary words will crack it as one word long before two. (As an aside, overlapping patterns are also the primary agent behind accidentally tragic domain name registrations, like `childrens-laughter.com` but without the hyphen.)

Search is the crux of the model. I'll start there and work backwards.

Minimum entropy search

zxcvbn calculates a password's entropy to be the sum of its constituent patterns. Any gaps between matched patterns are treated as brute-force "patterns" that also contribute to the total entropy. For example:

```
entropy("stockwell4$eR123698745") == surname_entropy("stockwell") +
                                     bruteforce_entropy("4$eR") +
                                     keypad_entropy("123698745")
```

That a password's entropy is the sum of its parts is a big assumption. However, it's a conservative assumption. By disregarding the "configuration entropy" — the entropy from the number and arrangement of the pieces — zxcvbn is purposely underestimating, by giving a password's structure away for free: It assumes attackers already know the structure (for example, *surname-bruteforce-keypad*), and from there, it calculates how many guesses they'd need to iterate through. This is a significant underestimation for complex structures. Considering`correcthorsebatterystaple`, *word-word-word-word*, an attacker running a program like L0phtCrack or John the Ripper would typically try many simpler structures first, such as *word*, *word-number*, or *word-word*, before reaching *word-word-word-word*. I'm OK with this for three reasons:

- It's difficult to formulate a sound model for structural entropy; statistically, I don't happen to know what structures people choose most, so I'd rather do the safe thing and underestimate.
- For a complex structure, the sum of the pieces alone is often sufficient to give an "excellent" rating. For example, even knowing the *word-word-word-word* structure of `correcthorsebatterystaple`, an attacker would need to spend centuries cracking it.
- Most people *don't* have complex password structures. Disregarding structure only underestimates by a few bits in the common case.

With this assumption out of the way, here's an efficient dynamic programming algorithm in CoffeeScript for finding the minimum non-overlapping match sequence. It runs in *O(n·m)* time for a length-*n* password with *m*(possibly overlapping) candidate matches.

```
# matches: the password's full array of candidate matches.
# each match has a start index (match.i) and an end index (match.j) into
# the password, inclusive.
minimum_entropy_match_sequence = (password, matches) ->
  # e.g. 26 for lowercase-only
  bruteforce_cardinality = calc_bruteforce_cardinality password
  up_to_k = []      # minimum entropy up to k.
  backpointers = [] # for the optimal sequence of matches up to k,
                    # holds the final match (match.j == k).
                    # null means the sequence ends w/ a brute-force char
  for k in [0...password.length]
    # starting scenario to try to beat:
    # adding a brute-force character to the minimum entropy sequence at k-1.
    up_to_k[k] = (up_to_k[k-1] or 0) + lg bruteforce_cardinality
    backpointers[k] = null
    for match in matches when match.j == k
      [i, j] = [match.i, match.j]
      # see if minimum entropy up to i-1 + entropy of this match is less
      # than the current minimum at j.
      candidate_entropy = (up_to_k[i-1] or 0) + calc_entropy(match)
      if candidate_entropy < up_to_k[j]
        up_to_k[j] = candidate_entropy
        backpointers[j] = match

  # walk backwards and decode the best sequence
  match_sequence = []
  k = password.length - 1
  while k >= 0
    match = backpointers[k]
    if match
      match_sequence.push match
      k = match.i - 1
    else
      k -= 1
  match_sequence.reverse()

  # fill in the blanks between pattern matches with bruteforce "matches"
  # that way the match sequence fully covers the password:
  # match1.j == match2.i - 1 for every adjacent match1, match2.
  make_bruteforce_match = (i, j) ->
    pattern: 'bruteforce'
    i: i
    j: j
    token: password[i..j]
    entropy: lg Math.pow(bruteforce_cardinality, j - i + 1)
    cardinality: bruteforce_cardinality
  k = 0
  match_sequence_copy = []
  for match in match_sequence # fill gaps in the middle
    [i, j] = [match.i, match.j]
    if i - k > 0
      match_sequence_copy.push make_bruteforce_match(k, i - 1)
    k = j + 1
    match_sequence_copy.push match
  if k < password.length # fill gap at the end
    match_sequence_copy.push make_bruteforce_match(k, password.length - 1)
  match_sequence = match_sequence_copy

  # or 0 corner case is for an empty password ''
  min_entropy = up_to_k[password.length - 1] or 0
  crack_time = entropy_to_crack_time min_entropy

  # final result object
  password: password
  entropy: round_to_x_digits min_entropy, 3
  match_sequence: match_sequence
  crack_time: round_to_x_digits crack_time, 3
  crack_time_display: display_time crack_time
  score: crack_time_to_score crack_time
```

`backpointers[j]` holds the match in this sequence that ends at password position `j`, or null if the sequence doesn't include such a match. Typical of dynamic programming, constructing the optimal sequence requires starting at the end and working backwards.

Especially because this is running browser-side as the user types, efficiency does matter. To get something up and running I started with the simpler *$O(2^m)$* approach of calculating the sum for every possible non-overlapping subset, and it slowed down quickly. Currently all together, zxcvbn takes no more than a few milliseconds for most passwords. To give a rough ballpark: running Chrome on a 2.4 GHz Intel Xeon, `correcthorsebatterystaple`took about 3ms on average. `coRrecth0rseba++ery9/23/2007staple$` took about 12ms on average.

Threat model: entropy to crack time

Entropy isn't intuitive: How do I know if 28 bits is strong or weak? In other words, how should I go from entropy to actual estimated crack time? This requires more assumptions in the form of a threat model. Let's assume:

- Passwords are stored as salted hashes, with a different random salt per user, making rainbow attacksinfeasible.
- An attacker manages to steal every hash and salt. The attacker is now guessing passwords offline at max rate.
- The attacker has several CPUs at their disposal.

Here's some back-of-the-envelope numbers:

```
# for a hash function like bcrypt/scrypt/PBKDF2, 10ms is a safe lower bound
# for one guess. usually a guess would take longer -- this assumes fast
# hardware and a small work factor. adjust for your site accordingly if you
# use another hash function, possibly by several orders of magnitude!
SINGLE_GUESS = .010 # seconds
NUM_ATTACKERS = 100 # number of cores guessing in parallel.

SECONDS_PER_GUESS = SINGLE_GUESS / NUM_ATTACKERS

entropy_to_crack_time = (entropy) ->
  .5 * Math.pow(2, entropy) * SECONDS_PER_GUESS
```

I added a .5 term because we're measuring the average crack time, not the time to try the full space.

This math is perhaps overly safe. Large-scale hash theft is a rare catastrophe, and unless you're being specifically targeted, it's unlikely an attacker would dedicate 100 cores to your single password. Normally an attacker has to guess online and deal with network latency, throttling, and CAPTCHAs.

Entropy calculation

Up next is how zxcvbn calculates the entropy of each constituent pattern. `calc_entropy()` is the entry point. It's a simple dispatch:

```
calc_entropy = (match) ->
  return match.entropy if match.entropy?
  entropy_func = switch match.pattern
    when 'repeat'     then repeat_entropy
    when 'sequence'   then sequence_entropy
    when 'digits'     then digits_entropy
    when 'year'       then year_entropy
    when 'date'       then date_entropy
    when 'spatial'    then spatial_entropy
    when 'dictionary' then dictionary_entropy
  match.entropy = entropy_func match
```

I gave an outline earlier for how `repeat_entropy` works. You can see the full scoring code on github, but I'll describe two other scoring functions here to give a taste: `spatial_entropy` and `dictionary_entropy`.

Consider the spatial pattern `qwertyhnm`. It starts at `q`, its length is 9, and it has 3 turns: the initial turn moving right, then down-right, then right. To parameterize:

```
s # number of possible starting characters.
  # 47 for QWERTY/Dvorak, 15 for pc keypad, 16 for mac keypad.
L # password length. L >= 2
t # number of turns. t <= L - 1
  # for example, a length-3 password can have at most 2 turns, like "qaw".
d # average "degree" of each key -- the number of adjacent keys.
  # about 4.6 for QWERTY/Dvorak. (g has 6 neighbors, tilda only has 1.)
```

The space of total possibilities is then all possible spatial patterns of length L or less with t turns or less:

$$\lg\left(\sum_{i=2}^{L}\sum_{j=1}^{\min t,i-1}\binom{i-1}{j-1}sd^j\right)$$

(i – 1) choose (j – 1) counts the possible configurations of turn points for a length-*i* spatial pattern with *j* turns. The -1 is added to both terms because the first turn always occurs on the first letter. At each of *j* turns, there's *d*possible directions to go, for a total of d^j possibilities per configuration. An attacker would need to try each starting character too, hence the *s*. This math is only a rough approximation. For example, many of the alternatives counted in the equation aren't actually possible on a keyboard: for a length-5 pattern with 1 turn, "start at q moving left" gets counted, but isn't actually possible.

CoffeeScript allows natural expression of the above:

```
lg = (n) -> Math.log(n) / Math.log(2)

nPk = (n, k) ->
  return 0 if k > n
  result = 1
  result *= m for m in [n-k+1..n]
  result

nCk = (n, k) ->
  return 1 if k == 0
  k_fact = 1
  k_fact *= m for m in [1..k]
  nPk(n, k) / k_fact

spatial_entropy = (match) ->
  if match.graph in ['qwerty', 'dvorak']
    s = KEYBOARD_STARTING_POSITIONS
    d = KEYBOARD_AVERAGE_DEGREE
  else
    s = KEYPAD_STARTING_POSITIONS
    d = KEYPAD_AVERAGE_DEGREE
  possibilities = 0
  L = match.token.length
  t = match.turns
  # estimate num patterns w/ length L or less and t turns or less.
  for i in [2..L]
    possible_turns = Math.min(t, i - 1)
    for j in [1..possible_turns]
      possibilities += nCk(i - 1, j - 1) * s * Math.pow(d, j)
  entropy = lg possibilities
  # add extra entropy for shifted keys. (% instead of 5, A instead of a.)
  # math is similar to extra entropy from uppercase letters in dictionary
  # matches, see the next snippet below.
  if match.shifted_count
    S = match.shifted_count
    U = match.token.length - match.shifted_count # unshifted count
    possibilities = 0
    possibilities += nCk(S + U, i) for i in [0..Math.min(S, U)]
    entropy += lg possibilities
  entropy
```

On to dictionary entropy:

```
dictionary_entropy = (match) ->
  entropy = lg match.rank
  entropy += extra_uppercasing_entropy match
  entropy += extra_l33t_entropy match
  entropy
```

The first line is the most important: The match has an associated **frequency rank**, where words like *the* and *good*have low rank, and words like *photojournalist* and *maelstrom* have high rank. This lets zxcvbn scale the calculation to an appropriate dictionary size on the fly, because if a password contains only common words, a cracker can succeed with a smaller dictionary. This is one reason why xkcd and zxcvbn slightly disagree on entropy for `correcthorsebatterystaple` (45.2 bits vs 44). The xkcd example used a fixed dictionary size of 2^{11}(about 2k words), whereas zxcvbn is adaptive. Adaptive sizing is also the reason `zxcvbn.js` includes entire dictionaries instead of a space-efficient Bloom filter — rank is needed in addition to a membership test.

I'll explain how frequency ranks are derived in the data section at the end. Uppercasing entropy looks like this:

```
extra_uppercase_entropy = (match) ->
  word = match.token
  return 0 if word.match ALL_LOWER
  # a capitalized word is the most common capitalization scheme,
  # so it only doubles the search space (uncapitalized + capitalized):
  # 1 extra bit of entropy.
  # allcaps and end-capitalized are common enough too,
  # underestimate as 1 extra bit to be safe.
  for regex in [START_UPPER, END_UPPER, ALL_UPPER]
    return 1 if word.match regex
  # otherwise calculate the number of ways to capitalize
  # U+L uppercase+lowercase letters with U uppercase letters or less.
  # or, if there's more uppercase than lower (for e.g. PASSwORD), the number
  # of ways to lowercase U+L letters with L lowercase letters or less.
  U = (chr for chr in word.split('') when chr.match /[A-Z]/).length
  L = (chr for chr in word.split('') when chr.match /[a-z]/).length
  possibilities = 0
  possibilities += nCk(U + L, i) for i in [0..Math.min(U, L)]
  lg possibilities
```

So, 1 extra bit for first-letter-uppercase and other common capitalizations. If the uppercasing doesn't fit these common molds, it adds:

$$\lg\left(\sum_{i=1}^{min(U,L)} \binom{U+L}{i}\right)$$

The math for l33t substitution is similar, but with variables that count substituted and unsubstituted characters instead of uppers and lowers.

Pattern matching

So far I covered pattern entropy, but not how zxcvbn finds patterns in the first place. Dictionary match is straightforward: check every substring of the password to see if it's in the dictionary:

```
dictionary_match = (password, ranked_dict) ->
  result = []
  len = password.length
  password_lower = password.toLowerCase()
  for i in [0...len]
    for j in [i...len]
      if password_lower[i..j] of ranked_dict
        word = password_lower[i..j]
        rank = ranked_dict[word]
        result.push(
          pattern: 'dictionary'
          i: i
          j: j
          token: password[i..j]
          matched_word: word
          rank: rank
        )
  result
```

`ranked_dict` maps from a word to its frequency rank. It's like an array of words, ordered by high-frequency-first, but with index and value flipped. l33t substitutions are detected in a separate matcher that uses`dictionary_match` as a primitive. Spatial patterns like `bvcxz` are matched with an adjacency graph approach that counts turns and shifts along the way. Dates and years are matched with regexes. Hit matching.coffee on github to read more.

Data

As mentioned earlier, the 10k password list is from Burnett, released in 2011.

Frequency-ranked names and surnames come from the freely available 2000 US Census. To help zxcvbn not crash ie7, I cut off the surname dictionary, which has a long tail, at the 80th percentile (meaning 80% of Americans have one of the surnames in the list). Common first names include the 90th percentile.

The 40k frequency list of English words comes from a project on Wiktionary, which counted about 29M words across US television and movies. My hunch is that of all the lists I could find online, television and movie scripts will capture popular usage (and hence likely words used in passwords) better than other sources of English, but this is an untested hypothesis. The list is a bit dated; for example, *Frasier* is the 824th most common word.

Conclusion

At first glance, building a good estimator looks about as hard as building a good cracker. This is true in a tautological sort of way if the goal is *accuracy*, because “ideal entropy” — entropy according to a perfect model — would measure exactly how many guesses a given cracker (with a smart operator behind it) would need to take. The goal isn’t accuracy, though. The goal is to give sound password advice. And this actually makes the job a bit easier: I can take the liberty of underestimating entropy, for example, with the only downside of encouraging passwords that are stronger than they need to be, which is frustrating but not dangerous.

Good estimation is still difficult, and the main reason is there’s so many different patterns a person might use. zxcvbn doesn’t catch words without their first letter, words without vowels, misspelled words, n-grams, zipcodes from populous areas, disconnected spatial patterns like `qzwxec`, and many more. Obscure patterns (like Catalan numbers) aren’t important to catch, but for each common pattern that zxcvbn misses and a cracker might know about, zxcvbn overestimates entropy, and that’s the worst kind of bug. Possible improvements:

- zxcvbn currently only supports English words, with a frequency list skewed toward American usage and spelling. Names and surnames, coming from the US census, are also skewed. Of the many keyboard layouts in the world, zxcvbn recognizes but a few. Better country-specific datasets, with an option to choose which to download, would be a big improvement.
- As this study by Joseph Bonneau attests, people frequently choose common phrases in addition to common words. zxcvbn would be better if it recognized “Harry Potter” as a common phrase, rather than a semi-common name and surname. Google’s n-gram corpus fits in a terabyte, and even a good bigram list is impractical to download browser-side, so this functionality would require server-side evaluation and infrastructure cost. Server-side evaluation would also allow a much larger single-word dictionary, such as Google’s unigram set.
- It’d be better if zxcvbn tolerated misspellings of a word up to a certain edit distance. That would bring in many word-based patterns, like skip-the-first-letter. It’s hard because word segmentation gets tricky, especially with the added complexity of recognizing l33t substitutions.

Even with these shortcomings, I believe zxcvbn succeeds in giving better password advice in a world where bad password decisions are widespread. I hope you find it useful. Please fork on github and have fun!

Big thanks to Chris Varenhorst, Gautam Jayaraman, Ben Darnell, Alicia Chen, Todd Eisenberger, Kannan Goundan, Chris Beckmann, Rian Hunter, Brian Smith, Martin Baker, Ivan Kirigin, Julie Tung, Tido the Great, Ramsey Homsany, Bart Volkmer and Sarah Niyogi for helping review this post.