



zxcvbn: Low-Budget Password Strength Estimation

Daniel Lowe Wheeler, *Dropbox Inc.*

<https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/wheeler>

This paper is included in the Proceedings of the
25th USENIX Security Symposium

August 10–12, 2016 • Austin, TX

ISBN 978-1-931971-32-4

Open access to the Proceedings of the
25th USENIX Security Symposium
is sponsored by USENIX

zxcvbn: Low-Budget Password Strength Estimation

Daniel Lowe Wheeler
Dropbox Inc.

Abstract

For over 30 years, password requirements and feedback have largely remained a product of *LUDS*: counts of lower- and uppercase letters, digits and symbols. *LUDS* remains ubiquitous despite being a conclusively burdensome and ineffective security practice.

zxcvbn is an alternative password strength estimator that is small, fast, and crucially no harder than *LUDS* to adopt. Using leaked passwords, we compare its estimations to the best of four modern guessing attacks and show it to be accurate and conservative at low magnitudes, suitable for mitigating online attacks. We find 1.5 MB of compressed storage is sufficient to accurately estimate the best-known guessing attacks up to 10^5 guesses, or 10^4 and 10^3 guesses, respectively, given 245 kB and 29 kB. *zxcvbn* can be adopted with 4 lines of code and downloaded in seconds. It runs in milliseconds and works as-is on web, iOS and Android.

1 Introduction

Passwords remain a key component of most online authentication systems [32], but the quest to replace them [20] is an active research area with a long history of false starts and renewed enthusiasm (recently e.g., [33]). Whatever the future may hold for passwords, we argue that one of the most unusable and ineffective aspects of password authentication as encountered in 2016 truly does belong in the past: composition requirements and feedback derived from counts of lower- and uppercase letters, digits and symbols – *LUDS* for short.

LUDS requirements appear in many incarnations: some sites require digits, others require the presence of at least 3 character classes, some banish certain symbols, and most set varying length minimums and maximums [21, 29, 52]. It is also now commonplace to provide real-time password feedback in the form of visual strength bars and dynamic advice [50, 28]. As with pass-

word requirements, inconsistent *LUDS* calculations tend to lurk behind these feedback interfaces [25].

We review the history of *LUDS* in Section 2 as well as its usability problems and ineffectiveness at mitigating guessing attacks, facts that are also well known outside of the security community [23, 44]. But because it isn't obvious what should go in place of *LUDS*, Section 3 presents a framework for evaluating alternatives such as *zxcvbn*. We argue that anything beyond a small client library with a simple interface is too costly for most would-be adopters, and that estimator accuracy is most important at low magnitudes and often not important past anywhere from 10^2 to 10^6 guesses depending on site-specific rate limiting capabilities.

The Dropbox tech blog presented an early version of *zxcvbn* in 2012 [55]. We've made several improvements since, and Section 4 presents the updated algorithm in detail. At its core, *zxcvbn* checks how common a password is according to several sources – common passwords according to three leaked password sets, common names and surnames according to census data, and common words in a frequency count of Wikipedia 1-grams. For example, if a password is the 55th most common entry in one of these ranked lists, *zxcvbn* estimates it as requiring 55 attempts to be guessed. Section 5.2.3 demonstrates that this simple minimum rank over several lists is responsible for most of *zxcvbn*'s accuracy.

Section 4 generalizes this idea in two ways. First, it incorporates other commonly used password patterns such as dates, sequences, repeats and keyboard patterns. Similar to its core, *zxcvbn* estimates the attempts needed to guess other pattern types by asking: if an attacker knows the pattern, how many attempts would they need to guess the instance? For example, the strength of the QWERTY pattern *zxcvfr* is estimated by counting how many other QWERTY patterns have up to six characters and one turn. Second, Section 4 generalizes beyond matching single patterns by introducing a search heuristic for multi-pattern sequences. That way, C0mpaq999

can be matched as the common password C0mpaq followed by the repeat pattern 999.

We simulate a professional guessing attack in Section 5 by running four attack models in parallel via Ur et al.'s Password Guessability Service [51, 8]. For each password in a test set sampled from real leaks, we take the minimum guess attempts needed over these four models as our conservative gold standard for strength. Section 5 measures accuracy by comparing strength estimations of each password to this gold standard. We investigate two other estimators in addition to `zxcvbn`: the estimator of KeePass Password Safe [5] (hereafter KeePass) as it is the only other non-LUDS estimator studied in [25],¹ and NIST entropy, an influential LUDS estimator reviewed in Section 2.

Our experiments are motivated by two intended uses cases: smarter password composition requirements and strength meters. Given a good estimator, we believe the best-possible strength requirement both in terms of usability and adopter control becomes a minimum check: does it take more than N attempts to guess the password? If not, tell the user their choice is too obvious and let them fix it however they want. For such a policy, an adopter needs confidence that their estimator won't overestimate below their chosen N value. Similarly, smarter strength meters need to know over what range their estimator can be trusted.

Contributions:

- We demonstrate how choice of algorithm and data impacts accuracy of password strength estimation, and further demonstrate the benefit of matching patterns beyond dictionary lookup. We observe that KeePass and NIST substantially overestimate within the range of an online guessing attack and suggest a fix for KeePass.
- We show `zxcvbn` paired with 1.5MB of compressed data is sufficient to estimate the best-known guessing attacks with high accuracy up to 10^5 guesses. We find 245 kB is sufficient to estimate up to 10^4 guesses, and 29 kB up to 10^3 guesses.
- We present the internals of `zxcvbn` in detail, to serve immediately as a LUDS replacement for web and mobile developers, and in the future as a reference point that other estimators can measure against and attempt to beat.
- We present an evaluation framework for low-cost estimators to help future improvements balance security, usability, and ease of adoption.

¹We exclude Google's estimator, a server-side estimator with no details available.

2 Background and Related Work

2.1 LUDS

LUDS has its roots at least as far back as 1985, tracing to the U.S. Defense Department's influential *Password Management Guideline* [12] (nicknamed the Green Book) and NIST's related *Password Usage* of the *Federal Information Processing Standards* series that same year [13]. The Green Book suggested evaluating password strength in terms of guessing space, modeled as $S = A^M$, where S is the maximum guess attempts needed to guess a password, M is the password's length, and A is its alphabet size. For example, a length-8 password of random lowercase letters and digits would have a guessing space of $S = (26 + 10)^8$, and a passphrase of three random words from a 2000-word dictionary would have $S = 2000^3$. S is often expressed in bits as $M \cdot \log_2(A)$, a simple metric that is properly the *Shannon entropy* [47] when every password is assigned randomly in this way.

While this metric works well for machine-generated passwords, NIST's related guideline applied similar reasoning to user-selected passwords at a time before much was known about human password habits. This erroneous randomness assumption persists across the Internet today. For example, consider our pseudocode summary of the metric NIST recommends (albeit with some disclaimers) in its most recent Special Publication 800-63-2 *Electronic Authentication Guideline* of August 2013 [22], commonly referred to as *NIST entropy* (hereafter NIST):

```
1: function NIST_ENTROPY( $p$ ,  $dict$ )
2:    $e \leftarrow 4 + 2 \cdot p[2:8].len + 1.5 \cdot p[9:20].len + p[21:].len$ 
3:    $e \leftarrow e + 6$  if  $p$  contains upper and non-alpha
4:    $e \leftarrow e + 6$  if  $p.len < 20$  and  $p \notin dict$ 
5:   return  $e$ 
```

That is, NIST adds 4 bits for the first character in a password p , 2 bits each for characters 2 through 8, progressively fewer bits for each additional character, 6 more bits if both an uppercase and non-alphabetic character are used – so far, all LUDS. Up to now the 0 in `Passw0rd` would give it a higher entropy than `QJqUbPpA`. NIST recommends optionally adding up to 6 more bits if the password is under 20 characters and passes a dictionary check,² the idea being that longer passwords are likely to be multiword passphrases that don't deserve a bonus. Even then, assuming `Passw0rd` fails the dictionary check and `QJqUbPpA` passes, these sample passwords would oddly have equal scores.

NIST entropy remains influential on password policy. Shay et al. [48] studied Carnegie Mellon University's

²a non-LUDSian detail.

policy migration as part of joining the InCommon Federation and seeking its NIST-entropy-derived Silver Assurance certification [9], to give one notable example.

Whether used for feedback or for requirements, the goal of any LUDS formulation is ultimately to guide users towards less guessable passwords,³ and herein lies the first problem – it’s ineffective. Numerous studies confirm that people use types of characters in skewed distributions [27, 38, 48]: title case, all caps, digit suffixes, some characters more often than others within a class – to give only a small taste. Worse, the most commonly used patterns in passwords cannot be captured by character class counts, such as words, dates, and keyboard patterns. By taking tens of millions of leaked passwords and comparing NIST entropy to the guess order enumeration of a modern password cracker, Weir et al. [53] conclusively demonstrated that even with an added dictionary check and varied parameters, LUDS counts cannot be synthesized into a valid metric for password guessability. In a collaboration with Yahoo, Bonneau [19] found that a six-character minimum length policy exhibited almost no difference in online guessing resistance compared to no length requirement.

The second problem with LUDS is its high usability cost [34]. Any LUDS requirement beyond a low minimum length check necessarily disallows many strong passwords and places a burden on everyone, instead of a subgroup with a known risk of having their password guessed. Florêncio and Herley [29] studied the password policies of 75 American websites and concluded that policy stringency did not correlate with a heightened need for security, but rather with absence of competition and insulation from the consequences of poor usability.

This usability problem is compounded by policy inconsistency among sites. Wang and Wang [52] studied the password composition policies of 50 sites in 2015 (30 from mainland China, the rest mostly American) and found that no two sites enforced the same policy. Bonneau and Preibusch [21] found 142 unique policies in a 2010 survey of 150 high-traffic sites. As a result of these inconsistent policies, people often have to jump through unique hoops upon selecting new passwords [23].

Password feedback is similarly inconsistent across the Internet. Carnavalet and Mannan [25] investigated the registration flows of 18 prominent services spanning multiple countries, platforms, and product domains, and with three exceptions (Google, Dropbox, and KeePass), they found simple but widely inconsistent LUDS-based calculations powering visual feedback, sometimes combined with dictionary checks.

³Proximate goals might include compliance or perception of security, both of which still derive from a notion of guessing resistance in most instances.

2.2 Password Guessing

In their seminal 1979 publication, Morris and Thompson [43] conducted one of the first studies of password habits and detailed the early UNIX history of co-evolving password attacks and mitigations. The decades that followed have seen immense development in password guessing efficiency.

Our gold standard for measuring a password’s *strength* is the minimum attempts needed to guess it over four modern guessing attacks, obtained by running the `min_auto` configuration of Ur et al.’s Password Guessability Service [8] (hereafter PGS), demonstrated in [51] to be a conservative estimate of an experienced and well-resourced professional. We run two cutting-edge attacks from the literature, consisting of a PCFG model [54] with Komanduri’s improvements [36] and a Smoothed Order-5 Markov model [39]. Because our gold standard should be a safe lower bound not just over the theoretical best attacks, but also the best-productized attacks in common use by professionals, we further run Hashcat [2] and John the Ripper [3] mangled dictionary models with carefully tuned rule sets.

Throughout this paper, we will differentiate between *online guessing*, where an attacker attempts guesses through a public interface, and *offline guessing*, where, following a theft of password hashes, an attacker makes guesses at a much higher rate on their own hardware. We recommend [30] for a more detailed introduction to guessing attacks.

2.3 Guessing Resistance

We focus on guessing resistance techniques that influence usability, as opposed to developments in cryptography, abuse detection, and other service-side precautions.

The idea of a *proactive password checker*, a program that offers feedback and enforces policy at composition time, traces to the late 80s and early 90s with pioneering work by Nagle [45], Klein [35], Bishop [17] and Spafford [49]. Eight days after the Morris worm, which spread in part by guessing weak passwords, Nagle presented his *Obvious Password Detector* program that rejected any password that didn’t have a sufficient number of uncommon triplets of characters. Klein focused on dictionary checks with various transformations, such as reversed token lookup, common character substitutions, and stemming, but also recommended LUDS rules including the rejection of all-digit passwords. In his `pwcheck` program, Bishop introduced a concise language for system administrators to formulate composition rules in terms of dictionary lookups, regular expression matching, and relations to personal information. This language is also one of the first to allow customized

user feedback, a precursor to today’s ubiquitous password strength meters. None of these early rule-based systems estimate password strength, making it hard to correctly balance usability and answer whether a password is sufficiently unguessable.

Spafford and others proposed space-efficient dictionary lookup techniques using Bloom filters [49, 40] and decision trees [15, 18]. These approaches similarly do not directly estimate guessing resistance or compare their output to modern guessing attacks, providing instead a binary pass/fail. Yan [56] highlights the need to catch patterns beyond dictionary lookups, something we find modest supporting evidence for in Section 5.2.3.

Castelluccia et al. [24] propose maintaining a production database of character *n-gram* counts in order to model a password’s guessability with an adaptive Markov model. Schechter et al. [46] outline a *count-min sketch* datastructure to allow all passwords that aren’t already too popular among other users. Both of these proposals have the advantage of modeling a service’s unique password distribution. Both aggregate information that, if stolen, aid offline cracking attacks, and both include noise mitigations to reduce that threat. We argue in Section 3 that their respective requirements to maintain and secure custom production infrastructure at scale is too costly for most would-be adopters.

Dell’Amico and Filippone [26] detail a Monte Carlo sampling method that converts a password’s probability as computed by any generative model into an estimate of a cracker’s guessing order when running that model. Given that some of today’s best guessing techniques are built on probabilistic models [39, 54, 36], the benefit of this approach is fast and accurate estimation of guessing order, even up to as high as 2^{80} (10^{24}) guesses. But while the conversion step itself is time- and space-efficient, we haven’t encountered investigations in the literature that limit the size of the underlying probability model to something that would fit inside a mobile app or browser script. Comparing space-constrained probabilistic estimators to today’s best guessing attacks (or perhaps a minimum over several parallel attacks as we do) would be valuable future work. Melicher et al.’s concurrent and independent research on lean estimation with Recurrent Neural Networks is quite promising [42].

Turning to open-source industry contributions, *zxcvbn* and *KeePass* [5] were originally designed for password strength feedback, but we consider them here for policy enforcement as well. Industry adoption of *zxcvbn* is growing, currently deployed by Dropbox, Stripe, Wordpress, Coinbase, and others. *KeePass* (reviewed in [25]) matches several common patterns including a dictionary lookup with common transformations, then applies an *optimal static entropy encoder* documented in their help center [4] to search for the

simplest set of candidate matches covering a password. We extracted *KeePass* into a stand-alone command-line utility such that we could compare it against realistic guessing attacks in Section 5.

Telepathwords [37] offers some of the best real-time password feedback currently available. It employs a client-server architecture that is hosted as a stand-alone site, and does not output a guess attempt estimate or equivalent, so we do not evaluate it as a candidate LUDS alternative.

Ur et al. [50] and Egelman et al. [28] studied the effect of strength meters on password composition behavior. The consensus is that users do follow the advice of these meters for accounts of some importance; however, both studies employed LUDS metrics to power their meters as is common in the wild, conditionally with an added dictionary check in the case of [50]. Our aim is to provide strength meters with more accurate underlying estimation.

3 Evaluation Framework

While the problems of LUDS are well understood, it isn’t obvious what should go in its place. We motivate some of the important dimensions and reference two LUDS methods for comparison: NIST as well *3class8* – the easy-to-adopt requirement that a password contain 8 or more characters of at least 3 types.

It should be no harder than LUDS to adopt

In the wider scheme of password authentication, composition policy and feedback are small details. Bonneau and Preibusch [21] demonstrated that the big details – cryptography and rate-limiting, for example – are commonly lacking throughout the Internet with little economic incentive to do better. To then have a starting chance of widespread adoption, a LUDS alternative cannot be harder than LUDS to adopt. We believe this realistically eliminates alternatives that require hosting and scaling special infrastructure from mainstream consideration, whereas small client libraries with simple interfaces are more viable. To give an example, the following is working web integration code for a policy that disallows passwords guessable in under 500 attempts according to *zxcvbn*:

```
var zxcvbn = require('zxcvbn');
var meets_policy = function(password) {
  return zxcvbn(password).guesses > 500;
};
```

This sample assumes a CommonJS module interface. For comparison, Appendix A lists our implementation of *3class8* back-to-back with two other *zxcvbn* integration options.

It should only burden at-risk users

Users face many threats in addition to guessing attacks, including phishing, keylogging, and credential reuse attacks. Because guessing attacks often rank low on a user's list of worries, short and memorable password choices are often driven by rational cost-benefit analysis as opposed to ignorance [31]. To encourage less guessable choices, a LUDS alternative must accept this reality by imposing as few rules as possible and burdening only those facing a known guessing risk. As examples, words should be recognized and weighted instead of, as with space-efficient dictionary lookups, rejected. All-digit and all-lowercase passwords – respectively the most common password styles in mainland China and the U.S. according to a comprehensive study [38] – should similarly be weighted instead of rejected via blanket rules. `3c1ass8` is a prime offender in this category.

While underestimation harms usability, overestimation is arguably worse given an estimator's primary goal of mitigating guessing attacks. In Section 5 we measure accuracy and overestimation as separate quantities in our comparison of alternative estimators. We find KeePass and NIST tend to overestimate at lower magnitudes.

It should estimate guessing order

The security community's consensus definition of a password's strength is *the number of attempts that an attacker would need in order to guess it* [26]. Strength estimators should thus estimate this guessing order directly, versus an entropy, percentage, score, or binary pass/fail. This detail provides adopters with an intuitive language for modeling guessing threats and balancing usability. An alternative should further measure its estimations against real guessing attacks and communicate its accuracy bounds. For example, given enough samples, [26] is accurate up to the highest feasible guessing ranges, whereas with 1.5MB of data, `zxcvbn` is only highly accurate up to 10^5 guesses.

It should be accurate at low magnitudes

Online guessing attacks are a threat that password authentication systems must defend against continually. While rate limiting and abuse detection can help, passwords guessable in, say, 10 to 100 guesses could remain vulnerable to capable adversaries over time. As we show in Section 5.2, NIST greatly overestimates password strength at low magnitudes. Similarly, `3c1ass8` permits obvious choices such as `Password1!`. A strict improvement over LUDS in terms of security, then, is to improve accuracy at low guessing magnitudes.

Past an online guessing threshold, the benefit of accurate estimation becomes more situation-dependent. Referencing the *online-offline chasm* of [30], the added security benefit from encouraging or requiring stronger passwords might only start to appear after many orders of

magnitude past an online guessing cutoff, indicating a substantial usability-security trade-off that often won't be justified. While we focus on online attack mitigation, key stretching techniques such as Argon2 [16] can further render offline attacks unfeasible at higher guessing magnitudes.

For the remainder of this paper, we will use 10^6 guesses as our approximate online attack cutoff, citing the back-of-the-envelope upper limit estimation in [30] for a depth-first online attack (thus also bounding an online trawling attack). By studying leaked password distributions, [30] also points out that an attacker guessing in optimal order would face a reduction in success rate by approximately 5 orders of magnitude upon reaching 10^6 guesses.

While we use 10^6 as an online cutoff for safe and simple analysis in Section 5, we recognize that an upper bound on online guessing is highly dependent on site-specific capabilities, and that some sites will be able to stop an online attack starting at only a few guesses. This motivates our next item:

It should have an adjustable size

An estimator's accuracy is greatly dependent on the data it has available. Adopters should be given control over this size / accuracy trade-off. Some might want to bundle an estimator inside their app, selecting a smaller size. We expect most will want to asynchronously download an estimator in the background on demand, given that password creation only happens once per user and typically isn't the first step in a registration flow. Current bandwidth averages should factor into this discussion: a South Korean product might tolerate a gzipped-5MB estimator (downloadable in 2 seconds at 20.5Mbps national average in Q3 2015 [14]⁴), whereas 1.5MB is a reasonable global upper bound in 2016 (2.3 seconds at 5.1Mbps Q3 2015 global default). Need should also factor in: a site that is comfortable in its rate-limiting might only need accurate estimation up to 10^2 guesses.

4 Algorithm

We now present the internals of `zxcvbn` in detail. Sections 4.2 and 4.3 explain how common token lookup and pattern matching are combined into a single guessing model. Section 4.4 is primarily about speed, providing a fast algorithm for finding the simplest set of matches covering a password. We start with a high-level sketch.

4.1 Conceptual Outline

`zxcvbn` is non-probabilistic, instead heuristically estimating a guessing attack directly. It models passwords

⁴We cite figures from Akamai's State of the Internet series.

as consisting of one or more concatenated patterns. The 2012 version of `zxcvbn` assumes the guesser knows the pattern structure of the password it is guessing, with bounds on how many guesses it needs per pattern. For example, if a password consists of two top-100 common words, it models an attacker who makes guesses as concatenations of two words from a 100-word dictionary, calculating 100^2 as its worst-case guess attempt estimate.

To help prevent overly complex matching, `zxcvbn` now loosens the 2012 assumption by instead assuming the attacker knows the patterns that make up a password, but not necessarily how many or in which order. To illustrate the difference, compare the respective 2012 and 2016 analyses of `jessiah03`:

```
jess(name) i(word) ah0(surname) 3(bruteforce)
jessia(name) h03(bruteforce)
```

The 2012 version has no bias against long pattern sequences, matching `i` and `jess` as common words. (`jessia` is in the common surname dictionary at about rank 3.5k, `jess` is at about rank 440, and `jessiah` is in neither dictionary.) To give another example, the random string `3vMs3o7B7eTo` is now matched as a single brute-force region by `zxcvbn`, but as a 5-pattern sequence by the 2012 version, including `7eT` as a l33ted “let.”

To formalize this difference in behavior, at a high level, both versions consist of three phases: match, estimate and search. Given a plaintext password input, the pattern matching phase finds a set \mathcal{S} of overlapping matches. For example, given `lenovo1111` as input, this phase might return `lenovo` (password token), `eno` (English “one” backwards), `no` (English), `no` (English “on” backwards), `1111` (repeat pattern), and `1111` (Date pattern, `1/1/2011`). Next, the estimation phase assigns a guess attempt estimation to each match independently. If `lenovo` is the 11007th most common password in one of our password dictionaries, it’ll be assigned 11007, because an attacker iterating through that dictionary by order of popularity would need that many guesses before reaching it. The final phase is to search for the sequence of non-overlapping adjacent matches S drawn from \mathcal{S} such that S fully covers the password and minimizes a total guess attempt figure. In this example, the search step would return `[lenovo (token), 1111 (repeat)]`, discarding the date pattern which covers the same substring but requires more guesses than the repeat. `zxcvbn`’s formalized assumption about what an attacker knows is represented by the following search heuristic:

$$\operatorname{argmin}_{S \subseteq \mathcal{S}} D^{|S|-1} + |S|! \prod_{m \in S} m.\text{guesses} \quad (1)$$

$|S|$ is the length of the sequence S , and D is a constant. The intuition is as follows: if an attacker knows the pattern sequence with bounds on how many guesses needed

for each pattern, the Π term measures how many guesses they would need to make in the worst case. This Π term, by itself, is the heuristic employed by the 2012 version. With the added $|S|!$ term, the guesser now knows the number of patterns in the sequence but not the order. For example, if the password contains a common word c , uncommon word u , and a date d , there are $3!$ possible orderings to try: `cud`, `ucd`, etc.

The $D^{|S|-1}$ term attempts to model a guesser who additionally doesn’t know the length of the pattern sequence. Before attempting length- $|S|$ sequences, `zxcvbn` assumes that a guesser attempts lower-length pattern sequences first with a minimum of D guesses per pattern, trying a total of $\sum_{l=1}^{|S|-1} D^l \approx D^{|S|-1}$ guesses for sufficiently large D . For example, if a password consists of the 20th most common password token t with a digit d at the end – a length-2 pattern sequence – and the attacker knows the $D = 10000$ most common passwords, and further, td is not in that top-10000 list (otherwise it would have been matched as a single token), the D^1 term models an attacker who iterates through those 10000 top guesses first before moving on to two-pattern guessing. While an attacker might make as few as 10 guesses for a single-digit pattern or as many as tens of millions of guesses iterating through a common password dictionary, we’ve found $D = 1000$ to $D = 10000$ to work well in practice and adopt the latter figure for `zxcvbn`.

In practical terms, the additive D penalty and multiplicative $|S|!$ penalty address overly complex matching in different ways. When two pattern sequences of differing length have near-equal Π terms, the $|S|!$ factor biases towards the shorter sequence. The D term biases against long sequences with an overall low Π term.

4.2 Matching

The matching phase finds the following patterns:

pattern	examples
<i>token</i>	logitech l0giT3CH ain’t parliamentarian 1232323q
<i>reversed</i>	DrowssaP
<i>sequence</i>	123 2468 jklm ywusq
<i>repeat</i>	zzz ababab l0giT3CHl0giT3CH
<i>keyboard</i>	qwertyuio qAzxcde3 diueoa
<i>date</i>	7/8/1947 8.7.47 781947 4778 7-21-2011 72111 11.7.21
<i>bruteforce</i>	x\$JQhMzt

The token matcher lowercases an input *password* and checks membership for each substring in each frequency-ranked dictionary. Additionally, it attempts

each possible l33t substitution according to a table. An input @BA1one is first lowercased to @ba1one. If the l33t table maps @ to a and 1 to either i or l, it tries two additional matches by subbing [@->a, 1->i] and [@->a, 1->l], finding abalone with the second substitution.

Taking a cue from KeePass, sequence matching in zxcvbn looks for sequences where each character is a fixed Unicode codepoint distance from the last. This has two advantages over the hardcoded sequences of the 2012 version. It allows skipping, as in 7531, and it recognizes sequences beyond the Roman alphabet and Arabic numerals, such as Cyrillic and Greek sequences. Unicode codepoint order doesn't always map directly to human-recognizable sequences; this method imperfectly matches Japanese kana sequences as one example.

The repeat matcher searches for repeated blocks of one or more characters, a rewrite of the 2012 equivalent, which only matched single-character repeats. It tries both greedy $(.+)\1+$ and lazy $(.+?)\1+$ regular expressions in search of repeated regions spanning the most characters. For example, greedy beats lazy for aabaab, recognizing (aab) repeated over the full string vs (a) repeated over aa, whereas lazy beats greedy for aaaaa, matching (a) spanning 5 characters vs (aa) spanning 4. The repeat matcher runs a match-estimate-search recursively on its winning repeated unit, such that, for example, repeated words and dates are identified.

The keyboard matcher runs through *password* linearly, looking for chains of adjacent keys according to each of its keyboard adjacency graphs. These graphs are represented as a mapping between each key to a clockwise positional list of its neighbors. The matcher counts chain length, number of turns, and number of shifted characters. On QWERTY, zxcvfr\$321 would have length 10, 2 turns, and 2 shifted characters. QWERTY, DVORAK, and Windows and Mac keypad layouts are included by default. Additional layouts can be prepackaged or dynamically added.

Date matching considers digit regions of 4 to 8 characters, checks a table to find possible splits, and attempts a day-month-year mapping for each split such that the year is two or four digits, the year isn't in the middle, the month is between 1 and 12 inclusive, and the day is between 1 and 31 inclusive. For example, a six-digit sequence 201689 could be broken into 2016-8-9, 20-16-89, or 2-0-1689. The second candidate would be discarded given no possible month assignment, and the third discarded because 0 is an invalid day and month. Given multiple valid splits, the choice with year closest to a reference year of 2016 wins. Two-digit years are matched as 20th- or 21st-century years, depending on whichever is closer to 2016. For ease of portability, date matching does not filter improper Gregorian dates; for example, it allows Feb. 29th on a non-leap year.

4.3 Estimation

Next, a guess attempt estimate *guesses* is determined for each match $m \in \mathcal{S}$. The guiding heuristic is to ask: if an attacker knows the pattern, how many guesses might they need to guess the instance? Green Book-style guessing space calculations then follow, but for patterns instead of random strings, where a guesser attempts simpler or more likely patterns first.

For tokens, we use the frequency rank as the estimate, because an attacker guessing tokens in order of popularity would need at least that many attempts. A reversed token gets a doubled estimate, because the attacker would then need to try two guesses (normal and reversed) for each token. A conservative factor of 2 is also added for an obvious use of uppercase letters: first-character, last-character, and all caps. The capitalization factor is otherwise estimated as

$$\frac{1}{2} \sum_{i=1}^{\min(U,L)} \binom{U+L}{i} \quad (2)$$

where U and L are the number of uppercase and lowercase letters in the token. For example, to guess paSsw0rd, an attacker would need to try a guessing space of 8 different single-character capitalizations plus 28 different two-character capitalizations. The $1/2$ term converts the total guessing space into an average attempts needed, assuming that each capitalization scheme is equally likely – this detail could be improved by better modeling observed distributions of capitalization patterns in leaked password corpora. The $\min()$ term flips the lowercasing game into an uppercasing game when there are more upper- than lowercase letters, yielding 8 for PAsWORD.

Guesses for keyboard patterns are estimated as:

$$\frac{1}{2} \sum_{i=1}^L \sum_{j=1}^{\min(T,i-i)} \binom{i-1}{j-1} S D^j \quad (3)$$

where L is the length of the pattern, T is the number of terms, D is the average number of neighbors per key (a tilde has one neighbor on QWERTY, the 'a' key has four) and S is the number of keys on the keyboard. For T turns throughout a length L keyboard pattern, we assume a guesser attempts lower-length, lower-turn patterns first, starting at length 2. The binomial term counts the different configuration of turning points for a length- i pattern with j turns, with -1 added to each term because the first turn is defined to occur on the first character. The $\min()$ term avoids considering more turns than possible on a lower-length pattern. The sequence might have started on any of S keys and each turn could have gone any of D ways on average, hence the $S \cdot D^j$. Equation 3 estimates about 10^3 guesses for kjhgfdsa on QWERTY and 10^6

guesses for `kjhgt543`. Shifted keys in the pattern add a factor according to expression 2, where L and U become shifted and unshifted counts.

Repeat match objects consist of a base repeated n times, where a recursive match-estimate-search step previously assigned a number of guesses g to the base. Repeat guess attempts are then estimated as $g \cdot n$. For example, `nownownow` is estimated as requiring 126 guesses: `now` is at rank 42 in the Wiktionary set, times 3.

Sequences are scored according to $s \cdot n \cdot |d|$, where s is the number of possible starting characters, n is the length, and d is the codepoint delta (e.g., -2 in 9753). s is set to a low constant 4 for obvious first choices like 1 and Z, set to 10 for other digits, or otherwise 26, an admittedly Roman-centric default that could be improved.

For dates, we assume guessers start at 2016 and guess progressively earlier or later dates, yielding a ballpark of $365 \cdot |2016 - year|$

Finally, bruteforce matches of length l are assigned a constant $C = 10$ guesses per character, yielding a total estimate of C^l . The 2012 version performs a guessing space calculation, treating bruteforce regions as random, and determines a cardinality C that adds 26 if any lowercase letters are present, 26 if uppercase, 10 if digits, and 33 for one or more symbols. This dramatically overestimates the common case, for example a token that isn't in the dictionary. The 2012 version scores `Teiubesc` (Romanian for "I love you") as $(26 + 26)^8 \approx 10^{14}$, whereas `zxcvbn` now estimates it 6 orders of magnitude lower at 10^8 . (Thanks to the addition of RockYou'09 data, it also matches it as a common password at rank 10^4).

4.4 Search

Given a string *password* and a corresponding set of overlapping matches \mathcal{S} , the last step is to search for the non-overlapping adjacent match sequence S that covers *password* and minimizes expression (1). We outline a dynamic programming algorithm that efficiently accomplishes this task. The idea is to iteratively find the optimal length- l sequence of matches covering each length- k character prefix of *password*. It relies on the following initial state:

```

1:  $n \leftarrow password.length$ 
2:  $\mathcal{B}_{opt} \leftarrow [] \times n$ 
3:  $\Pi_{opt} \leftarrow [] \times n$ 
4:  $l_{opt} \leftarrow 0$ 
5:  $g_{opt} \leftarrow null$ 

```

\mathcal{B}_{opt} is a backpointer table, where $\mathcal{B}_{opt}[k][l]$ holds the ending match in the current optimal length- l match sequence covering the length- k prefix of *password*. $\Pi_{opt}[k][l]$ correspondingly holds the product term in ex-

pression (1) for that sequence. When the algorithm terminates, g_{opt} holds the optimum guesses figure and l_{opt} holds the length of the corresponding optimal sequence. Note that if no length- l sequence exists such that it scores lower than every alternative sequence with fewer matches covering the same k -prefix, then $l \notin \mathcal{B}_{opt}[k]$.

Each match object m has a guess value $m.guesses$ and covers a substring of *password* at indices $m.i$ and $m.j$, inclusive. The search considers one character of *password* at a time, at position k , and for each match m ending at k , evaluates whether adding m to any length- l optimal sequence ending just before m (at $m.i - 1$) leads to a new candidate for the optimal match sequence covering the prefix up to k :

```

1: function SEARCH( $n, \mathcal{S}$ )
2:   for  $k \in 0$  to  $n - 1$ 
3:      $g_{opt} \leftarrow \infty$ 
4:     for  $m \in \mathcal{S}$  when  $m.j = k$ 
5:       if  $m.i > 0$ 
6:         UPDATE( $m, l + 1$ ) for  $l \in \mathcal{B}_{opt}[m.i - 1]$ 
7:       else
8:         UPDATE( $m, 1$ )
9:       BF_UPDATE( $k$ )
10:  return UNWIND( $n$ )

```

Instead of including a bruteforce match object in \mathcal{S} for every $O(n^2)$ substring in *password*, bruteforce matches are considered incrementally by BF_UPDATE:

```

1: function BF_UPDATE( $k$ )
2:    $m \leftarrow$  bruteforce from 0 to  $k$ 
3:   UPDATE( $m, 1$ )
4:   for  $l \in \mathcal{B}_{opt}[k - 1]$ 
5:     if  $\mathcal{B}_{opt}[k - 1][l]$  is bruteforce
6:        $m \leftarrow$  bruteforce from  $\mathcal{B}_{opt}[k - 1][l].i$  to  $k$ 
7:       UPDATE( $m, l$ )
8:     else
9:        $m \leftarrow$  bruteforce from  $k$  to  $k$ 
10:    UPDATE( $m, l + 1$ )

```

That is, at each index k , there are only three cases where a bruteforce match might end an optimal sequence: it might span the entire k -prefix, forming a length-1 sequence, it might extend an optimal bruteforce match ending at $k - 1$, or it might start as a new single-character match at k . Note that given the possibility of expansion, it is always better to expand by one character than to append a new bruteforce match, because either choice would contribute equally to the Π term, but the latter would increment l .

The UPDATE helper computes expression (1) and updates state if a new minimum is found. Thanks to the Π_{opt} table, it does so without looping:

```

1: function UPDATE( $m, l$ )
2:    $\Pi \leftarrow m.guesses$ 
3:   if  $l > 1$ 
4:      $\Pi \leftarrow \Pi \times \Pi_{opt}[m.i-1][l-1]$ 
5:    $g \leftarrow D^{l-1} + l! \times \Pi$ 
6:   if  $g < g_{opt}$ 
7:      $g_{opt} \leftarrow g$ 
8:      $l_{opt} \leftarrow l$ 
9:      $\Pi_{opt}[k][l] \leftarrow \Pi$ 
10:     $\mathcal{B}_{opt}[k][l] \leftarrow m$ 

```

At the end, UNWIND steps through the backpointers to form the final optimal sequence:

```

1: function UNWIND( $n$ )
2:    $S \leftarrow []$ 
3:    $l \leftarrow l_{opt}$ 
4:    $k \leftarrow n - 1$ 
5:   while  $k \geq 0$ 
6:      $m \leftarrow \mathcal{B}_{opt}[k][l]$ 
7:      $S.prepend(m)$ 
8:      $k \leftarrow m.i - 1$ 
9:      $l \leftarrow l - 1$ 
10:  assert  $l = 0$ 
11:  return  $S$ 

```

Each match $m \in \mathcal{S}$ is considered only once during the search, yielding a runtime of $O(l_{max} \cdot (n + |\mathcal{S}|))$, where l_{max} is the maximum value of l_{opt} over each k iteration. In practice, l_{max} rarely exceeds 5, and this method rapidly terminates even for passwords of hundreds of characters and thousands of matches.

4.5 Deployment

zxcvbn is written in CoffeeScript and compiled via an npm build flow into both a server-side CommonJS module and a minified browser script. The ranked token lists take up most of the total library size: each is represented as a sorted comma-separated list of tokens which then get converted into an object, mapping tokens to their ranks. The browser script is minified via UglifyJS2 with instructions on how to serve as gzipped data.

zxcvbn works as-is on most browsers and javascript server frameworks. Because iOS and Android both ship with javascript interpreters, zxcvbn can easily interface with most mobile apps as well. JSContext on iOS7+ or UIWebView for legacy support both work well. Running javascript with or without a web view works similarly on Android.

Dropbox uses zxcvbn for feedback and has never enforced composition requirements other than a 6-character minimum. For those implementing requirements, we suggest a client-side soft enforcement for simplicity,

such as a submit button that is disabled until the requirement is met. Because different versions and ports give slightly different estimates, we suggest those needing server-side validation either skip client-side validation or make sure to use the exact same build across their server and various clients. zxcvbn ports exist for Java, Objective-C, Python, Go, Ruby, PHP, and more.

4.6 Limitations

zxcvbn doesn't model interdependencies between patterns, such as common phrases and other collocations. However, its ranked password dictionaries include many phrases as single tokens, such as `opensesame`. It only matches common word transformations that are easy to implement given limited space; it doesn't match words with deleted letters and other misspellings, for example. Unmatched regions are treated equally based on length; the English-sounding made-up word `novanoid` gets the same estimate as a length-8 random string, and unmatched digits and symbols are treated equally even though some are more common than others.

5 Experiments

We investigate how choice of algorithm and dataset impacts the estimation accuracy of a realistic guessing attack. We also show the impact of matching patterns beyond token lookup. Our experiments employ a test set of 15k passwords from the RockYou'09 leak [7]. Appendix C includes the same analysis on a 15k sample from the Yahoo'12 leak [11]. We close the Section with runtime benchmarks for zxcvbn.

5.1 Methodology

Algorithms and Data

The algorithms we selected for our experiment – NIST, KeePass, and zxcvbn – estimate guess attempts or equivalent (excludes [49, 17]) and can operate without a backing server (excludes [46, 24]).

For our password strength gold standard, as introduced in Section 2.2, we ran the `min_auto` configuration of PGS [8] with the same training data found to be most effective in [51]. The PGS training data (roughly 21M unique tokens) consists of the RockYou'09 password leak (minus a randomly sampled 15k test set), Yahoo'12 leak (minus a similar 15k test set), MySpace'06 leak, 1-grams from the Google Web Corpus, and two English dictionaries. To make brute force guessing attacks infeasible, the 15k test sets are sampled from the subset of passwords that contain at least 8 characters. Of the four attacks, we ran the Markov attack up through 10^{10} guesses, John the Ripper and Hashcat up through 10^{13} ,

and PCFG up to 10^{14} . Detailed specifics can be found in [51].

While our test data is distinct from our training data, it is by design that both include samples from the same RockYou'09 distribution; our aim is to simulate an attacker with knowledge of the distribution they are guessing. While a real attacker wouldn't have training data from their target distribution, they might be able to tailor their attack by deriving partial knowledge – common locales and other user demographics (RockYou includes many Romanian-language passwords in addition to English), site-specific vocabulary (game terminology, say), and so on.

Our estimators are given ranked lists of common tokens as their training data, with one separately ranked list per data source. NIST and KeePass do not make use of rank, instead performing membership tests on a union of their lists. Rather than attempting to precisely match the training sources supplied to PGS, our estimator sources more closely match those used in the current production version of `zxcvbn`. For example, in the spirit of free software, we avoid the Google Web Corpus which is only available through a license via the Linguistic Data Consortium. Instead of counting top passwords from the MySpace'06 leak, our estimators use the Xato'15 corpus which is over 200 times bigger.

In all, we count top tokens from the PGS training portion of RockYou'09 and Yahoo'12 (test sets are excluded from the count), Xato'15, 1-grams from English Wikipedia, common words from a Wiktionary 29M-word frequency study of US television and film [10], and common names and surnames from the 1990 US Census [1]. Appendix B has more details on our data sources and algorithm implementations.

We experiment with three estimator training set sizes by truncating the frequency lists at three points: 100k (1.52 MB of gzipped storage), 10k (245 kB), and 1k (29.3 kB). In the 10k set, for example, each ranked list longer than 10k tokens is cut off at that point.

Metrics

When PGS is unable to guess a password, we exclude it from our sample set S . On each sampled password $x_i \in S$, we then measure an algorithm's estimation error by computing its order-of-magnitude difference Δ_i from PGS,

$$\Delta_i = \log_{10} \frac{g_{alg}(x_i)}{g_{pgs}(x_i)} \quad (4)$$

where g_{alg} is the guess attempt estimate of the algorithm and g_{pgs} is the minimum guess order of the four PGS guessing attacks. For example, $\Delta_i = -2$ means the algorithm underestimated guesses by 2 orders of magnitude compared to PGS for password x_i .

We compare PGS to the estimator algorithms in three ways. First, to give a rough sense of the shape of estimator accuracy, we show log-log scatter plots spanning from 10^0 to 10^{15} guesses, with g_{pgs} on the x axis, g_{alg} on the y axis, and a point for every $x_i \in S$. Second, we show the distribution of Δ_i as a histogram by binning values to their nearest multiple of .5, corresponding to half-orders of magnitude. Third, we calculate the following summary statistics:

$$|\Delta| = \frac{1}{|S|} \sum_{i \in S} |\Delta_i| \quad (5)$$

$$\Delta^+ = \frac{1}{|S|} \sum_{i \in S} \begin{cases} \Delta_i & \text{if } \Delta_i \geq 0 \\ 0 & \text{if } \Delta_i < 0 \end{cases} \quad (6)$$

$|\Delta|$ gives a sense of accuracy, equally penalizing under- and overestimation. Δ^+ measures overestimation. Fewer and smaller overestimations improve (reduce) this metric. We calculate summary statistics within an online range $g_{pgs} < 10^6$ and separately higher magnitudes.

One comparison challenge is that KeePass and NIST output entropy as bits, whereas we want to compare algorithms in terms of guesses. While commonplace among password strength estimators, including the 2012 version of `zxcvbn`, it is mathematically improper to apply entropy, a term that applies to distributions, to individual events. Neither KeePass nor NIST formalize the type of entropy they model, so we assume that n bits of strength means guessing the password is equivalent to guessing a value of a random variable X according to

$$n = H(X) = -\sum_i p(x_i) \log_2 p(x_i) \quad (7)$$

Assuming the guesser knows the distribution over X and tries guesses x_i in decreasing order of probability $p(x_i)$, a lower bound on the expected number of guesses $E[G(X)]$ can be shown [41] to be:

$$E[G(X)] \geq 2^{H(X)-2} + 1 \quad (8)$$

provided that $H(X) \geq 2$. We use this conservative lower bound to convert bits into guesses. Had we additionally assumed a uniform distribution, our expected guesses would be $2^{H(X)-1}$, a negligible difference for our logarithmic accuracy comparisons.

5.2 Results

Of the RockYou 15k test set, PGS cracked 39.68% within our online guessing range of up to 10^6 guesses and 52.65% above 10^6 , leaving 7.67% unguessed.

5.2.1 Choice of Algorithm

Figures 1-3 give a sense of how algorithm choice affects guess attempt estimation. The solid diagonal corresponds to $\Delta = 0$, indicating estimator agreement with

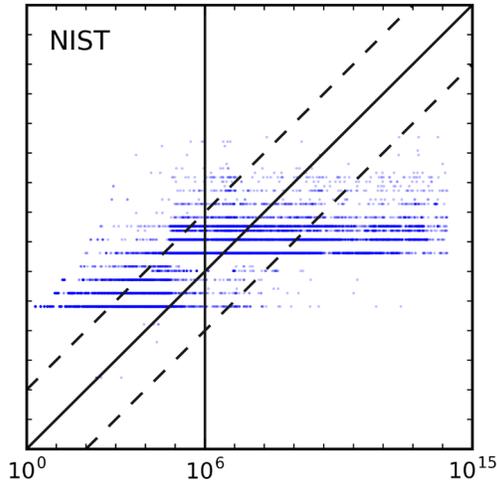


Figure 1: PGS (x axis) vs. NIST (y axis), 100k token set. Points on the solid diagonal indicate agreement between PGS and NIST ($\Delta_i = 0$). Points above the solid diagonal indicate overestimation. Points above the top dashed diagonal indicate overestimation by more than two orders of magnitude ($\Delta_i > 2$).

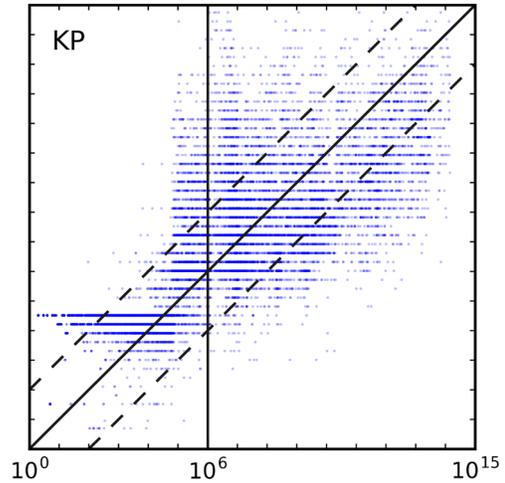


Figure 2: PGS (x) vs. KeePass (y), 100k token set.

PGS. Points above the top / below the bottom dotted lines over/underestimate by more than 2 orders of magnitude ($\Delta_i > 2$ above the top line, $\Delta_i < -2$ below the bottom line). Points to the left of $g_{pgs} = 10^6$ indicate samples potentially susceptible to online guessing as argued in Section 3.

NIST and KeePass both exhibit substantial horizontal banding in the low online range. Figure 2 shows that a KeePass estimate of about $10^{4.5}$ can range anywhere from about 10^{-25} to 10^6 PGS guesses. Figure 1 shows that NIST has a similar band at about $10^{4.8}$, and that NIST tends to overestimate in the online range and lean towards underestimation at higher magnitudes. In Table 1 we measured NIST and KeePass to be respectively off by $|\Delta| = 1.81$ and 1.43 orders of magnitude on average within the online range. We conclude neither are suitable for estimating online guessing resistance; however, we expect KeePass could fix its low-order banding problem by incorporating token frequency rank instead of a fixed entropy for each dictionary match.

Figure 3 demonstrates that zxcvbn grows roughly linear with PGS up until about 10^5 , corresponding to the maximum rank of the 100k token set. Both zxcvbn and KeePass suffer from a spike in overestimation approximately between 10^5 and 10^7 . We speculate this is because PGS is trained on 21M unique tokens and, in one attack, tries all of them in order of popularity before moving onto more complex guessing. Hence, the greatest overestimation in both cases happens between the estimator dictionary cutoff and PGS dictionary cutoff, high-

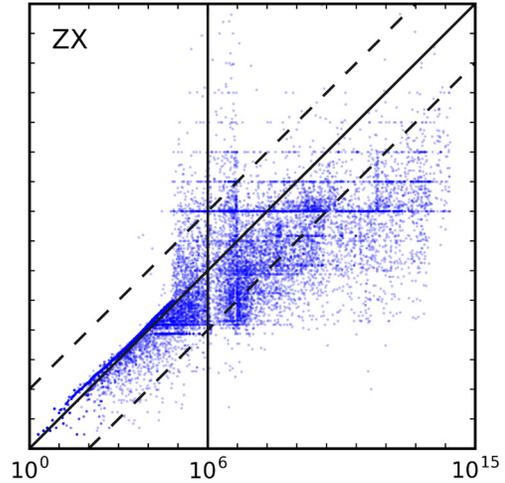


Figure 3: PGS (x) vs. zxcvbn (y), 100k token set.

lighting the sensitivity of estimator dictionary size.

The horizontal banding at fixed orders of magnitude in zxcvbn corresponds to brute-force matches where no other pattern could be identified. Detailed in Section 4, zxcvbn rewards 10^l guesses for length- l unmatched regions. zxcvbn has comparable but slightly worse $|\Delta|$ and Δ^+ than NIST past the online range. Given both have a low Δ^+ , this primarily demonstrates a usability problem at higher magnitudes (overly harsh feedback).

Figure 4 counts and normalizes Δ_i in bin multiples of .5, demonstrating that zxcvbn is within $\pm .25$ orders of magnitude of PGS about 50% of the time within the online range. The sharp drop-off to the right indicates infrequent overestimation in this range. Figure 4 also shows that, within the online range, NIST and KeePass accuracy could be improved by respectively dividing their estimates by about 10^2 and 10^1 guesses; however, both

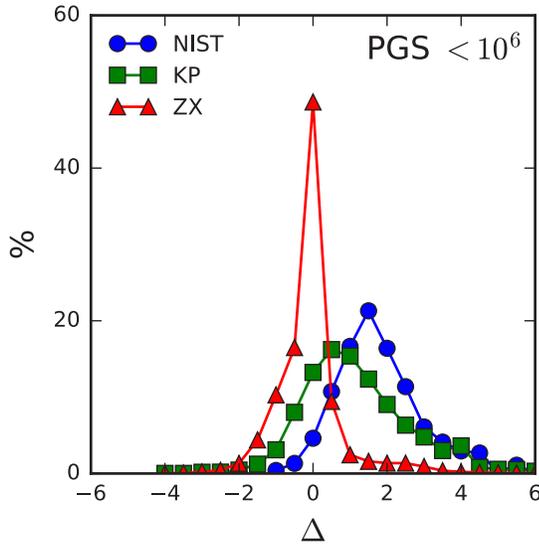


Figure 4: Δ histograms, 100k token set, online attack range ($g_{pgs} < 10^6$). zxcvbn spikes at $\Delta = 0$ then conservatively falls off to the right.

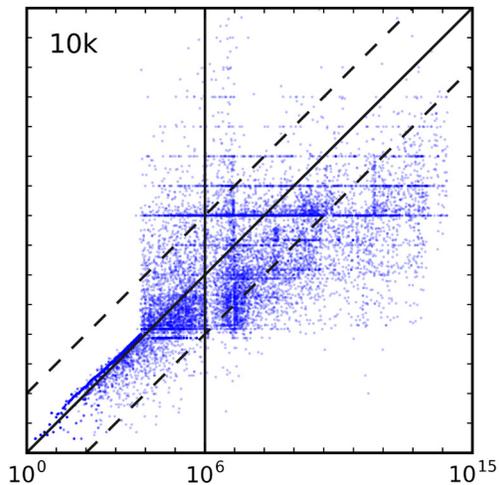


Figure 5: PGS (x) vs. zxcvbn (y), 10k token set.

would still exhibit substantial overestimation tails.

5.2.2 Choice of Data

We now contrast a single algorithm zxcvbn with varying data. Figures 3, 5, and 6 show zxcvbn with the 100k, 10k and 1k token sets. The noticeable effect is linear growth up until 10^5 , 10^4 and 10^3 , respectively. Overestimation is nearly non-existent in these respective ranges.

Within the online range, $|\Delta|$ and Δ^+ noticeably improve with more data. Past the online range, more data makes the algorithm more conservative, with progressively higher $|\Delta|$ and lower Δ^+ .

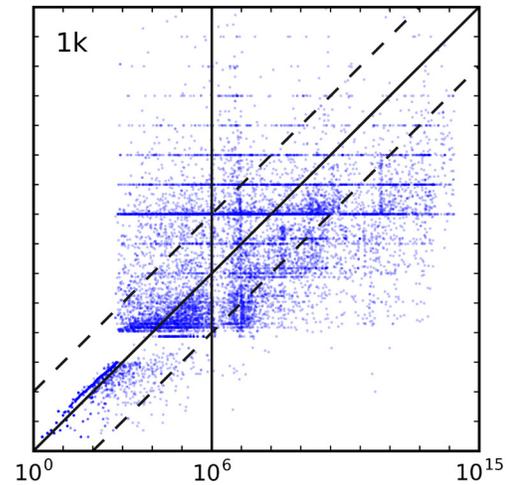


Figure 6: PGS (x) vs. zxcvbn (y), 1k token set.

5.2.3 Impact of Pattern Matching

We lastly measure the impact of matching additional patterns beyond token lookup. Figure 7 shows a variant of zxcvbn that recognizes case-insensitive token lookups only. Differences from Figure 3 include noticeably more overestimation before 10^5 and more prominent horizontal banding.

For our $|\Delta|$ and Δ^+ figures, we show the cumulative effect of starting with case-insensitive token matching only, and then incrementally matching additional types of patterns. Overall the impact is small compared to supplying additional data, but the space- and time- cost is near-zero, hence we consider these extra patterns a strict

	PGS < 10^6		PGS > 10^6	
	$ \Delta $	Δ^+	$ \Delta $	Δ^+
NIST-100k	1.81	1.79	2.04	0.14
KP-100k	1.43	1.31	1.81	0.70
ZX-100k	0.58	0.27	2.20	0.21
ZX-1k	1.47	1.23	2.13	0.46
ZX-10k	0.82	0.53	2.18	0.28
ZX-100k	0.58	0.27	2.20	0.21
ZX-100k:				
tokens only	0.68	0.48	1.85	0.30
+reversed/133t	0.68	0.47	1.87	0.29
+date/year	0.60	0.35	2.13	0.23
+keyboard	0.60	0.34	2.13	0.22
+repeat	0.57	0.28	2.19	0.21
+sequence	0.58	0.27	2.20	0.21

Table 1: $|\Delta|$ and Δ^+ summary statistics. The top, middle and bottom portions correspond to Sections 5.2.1-5.2.3, respectively.

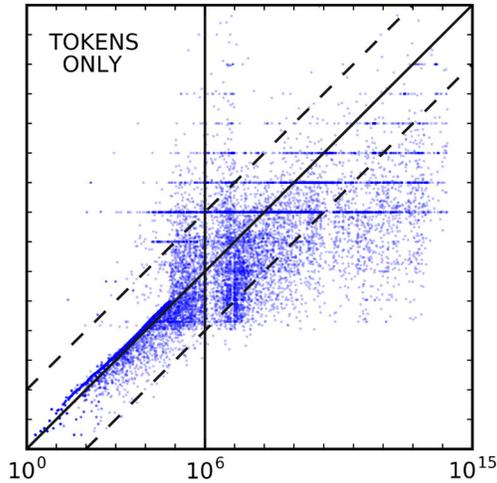


Figure 7: PGS (x) vs. zxcvbn (y), 100k token set, case-insensitive token lookups only. Table 1 shows the cumulative benefit of matching additional patterns.

improvement.

Within the online range, $|\Delta|$ shrinks by 15% after all pattern types are included. Δ^+ shrinks by 44%. As with adding more data, past 10^6 , adding patterns increases estimator conservatism, with a progressively higher $|\Delta|$ and lower Δ^+ .

5.3 Performance Benchmarks

To measure zxcvbn runtime performance, we concatenated the RockYou'09 and Yahoo'12 test sets into a single 30k list. We ran through that sample 1000 times, recording zxcvbn runtime for each password, and finally averaged the runtimes across the batches for each password. We obtained the following runtime percentiles in milliseconds:

	25 th	50 th	75 th	99.9 th	max
Chrome (ms)	0.31	0.44	0.60	3.34	27.33
node (ms)	0.38	0.53	0.72	3.00	29.61

We checked that these numbers are comparable to running a single batch, to verify that we avoided caching effects and other interpreter optimizations. Our trials used 64-bit Chrome 48 and 64-bit node v5.5.0 on OS X 10.10.4 running on a late 2013 MacBook Pro with a 2.6 GHz Intel Core i7 Haswell CPU.

5.4 Limitations

Because we measured estimator accuracy against the current best guessing techniques, accuracy will need to be reevaluated as the state of the art advances. By including training and test data from the same distribution, we

erred on the side of aggressiveness for our guessing simulation; however, test data aside, to the extent that PGS and zxcvbn are trained on the same or similar data with the same models, we expect similar accuracy at low magnitudes up until zxcvbn's frequency rank cutoff (given a harder guessing task, that range might span a lower percentage of the test set). Our experiments measured estimator accuracy but not their influence on password selection behavior; a separate user study would be valuable, with results that would likely depend on how competing estimators are used and presented.

6 Conclusion

To the extent that our estimator is trained on the same or similar password leaks and dictionaries as employed by attackers, we've demonstrated that checking the minimum rank over a series of frequency ranked lists, combined with light pattern matching, is enough to accurately and conservatively predict today's best guessing attacks within the range of an online attack. zxcvbn works entirely client-side, runs in milliseconds, and has a configurable download size: 1.5MB of compressed storage is sufficient for high accuracy up to 10^5 guesses, 245 kB up to 10^4 guesses, or 29 kB up to 10^3 guesses. zxcvbn can be bundled with clients or asynchronously downloaded on demand. It works as-is on most browsers, browser extensions, Android, iOS, and server-side javascript frameworks, with ports available in several other major languages. In place of LUDS, it is our hope that client-side estimators such as zxcvbn will increasingly be deployed to allow more flexibility for users and better security guarantees for adopters.

Acknowledgments

I'd like to thank Tom Ristenpart for shepherding this paper as well as the anonymous reviewers for their helpful comments. Thanks to Blase Ur, Sean Segreti, Henry Dixon, and the rest of the Password Research Group for their advice and help running the Password Guessability Service. Thanks to Mark Burnett for his password corpus. Thanks to Rian Hunter for extracting the KeePass estimator into a standalone Mono binary. Thanks to Devdatta Akhawe, Andrew Bortz, Hongyi Hu, Anton Mityagin, Brian Smith, and Josh Lifton for their feedback and guidance. Last but not least, a big thanks to Dropbox for sponsoring this research.

Availability

zxcvbn is free software under the MIT License:

<http://github.com/dropbox/zxcvbn>

References

- [1] Frequently Occurring Surnames from Census 1990. http://www.census.gov/topics/population/genealogy/data/1990_census/1990_census_namefiles.html.
- [2] Hashcat advanced password recovery. <http://hashcat.net>.
- [3] John the Ripper password cracker <http://www.openwall.com/john>.
- [4] KeePass Help Center: Password Quality Estimation. http://keepass.info/help/kb/pw_quality_est.html.
- [5] KeePass Password Safe. <http://keepass.info>.
- [6] Penn Treebank tokenization. <http://www.cis.upenn.edu/~treebank/tokenization.html>.
- [7] RockYou Hack: From Bad To Worse. <http://techcrunch.com/2009/12/14/rockyou-hack-security-myspace-facebook-passwords>.
- [8] The CMU Password Research Group's Password Guessability Service. <https://pgs.ece.cmu.edu/>.
- [9] The InCommon Assurance Program. <https://incommon.org/assurance>.
- [10] Wiktionary: Frequency lists. https://en.wiktionary.org/wiki/Wiktionary:Frequency_lists.
- [11] Yahoo hacked, 450,000 passwords posted online. <http://www.cnn.com/2012/07/12/tech/web/yahoo-users-hacked>.
- [12] Password management guideline. *CSC-STD-002-85*. U.S. Department of Defense, May 1985.
- [13] Password usage. *Federal Information Processing Standards Publication 112*. U.S. National Institute of Standards and Technology, April 1985.
- [14] Q3 2015 State Of The Internet. Akamai Technologies, December 2015.
- [15] BERGADANO, F., CRISPO, B., AND RUFFO, G. Proactive password checking with decision trees. In *4th ACM Conference on Computer and Communications Security* (New York, NY, USA, 1997), CCS '97, ACM, pp. 67–77.
- [16] BIRYUKOV, A., DINU, D., AND KHOVRATOVICH, D. Argon2: New generation of memory-hard functions for password hashing and other applications. In *2016 IEEE European Symposium on Security and Privacy* (March 2016), pp. 292–302.
- [17] BISHOP, M. Anatomy of a proactive password changer. In *3rd UNIX Security Symposium* (Berkeley, CA, USA, Sep. 1992), USENIX, pp. 171–184.
- [18] BLUNDO, C., D'ARCO, P., DE SANTIS, A., AND GALDI, C. Hypocrates: a new proactive password checker. *Journal of Systems and Software* 71, 1 (2004), 163–175.
- [19] BONNEAU, J. The science of guessing: Analyzing an anonymized corpus of 70 million passwords. In *2012 IEEE Symposium on Security and Privacy* (May 2012), pp. 538–552.
- [20] BONNEAU, J., HERLEY, C., OORSCHOT, P. C. V., AND STAJANO, F. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *2012 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2012), IEEE Computer Society, pp. 553–567.
- [21] BONNEAU, J., AND PREIBUSCH, S. The password thicket: Technical and market failures in human authentication on the web. In *WEIS* (2010).
- [22] BURR, W., DODSON, D., NEWTON, E., PERLNER, R., POLK, T., GUPTA, S., AND NABBUS, E. NIST Special Publication 800-63-2 Electronic Authentication Guideline. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, August 2013.
- [23] CALIFA, J. Patronizing passwords. <http://joelcalifa.com/blog/patronizing-passwords>, 2015.
- [24] CASTELLUCCIA, C., DÜRMUTH, M., AND PERITO, D. Adaptive password-strength meters from markov models. In *NDSS* (2012), The Internet Society.
- [25] DE CARNÉ DE CARNAVALET, X., AND MANNAN, M. A large-scale evaluation of high-impact password strength meters. *ACM Transactions on Information and System Security (TISSEC)* 18, 1 (2015).
- [26] DELL'AMICO, M., AND FILIPPONE, M. Monte Carlo strength evaluation: Fast and reliable password checking. In *22nd ACM Conference on Computer and Communications Security* (Denver, CO, USA, October 2015).
- [27] DELL'AMICO, M., MICHIARDI, P., AND ROUDIER, Y. Password strength: An empirical analysis. In *INFOCOM, 2010 Proceedings IEEE* (March 2010), pp. 1–9.
- [28] EGELMAN, S., SOTIRAKOPOULOS, A., MUSLUKHOV, I., BEZNOV, K., AND HERLEY, C. Does my password go up to eleven? the impact of password meters on password selection. In *SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2013), CHI '13, ACM, pp. 2379–2388.
- [29] FLORÊNCIO, D., AND HERLEY, C. A large-scale study of web password habits. In *16th international conference on the World Wide Web* (May 2007), ACM, pp. 657–666.
- [30] FLORÊNCIO, D., HERLEY, C., AND OORSCHOT, P. C. V. An administrator's guide to internet password research. In *28th USENIX Conference on Large Installation System Administration* (Berkeley, CA, USA, 2014), LISA'14, USENIX, pp. 35–52.
- [31] HERLEY, C. So long, and no thanks for the externalities: the rational rejection of security advice by users. In *New Security Paradigms Workshop* (2009), ACM, pp. 133–144.
- [32] HERLEY, C., AND OORSCHOT, P. C. V. A research agenda acknowledging the persistence of passwords. *2012 IEEE Symposium on Security and Privacy* 10, 1 (Jan 2012), 28–36.
- [33] HOLLY, R. Project Abacus is an ATAP project aimed at killing the password. *Android Central*, May 2015.
- [34] INGLESANT, P. G., AND SASSE, M. A. The true cost of unusable password policies: Password use in the wild. In *SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2010), CHI '10, ACM, pp. 383–392.
- [35] KLEIN, D. V. Foiling the cracker: A survey of, and improvements to, password security. In *2nd USENIX Security Workshop* (1990), pp. 5–14.
- [36] KOMANDURI, S., BAUER, L., CHRISTIN, N., AND OORSCHOT, P. C. V. *Modeling the adversary to evaluate password strength with limited samples*. PhD thesis, Carnegie Mellon University, 2015.
- [37] KOMANDURI, S., SHAY, R., CRANOR, L. F., HERLEY, C., AND SCHECHTER, S. Telepathwords: Preventing weak passwords by reading users' minds. In *23rd USENIX Security Symposium* (2014), pp. 591–606.
- [38] LI, Z., HAN, W., AND XU, W. A large-scale empirical analysis of chinese web passwords. In *23rd USENIX Security Symposium* (San Diego, CA, Aug. 2014), USENIX, pp. 559–574.
- [39] MA, J., YANG, W., LUO, M., AND LI, N. A study of probabilistic password models. In *2014 IEEE Symposium on Security and Privacy* (2014), IEEE, pp. 689–704.
- [40] MANBER, U., AND WU, S. An algorithm for approximate membership checking with application to password security. *Information Processing Letters* 50, 4 (1994), 191–197.
- [41] MASSEY, J. L. Guessing and entropy. In *IEEE International Symposium on Information Theory* (1994), IEEE, p. 204.

- [42] MELICHER, W., UR, B., SEGRETI, S. M., KOMANDURI, S., BAUER, L., CHRISTIN, N., AND CRANOR, L. F. Fast, lean, and accurate: Modeling password guessability using neural networks. In *Proceedings of the 25th USENIX Security Symposium* (Aug. 2016). To appear.
- [43] MORRIS, R., AND THOMPSON, K. Password security: A case history. *Communications of the ACM* 22, 11 (1979), 594–597.
- [44] MUNROE, R. xkcd: password strength. <https://xkcd.com/936/>, August 2011.
- [45] NAGLE, J. An Obvious Password Detector. <http://securitydigest.org/phage/archive/240>, November 1988.
- [46] SCHECHTER, S., HERLEY, C., AND MITZENMACHER, M. Popularity is everything: A new approach to protecting passwords from statistical-guessing attacks. In *5th USENIX Conference on Hot Topics in Security* (Berkeley, CA, USA, 2010), HotSec’10, USENIX, pp. 1–8.
- [47] SHANNON, C. E. A mathematical theory of communication. In *The Bell System Technical Journal* (1948), vol. 27, pp. 379–423, 623–656.
- [48] SHAY, R., KOMANDURI, S., KELLEY, P. G., LEON, P. G., MAZUREK, M. L., BAUER, L., CHRISTIN, N., AND CRANOR, L. F. Encountering stronger password requirements: User attitudes and behaviors. In *6th Symposium on Usable Privacy and Security* (New York, NY, USA, 2010), SOUPS ’10, ACM, pp. 2:1–2:20.
- [49] SPAFFORD, E. H. OPUS: Preventing weak password choices. *Computers & Security* 11, 3 (May 1992), 273–278.
- [50] UR, B., KELLEY, P. G., KOMANDURI, S., LEE, J., MAASS, M., MAZUREK, M. L., PASSARO, T., SHAY, R., VIDAS, T., BAUER, L., CHRISTIN, N., AND CRANOR, L. F. How does your password measure up? the effect of strength meters on password creation. In *21st USENIX Security Symposium* (Bellevue, WA, 2012), USENIX, pp. 65–80.
- [51] UR, B., SEGRETI, S. M., BAUER, L., CHRISTIN, N., CRANOR, L. F., KOMANDURI, S., KURILOVA, D., MAZUREK, M. L., MELICHER, W., AND SHAY, R. Measuring real-world accuracies and biases in modeling password guessability. In *24th USENIX Security Symposium* (2015), pp. 463–481.
- [52] WANG, D., AND WANG, P. The emperor’s new password creation policies. In *Computer Security, ESORICS 2015*. Springer, 2015, pp. 456–477.
- [53] WEIR, M., AGGARWAL, S., COLLINS, M., AND STERN, H. Testing metrics for password creation policies by attacking large sets of revealed passwords. In *17th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2010), CCS ’10, ACM, pp. 162–175.
- [54] WEIR, M., AGGARWAL, S., DE MEDEIROS, B., AND GLODEK, B. Password cracking using probabilistic context-free grammars. In *2009 IEEE Symposium on Security and Privacy* (May 2009), pp. 391–405.
- [55] WHEELER, D. zxcvbn: Realistic password strength estimation. Dropbox Tech Blog, 2012.
- [56] YAN, J. J. A note on proactive password checking. In *2001 Workshop on New Security Paradigms* (2001), ACM, pp. 127–135.

A zxcvbn vs. 3class8

We made the claim that zxcvbn is no harder to adopt than LUDS strategies such as 3class8. We provided a CommonJS implementation in

Section 3 that rejects passwords guessable in 500 attempts according to zxcvbn. For comparison, here we provide our implementation of 3class8 back-to-back with equivalent zxcvbn integrations using two other common JavaScript module interfaces: global namespacing and Asynchronous Module Definition with RequireJS.

```
var meets_3class8 = function(password) {
  var classes = 0;
  if /[a-z]/.test(password) {classes++;}
  if /[A-Z]/.test(password) {classes++;}
  if /\d/.test(password) {classes++;}
  if /[\W_]/.test(password) {classes++;}
  return classes >= 3 and password.length > 8;
}

// in .html: <script src="zxcvbn.js"></script>
var meets_policy_global = function(password) {
  return zxcvbn(password).guesses > 500;
};

requirejs(["path/to/zxcvbn"], function(zxcvbn) {
  var meets_policy_and = function(password) {
    return zxcvbn(password).guesses > 500;
  };
});
```

B Experiment implementation details

For the sake of reproducibility, we detail the specifics of the algorithms and data we employed in our experiments.

Algorithms

KeePass: We downloaded the C# source of KeePass 2.31 released on 1/9/2016 and extracted its strength estimator into a stand-alone Mono executable that takes a token dictionary as input.

NIST: calculated as specified in Section 2. The NIST 2013 guideline [22] does not precisely define the dictionary check but recommends applying common word transformations. We ignore case and check for reversed words and common l33t substitutions, the same as in zxcvbn. NIST specifies awarding up to 6 bits for passing the dictionary check, decreasing to 0 bits at or above 20 characters, but doesn’t otherwise specify how to award bits. We award a full 6 bits for passwords at or under 10 characters, 4 bits if between 11 and 15, and otherwise 2 bits. NIST recommends a dictionary of at least 50k tokens. The 100k token set described in Section 5 consists of about 390k unique tokens (consisting of several lists ending up to rank-100k).

zxcvbn: Outlined in detail in Section 4.

Data

Within each data source, all tokens were lowercased, counted, and sorted by descending count. When multiple lists contained the same token, that token was filtered from every list but the one with the lowest (most popular) rank. We made use of the following raw data:

RockYou: 32M passwords leaked in 2009 [7], excluding a random 15k test set consisting of passwords of 8 or more characters.

Yahoo: 450k passwords leaked in 2012 [11], excluding a random 15k test set consisting of passwords of 8 or more characters. We cut this list off at 10k top tokens, given the smaller size of the leak.

Xato: Mark Burnett’s 10M password corpus, released in 2015 on Xato.net and compiled by sampling thousands of password leaks

over approximately 15 years. These passwords mostly appear to be from Western users. The authors confirmed with Burnett that the RockYou set is not sampled in Xato; however, Xato likely includes a small number of samples from the Yahoo set. Given the relative sizes of the two sets, Yahoo could at most make up 4.5% of Xato; however, we expect a much smaller percentage from talking to Mark.

Wikipedia: 1-grams from the the English Wikipedia database dump of 2015-10-2. We include all Wikipedia articles but not previous revisions, edit histories, template files, or metadata. We parse text from wiki markup via the open-source WikiExtractor.py and tokenize according to the Penn Treebank method [6].

Wiktionary: Words from a 2006 Wiktionary word frequency study counting 29M words from US television and film [10]. This list balances Wikipedia’s formal English with casual language and slang. 40k unique tokens.

USCensus: Names and surnames from the 1990 United States Census [1] ranked by frequency as three separate lists: surnames, female names, and male names. We cut surnames off at 10k tokens.

C Yahoo Analysis

For reference, we reproduce the results of Section 5.2 with a sample of Yahoo’12 passwords instead of RockYou’09. Of the 15k test set, PGS cracked 29.05% within our Section 3 online guessing cutoff at 10^6 guesses and 60.07% above 10^6 , leaving 10.88% unguessed.

Choice of Algorithm

Figures 8-10 respectively show PGS vs NIST, KeePass, and zxcvbn, with each estimator supplied with the same 100k token set. As in the RockYou sample, NIST and KeePass exhibit substantial horizontal banding and overestimate at low magnitudes. At higher magnitudes, NIST tends to underestimate.

Figure 10 demonstrates that zxcvbn grows roughly linear with PGS, leaning towards underestimation, up until 10^5 guesses. Observable in the RockYou sample but more pronounced here, KeePass and zxcvbn both experience a spike in overestimation between 10^5 and 10^7 . We offer the same explanation as with RockYou: PGS is trained on a little over 10^7 unique tokens, some of which are long and unrecognized by the estimators. PGS occasionally succeeds making single-token guesses at these higher magnitudes, leading to a spike in inaccuracy between estimator dictionary cutoff and PGS dictionary cutoff.

In Figure 11, we see a similar Δ_i spike at zero for zxcvbn followed by a sharp decline to the right, indicating high accuracy and low overestimation within an online range.

Choice of Data

Figures 12-13 show PGS vs. zxcvbn with 10k and 1k token sets, respectively. We observe the same noticeable effect as with RockYou: high accuracy at low magnitudes up until the max token rank cutoff at 10^4 and 10^3 , respectively. Referring to Table 2, $|\Delta|$ and Δ^+ noticeably improve with more data within the online range. Past the online range, more data makes the algorithm more conservative, with progressively higher $|\Delta|$ and lower Δ^+ .

Impact of Pattern Matching

Figure 14 shows a variant of zxcvbn, supplied with the 100k token set, that matches case-insensitive token lookups only. We similarly observe more overestimation before $g_{pgs} = 10^5$ and more prominent horizontal banding at higher magnitudes compared to Figure 10.

The bottom portion of Table 2 shows the cumulative effect of matching additional pattern types. Within the online range, $|\Delta|$ and Δ^+ shrink by about 5% and 46%, respectively.

	PGS < 10^6		PGS > 10^6	
	$ \Delta $	Δ^+	$ \Delta $	Δ^+
NIST-100k	1.46	1.43	2.19	0.13
KP-100k	1.24	1.05	1.85	0.83
ZX-100k	0.74	0.19	2.45	0.26
<hr/>				
ZX-1k	0.74	0.19	2.45	0.26
ZX-10k	0.91	0.39	2.40	0.32
ZX-100k	1.42	1.04	2.28	0.50
<hr/>				
ZX-100k:				
tokens only	0.78	0.35	2.13	0.33
+reversed/l33t	0.79	0.33	2.19	0.32
+date/year	0.74	0.26	2.35	0.28
+keyboard	0.74	0.25	2.36	0.27
+repeat	0.73	0.19	2.43	0.26
+sequence	0.74	0.19	2.45	0.26

Table 2: $|\Delta|$ and Δ^+ summary statistics.

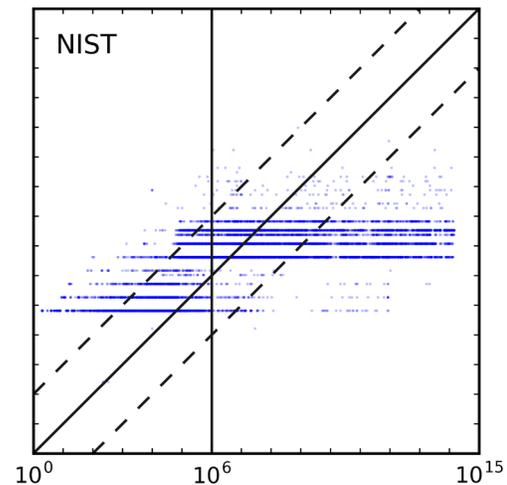


Figure 8: PGS (x) vs. NIST (y), 100k token set.

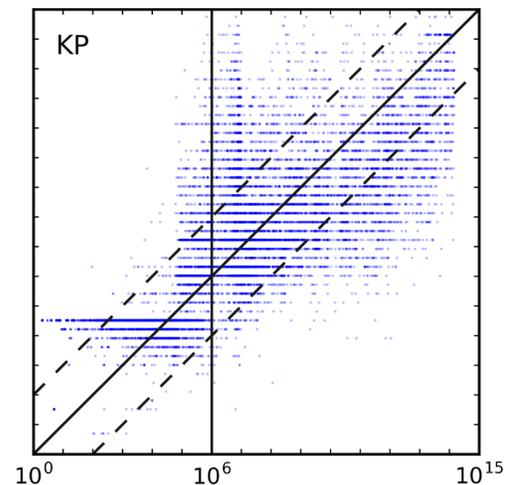


Figure 9: PGS (x) vs. KeePass (y), 100k token set.

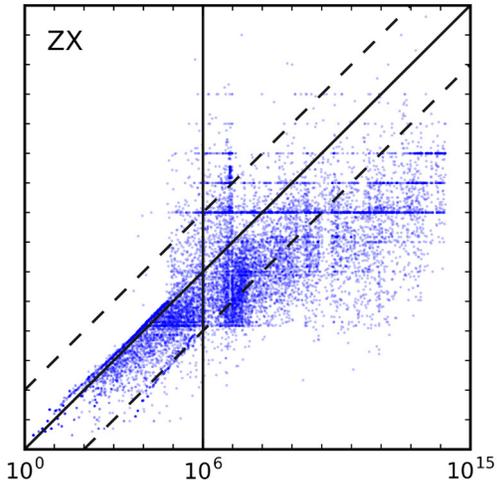


Figure 10: PGS (x) vs. zxcvbn (y), 100k token set.

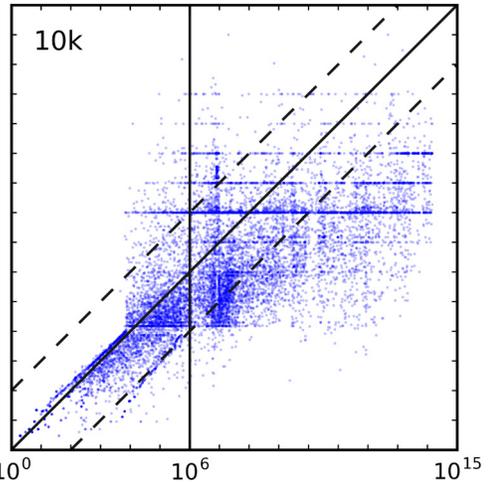


Figure 12: PGS (x) vs. zxcvbn (y), 10k token set.

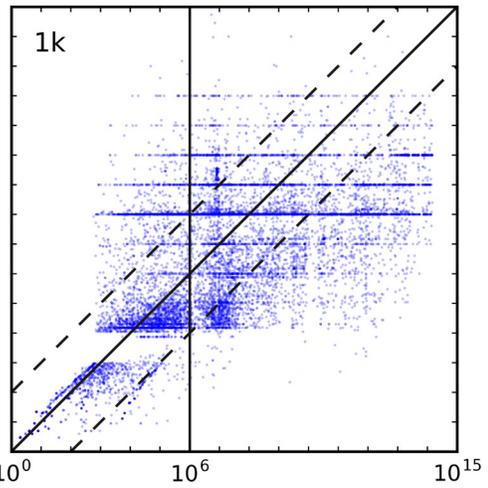


Figure 13: PGS (x) vs. zxcvbn (y), 1k token set.

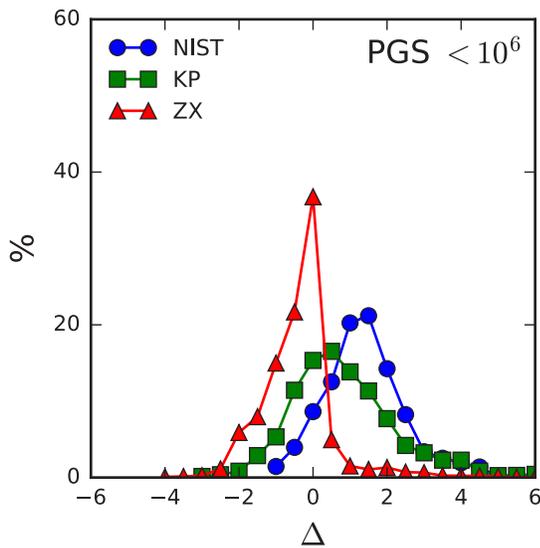


Figure 11: Δ histograms, 100k token set, online attack range ($g_{pgs} < 10^6$). zxcvbn spikes at $\Delta = 0$ then conservatively falls off to the right.

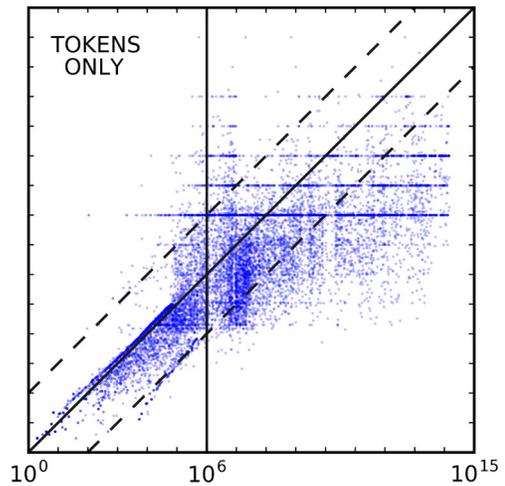


Figure 14: PGS (x) vs. zxcvbn (y), 100k token set, case-insensitive token lookups only.