pASSWORD tYPOS and How to Correct Them Securely

Rahul Chatterjee[*], Anish Athalye[†‡], Devdatta Akhawe[‡], Ari Juels[*], Thomas Ristenpart[*]
[*] Cornell Tech, [†] MIT, [‡] Dropbox

Abstract—**We provide the first treatment of typo-tolerant password authentication for arbitrary user-selected passwords. Such a system, rather than simply rejecting a login attempt with an incorrect password, tries to correct common typographical errors on behalf of the user. Limited forms of typo-tolerance have been used in some industry settings, but to date there has been no analysis of the utility and security of such schemes.**

We quantify the kinds and rates of typos made by users via studies conducted on Amazon Mechanical Turk and via instrumentation of the production login infrastructure at Dropbox. The instrumentation at Dropbox did not record user passwords or otherwise change authentication policy, but recorded only the frequency of observed typos. Our experiments reveal that almost 10% of failed login attempts fail due to a handful of simple, easily correctable typos, such as capitalization errors. We show that correcting just a few of these typos would reduce login delays for a significant fraction of users as well as enable an additional 3% of users to achieve successful login.

We introduce a framework for reasoning about typo-tolerance, and investigate the seemingly inherent tension here between security and usability of passwords. We use our framework to show that there exist typo-tolerant authentication schemes that can get corrections for "free": we prove they are as secure as schemes that always reject mistyped passwords. Building off this theory, we detail a variety of practical strategies for securely implementing typo-tolerance.

I. Introduction

Despite repeated calls for their demise (cf. [11]), human-chosen passwords remain the primary form of user authentication on the Internet. A long line of investigation has shown that passwords are easily predicted by attackers (cf. [9], [21], [42]), that strength meters offer limited improvements to security [41], that password expiration does not increase security [45], and that users have a hard time remembering complex passwords [12], [37], [38], [41].

A handful of works have pointed out that complex, user-chosen passwords are not only more difficult to remember, but also more difficult to type [26], [27], [38]. But these studies are quite limited, investigating neither the prevalence nor form of typos across a wide user base. Additional anecdotes arise in industry, where a few web services seem to intentionally allow a small set of typos [2], [3], [6], [32]. Facebook currently accepts a password whether or not the user capitalizes the first letter of their password (assuming it starts with a letter), and whether or not they have the caps lock on. But no information about why they do this has been published, and, more importantly, whether this degrades security is unclear.

We provide the first detailed treatment of password typos. We start by measuring empirically the rates and nature of typos made by users. We perform preliminary experiments with Amazon Mechanical Turk (MTurk) in which we task human workers with transcribing passwords drawn from the RockYou password leak.[1] This does not perfectly model password entry (among other reasons, because the passwords were not the workers' own), but allows us to collect over 100,000 submissions in short order across thousands of workers. Our experiment provides important, basic insights into common typographical errors. We find that a large number are proximity errors (hitting a key near the intended one). Several other common ones are what we call "easily-correctable" typos: they can all be corrected by simple functions applied to the submitted, incorrect password. Examples of the latter include accidentally hitting the caps lock, implementing incorrect first-letter capitalization, adding a character to the front or end of a password, and missing the shift key when entering a symbol at the end of a password. These easily-correctable typos account for 20% of the typos observed in our MTurk study, an observation that serves as a key basis for our work.

Armed with correction functions for easily-correctable typos, we instrument Dropbox's production, Internet-scale login infrastructure. This permits measurement of typo prevalence at scale without changing the way Dropbox currently performs user authentication and with no increased risk of exposure of passwords (i.e., we never store passwords or any information that could help in guessing them). While we cannot reveal the absolute number of requests seen during measurements for reasons of confidentiality, we note that Dropbox has hundreds of millions of customers and all user accounts were instrumented. We first perform a 24-hour measurement to identify login attempts involving the easily-correctable, common typos surfaced in the MTurk study. We find that over 9% of failed login attempts result from just one of three easily-correctable typos (caps lock, first letter case, and adding a character to the end). We perform a subsequent 24-hour experiment to analyze the impact of correcting just these top three typos. This experiment reveals that *3% of all users failed to login, but could have done so given correction of one of these three easily-correctable typos*. Many other users could have avoided multiple login attempts, significantly decreasing the time required to login. In summary, our measurements suggest that easily-correctable password typos represent a significant burden on users and businesses.

[1] We submitted our experiment design to our IRB, but received an exemption for lack of collecting any PII.

All of this suggests typo-tolerance could bring substantial usability benefits. What remains is to determine if such typo-tolerance necessarily degrades security. The intuition would be that it does, because guesses might cover multiple possible passwords. We show, however, that this intuition is flawed.

We provide a formal framework that enables principled investigation. We define *typo-tolerant password checkers*, and among these a special class that we call *relaxed checkers*. Relaxed checkers are systems that start with an existing exact system (e.g., comparing salted bcrypt hashes or using an encrypted password onion [18]). The system is "relaxed" through a modification that additionally searches a small space of corrections to the submitted password. This search allows easy deployment of typo-tolerance, while ensuring that security in the face of server compromise is as in the exact checking case (since stored values remain unchanged). Thus we focus on analyzing online guessing attacks that seek to maximize their probability of success by exploiting the extra typo checks.

We prove a *free corrections theorem*. It states that there exists an optimal, fully secure typo-tolerant checker for any desired set of corrections. Consequently: (1) The optimal remote attack up to some query budget q is no more successful than the optimal attack against an exact checker and (2) No other checking scheme can improve the utility of typo corrections while maintaining no loss in security. The key insight is that one can build a typo-tolerant checker that forgoes corrections in the rare cases when doing so will allow checking for multiple high-probability passwords.

Unfortunately, the optimal checker underlying the free corrections theorem must be based on exact knowledge of the password distribution—an assumption unlikely to be realizable in practice. We therefore explore the security of a number of practical typo-tolerant checkers, such as always correcting the top three typos, checking corrections only when they do not appear on a blacklist of common passwords, and a version of the optimal checker that uses the RockYou password leak [39] to estimate the password distribution. We then perform a number of simulations to show that these typo-tolerant checkers improve usability while remote guessing attacks improve negligibly, even in the worst case of attackers that somehow know the precise password distribution. For real attackers that estimate the distribution, our simulations suggest no improvement in online guessing attacks.

The contributions of this paper are the following:

- We are the first to investigate the rate and nature of password typos made by users via measurement studies using Mechanical Turk and the production login infrastructure at Dropbox. Our work surfaces a small set of easily-correctable typos, such as capitalization errors, that alone prevent 3% of users from logging in during the period of study. Correcting these few typos could therefore non-negligibly boost user access to the Dropbox service.
- We introduce a formal framework for typo-tolerant password checkers and prove a free corrections theorem that establishes the existence, in theory, of an optimal typo-tolerant password checker that has no loss in security over exact checking.
- We introduce a number of practical typo-tolerant password checkers that are compatible with existing password storage systems. Simulations show that these checkers can achieve no degradation in security yet still significantly improve login success rates.

Our focus is on web password ecosystems, but nothing about our techniques is unique to this setting. Our typo-tolerant checkers could easily be integrated in other settings such as logging into a desktop or laptop computer, though measurements are probably warranted to understand what are the best typo corrections for these other settings. We leave this to future work.

Immediate impact of our work. In the course of this research, our results prompted Dropbox to deploy a caps lock indicator on their website's password login interface. This indicator is like the one already available in Apple OS X, but appears in all browsers. Preliminary results suggest that it reduces caps lock errors by about 75%, and thus provides significant benefit. Unfortunately, it does not eliminate caps-lock errors nor assist with other sorts of common typos. So while our results in this paper have already had a practical impact, we hope that they will also fuel further advances in the mitigation of password typos.

II. Background and Related Work

Password checking and threats. Traditional password-based authentication systems work as follows. A user chooses a username and a password at registration time. For subsequent logins, the user submits their username and password. Using some stored representation of the password (e.g., a salted hash), a password checking scheme determines whether the submitted password matches the registered one. Login is allowed only if the equality check passes. A more formal treatment appears later in Section V.

In terms of security, two main threats arise in the context of password checking systems. The first is online guessing attacks in which the attacker can submit guesses to the checking system via the standard interface. The attacker might target a particular login (a vertical attack), or try popular passwords against multiple accounts (a horizontal attack). Here the system can employ various countermeasures to mitigate online attacks, such as slowing down how quickly responses are returned, locking accounts after a certain number of queries per unit time, and using anomaly detection mechanisms to flag requests as unauthentic based on contextual information. The second main threat is leakage to attackers of password hash databases due to compromise of authentication systems or accidental data disclosure. These attackers can mount offline brute-force attacks in an attempt to crack the passwords. Our focus will be on online brute-force attacks as, looking ahead, our checkers will be compatible with existing password storage schemes and thus no alter security with respect to offline attacks. We discuss more in Section V.

Typos in user-selected passwords. A number of prior measurement studies reveal the tendency of users to choose weak passwords [12], [21], [31] with highly skewed, heavy-headed distributions (i.e., a relatively small number of passwords are chosen by a large number of people). The most-stated reason is that ease-of-memorability guides users to simple, common passwords. While memorability is clearly a critical aspect of usability, users may also be reluctant to choose more complex passwords because entering them is difficult, error-prone, and slow. The problem may be exacerbated by various input device form factors, e.g., mobile phone touch keyboards.

Few works have measured the difficulty of correctly entering user-chosen passwords. Keith et al. [27] measured the usability of user-selected passphrases in comparison to passwords for a cohort of 56 undergraduate students. They showed that 2.2% of entries of user-chosen passwords had a typo (defined by thresholding via Levenshtein distance), and the rate of typos roughly doubles for more complex passwords (at least length 7, one upper-case, one lower-case, one non-letter). A follow-up study by the same authors also revealed a typo rate of roughly 2% with another small corpus of students [26]. Their studies, being of small scale, may not generalize to other settings, and the authors do not analyze the types of errors subjects made.

Mazurek et al. [29] hypothesize that users may pick weaker passwords because they are simpler to type and that more complex passwords are harder to type. Via large-scale measurements of a university authentication system, they show that login errors are correlated with stronger passwords. However, they do not analyze the nature of errors, i.e., whether they were in fact typos, typing in the entirely wrong password, or some other problem.

Server-side hashing changes. In theory a secure sketch [17] could be used to correct some typos in the server side. However, the proven bounds for existing constructions are too weak to provide meaningful protection for our setting (in which entropy is quite low). More details are given in Appendix A. Mehler and Skiena [30] propose to allow controlled collisions in password hashing so that, with high probability, passwords with a transposition or substitution error hash to the same value. Both such approaches to typo-tolerant techniques are not backwards-compatible with existing password storage and also will degrade offline attack security.

Typos in passphrase systems. Shay et al. [37] perform a study of system-generated passwords that are chosen uniformly for a user, chosen uniformly to be pronounceable, or chosen uniformly among CorrectHorseBatteryStaple-type passphrases [33] of various word lengths. They measure typos and investigate the correlation between password/passphrase length and typing errors, and investigate simple typo-tolerance strategies such as ignoring case completely and, for passphrases, combining words from a dictionary whose strings have large pairwise Damerau-Levenshtein distance [15], [28], which, in turn, enables correction by comparison with the dictionary. This latter suggestion is originally due to Bard [7]. Later, Jakobsson and Akavipat [24] suggest similar dictionary-checking-based error correction in what they call fastwords. In contrast to the above works, we focus on arbitrary user-chosen passwords, so these previous measurements and mechanisms unfortunately do not apply to our setting.

Typo-tolerant checking in industry. There have been several examples of major websites accepting slightly incorrect versions of user-chosen passwords. Facebook as early as 2011 accepted the correct password, the password with all letters' cases flipped, or the password with the first letter's case flipped (if it is indeed a letter) [2], [32]. These two modifications correspond to errors resulting from leaving the caps lock on (or off) or the tendency of (particularly) mobile phone keyboards to automatically capitalize the first entered character. Early password authentication mechanisms at Amazon allegedly ignored case and any characters beyond the eighth position due to a bug [1]. Users of Vanguard (an investment management company) reported that the answers to security questions could have typographical errors and still be accepted [3].

These companies faced significant backlash in the media and from some security professionals [1]–[3]. The assumption underlying the criticism seems to be that accepting any variant of a password will necessarily speed up online guessing attacks.

Open questions. To summarize, before our work there was no information available about the kinds of typos that burden users typing user-selected passwords and whether typo-tolerant password checking systems are achievable without degrading security. We seek to answer these questions here.

III. Understanding Typos Empirically

We start with experiments using Amazon Mechanical Turk (MTurk) [14] to measure the kinds of typos that people make when typing passwords. The goal of this preliminary measurement study is to discover the most frequent typos across a population for typical user-chosen passwords. We will follow on up these MTurk experiments with real user data using instrumentation of the Dropbox operational environment (discussed in Section IV).

Experiment design. MTurk allows custom-designed human-intelligence tasks (HITs) to be submitted to workers over the web. We created a password-typing HIT that asks a worker to type k passwords within a given time limit. Inside a HIT, every password needs to be typed within a conventional HTML password-type input box, i.e., each typed character shows up as a dot. Copy-paste functionality is disabled in the input boxes using the html "`onpaste=false oncopy=false`" option. This check can be circumvented by changing the browser settings, but we recorded all key presses inside an input box and used this to help filter out copy-paste attempts. We did not find any circumvention in the collected data.

Our MTurk experiment design mainly aims at gathering data efficiently to identify common typos and trends. We note that the typos found in transcribing passwords in MTurk may not be truly representative of the typos users make when typing their own passwords. Performing a longitudinal study using MTurk where users retype their chosen password multiple times would be interesting, but it would be logistically complicated and would greatly slow the data collection rates while still not providing real ecological validity [19]. The reasons for experimenting in MTurk is that prospecting for common typos in a real operational environment, such as Dropbox's, would seem to require storing information about plaintext passwords in between logins, which could represent a significant security problem. Thus we adopt the two-phase investigative approach. First prospecting for common typos via MTurk and, given a list of such typos, presenting a measurement of real-world Dropbox user typos later in the paper.

In our MTurk experiments we ask workers to type the passwords which are sourced from the RockYou password leak [39]. This data set is the largest plaintext password leak to date, with passwords from over 32 million users. It has been used widely for password-related studies and the distribution of passwords is similar to other leaks. The data set contains a number of passwords that may be objectionable to some people (e.g., many popular passwords are based on profanities from a wide number of languages). Instead of removing these passwords, which would bias the study, we used the MTurk mechanism of indicating that there may be vulgar content in the HITs. This restricts the HITs to be used only by adult workers as well as providing a warning to them about the potential for objectionable content. We also removed all passwords of length greater than 25, as these are (by manual inspection) not user-selected passwords.

Amazon allows the HIT creator to specify the required qualification and location of the worker. We allowed workers with more than 10% acceptance rate[2] and that were located in countries whose official language is English.

None of the data we recorded contains personally identifying information. We nevertheless submitted our experiment designs to our institutional review board and received an IRB exemption.

A. *Measured Typo Rates*

We sampled 100,000 passwords randomly with replacement according to the empirical probability distribution of RockYou passwords of length 6 or more. (The length requirement matches the Dropbox policy for passwords, as discussed in the next section.) By sampling with replacement, this means we match the expected distribution of submitted passwords a web service might see across their entire user base (e.g., the password "123456" appears frequently). We split the sample into HITs, ensuring that none of the HITs contain more than 180 characters in total. (This approximately normalizes the amount of typing effort of a single HIT.) Each MTurk worker is given 300 seconds to type all the passwords in the HIT. The number of passwords in a HIT ranges between 16 to 22. To ensure a broad pool of users, a worker is only allowed to submit a maximum of 3 HITs. To impose this restriction, we used a third-party JavaScript function provided by a website called Unique Turker [5]. In addition to the submitted passwords, we collected the user agent string of the browser from which the worker submitted the job, and all the key presses (and their timestamps) inside the input boxes within the HIT.

A total of 4,362 workers participated in our study. Several passwords were not typed at all (e.g., the worker accidentally submitted the HIT before typing all the passwords), and sometimes the wrong password was entered (e.g., when prompted "123456" the user entered "password").

Sanitization. We sanitize the received data first, by removing the submissions where either no password was typed or typed passwords have a case-independent edit distance of five or more from the prompted password. This excluded 226 of the password samples. Here and throughout this section edit distance includes insertions, deletions, and substitutions each as unit cost.

Preliminary analysis of the remaining data revealed that a large fraction of errors were caused by accidental pressing of the caps-lock key. Looking at the data, it was clear that in many cases workers had caps lock on for a large number of contiguous entries. We therefore sanitized our data with a heuristic to figure out improper propagation of caps-lock errors across multiple entries. The details are given in Appendix B.

After sanitization, there were in total 4,364 incorrect submissions across 97,632 valid submissions (4.5%). There were 81,595 unique passwords among the sanitized submissions, and 5.5% of all unique passwords were mistyped at least once. Because we instrumented all key presses, we could see when users corrected entries before submission. An additional 8.2% of submissions were first incorrectly typed by the workers, but corrected before submission. In total, we found that 42% of the workers made at least one typo across all their submissions, while 1.6% submitted more than four mistyped passwords in their submissions.

From now on our analyses are based on the submitted passwords unless otherwise specified. We include duplicate passwords in our analyses because they reflect the distribution of passwords a provider would see.

The data resulting from the MTurk measurements suggest that there is some correlation between typo likelihood and password complexity under various measures such as length and lexical diversity. As this is not our main focus we defer discussion to Appendix C.

B. *The Nature of Typos*

We now analyze the nature of typos made by the MTurk workers. First, we look at typos based on the edit distance between the mistyped password and the correct password.

[2] Acceptance rate in MTurk paradigm means the percentage of HITs, that the worker has submitted, have been accepted by the requester of the work. This is an eligibility filter provided by MTurk.

For 86% of incorrectly typed passwords, if we normalize the cases of alphabetic characters, the edit distance between the submission and the correct password was one. This suggests most password typos are relatively simple.

To obtain better clarity on the kinds of typos made, we analyze typographical errors on a representation of strings that accounts for the keys that must be pressed while entering it. This allows us in particular to highlight the role of capitalization errors due to shift and caps-lock mistakes during password entry.

The key-press representation of a string is defined as follows. First, recall that our standard alphabet includes upper- and lower-case letters, numbers, symbols, and the space character. We define a key-press alphabet that includes only the keys on a standard US keyboard: lower-case letters, numbers, symbols that can be entered without shift (such as the period), the caps-lock key denoted by $\langle c \rangle$, and either shift key represented by $\langle s \rangle$. Then, we convert each password and submitted string from the MTurk study to a key-press string by replacing characters omitted from the key-press alphabet by appropriate combinations of $\langle s \rangle$ or $\langle c \rangle$ tokens and tokens for keys in the key-press alphabet. We heuristically assume any sequence of 3 or more capital letters was entered using caps lock and singleton or doubles using a shift key. So for example, the string "Password" would be converted to the string "$\langle s \rangle$-p-a-s-s-w-o-r-d" over our new alphabet, and likewise "ABC12!@" would be converted to "$\langle c \rangle$-a-b-c-$\langle c \rangle$-1-2-3-$\langle s \rangle$-1-$\langle s \rangle$-2". As seen in this last example, in our conversion we insert a $\langle s \rangle$ token for each character modified by it (despite the fact that the user may hold it down for the duration).

To gain insight into what kinds of typos the workers made, we constructed a confusion matrix in which rows represent the true keys that should have been pressed, and columns represent the keys that were actually pressed. We filled this matrix in the following way. For each pair of prompted password and submitted string, we find an optimal alignment of the corresponding key presses that minimizes the total cost of the edit operations. We can extract an optimal alignment in the process of computing the minimum edit distance using a dynamic programming algorithm approach proposed in [40]. We then counted the frequency of $c \to c'$ pairs, where c is the key in the given password and c' is the key which is typed. We allowed c to take on a placeholder value [ins] signifying when c' was inserted into a password, and c' to take on a placeholder value [del] to denote that c was deleted from a password. We omitted the case $c = c'$ from our tabulation, as it represents no typographical error.

For example, consider if the prompted password was the string "Password", (or "$\langle s \rangle$-p-a-s-s-w-o-r-d" in key-press representation) and the study participant submitted the string "passw0rd1" (or "p-a-s-s-w-0-r-d-1" in key-press representation). Our algorithm would increment the counts for $\langle s \rangle \to$ [del], $o \to 0$, and [ins] $\to 1$. The corresponding typos are: forgetting $\langle s \rangle$ to capitalize the first letter, changing an 'o' to '0', and adding a (spurious) '1' at the end.

The histogram resulting from doing this for all submitted passwords is shown as a heatmap in Figure 1. Darker colors signify higher counts. The keys are sorted according to a standard US keyboard layout.

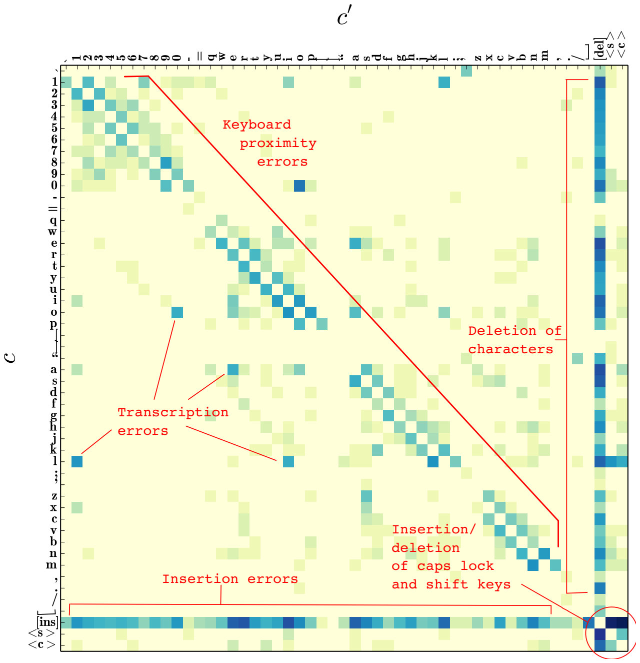

Fig. 1: Heatmap showing the counts of edits that arose in computing edit distance from the key-press sequence of the submitted passwords to the key-press sequence of the prompted passwords. The color in row c and column c' indicates how often the edit $c \to c'$ was observed across all distance calculations. The darker the color the higher the count. Labels [ins] and [del] denote insertion (character mistakenly inserted) and deletion (failure to type a character). Tokens ␣, $\langle s \rangle$, and $\langle c \rangle$ respectively denote the and space-bar, shift, and caps lock.

Several common typographical errors stand out:

- *Insertion and deletion of shift and caps-lock keys*: In the right bottom corner appears a dark patch of 3×3 squares. This reflects the frequency of erroneous use or lack of use of shift and caps lock—equivalently, incorrect insertion or deletion of the $\langle s \rangle$ and $\langle c \rangle$ tokens. These typos will switch the case of the password if it contains English letters as well as changing the shift status of digits and symbols (e.g., $4 \to \$$).
- *Keyboard proximity errors*: The slightly darker cells near the diagonal represent typos due to mistakenly pressing a neighboring key to the left or right of the intended key. We found more generally that there are a significant number of typos for which a key is replaced by an adjacent one (left, right, above, or below). We collectively refer to these as proximity errors.
- *Number-to-number errors*: We see a square cluster of moderately high-frequency errors in the top left that represent digit-to-digit typos. Some of these are proximity

errors, but many such errors confuse widely separated numbers, e.g., $3 \rightarrow 9$.

- *Insertions and deletions*: There are throughout a large number of insertion (third row from the bottom) and deletion (third column from the right) errors. Deletions are slightly more common.
- *Transcription errors*: The heatmap has sporadic dark cells, including (l,1), (o,0), (0,o), (l,i). These represent transcription errors due to a worker confusing similar-looking characters. We presume that the prevalence of reading errors are an artifact of the experiment design, and will be less frequent for entry of memorized passwords. Nevertheless, such errors could arise for users that write down their passwords to remember them.

Our analysis suggests that a large fraction of common typos fall into a few classes. A subset of these are what we refer to as "easily correctable," as we discuss shortly.

C. Touchscreen Keyboards

We performed a smaller, but similar, study in which workers were required to use touchscreen keyboards. The hypothesis here is that the distribution of typos may differ due to keyboard type. We submitted 24,000 passwords drawn from RockYou across 1,987 HITs using the same methodology of approximately normalizing effort by restricting total character counts to be less than 110. Workers were given 300 seconds to perform a HIT. We restricted workers to using touchscreen keyboards by checking the `user-agent` string of the worker's browser.

Unlike the desktop user experiment earlier in this section, we did not need to adjust for the caps lock propagation error on touchscreen devices. This was because of the fact that in touch screen devices the caps lock key is auto reset every time the focus shifts from one input field to the other. We performed an analysis that was otherwise similar to the analysis used above for the general MTurk experiment. To calculate proximity errors, we used the Android keyboard layout, which we believe is a sufficiently good proxy for all touch screen keyboards.

The probability of a typo here was 9.0%, an increase over the 4.5% for unrestricted workers. We compare the types of typos across the two data sets quantitatively below.

D. Easily-Correctable Typo Classes and Correctors

Using all the data above we manually enumerate a set of common typo types, or classes. The resulting classes are detailed in Figure 2, and shown for both the first general MTurk experiment and the touchscreen-restricted experiment. The column labeled "Corrector" identifies the function that can be used to correct the corresponding typos: swc-all switches the case of all letters in a password, swc-first switches the case of the first letter, rm-last removes the last character, rm-first removes the first character, and n2s-last changes the last character to its equivalent character under the shift-key modifier (e.g., '1' becomes '!', 'a' becomes 'A', etc.). The correctors mentioned above are mutually exclusive, that is, any two correctors, when applied to an input password of length larger than one, will produce two different passwords (assuming at least one of the correctors is applicable).

Typo type	**Corrector**	**% of typos**	
		Any	**Mobile**
Case of all letters flipped	swc-all	10.9	8.3
Case of first letter flipped	swc-first	4.5	4.7
Added extra character to end	rm-last	4.6	0.9
Added extra character to front	rm-first	1.3	0.5
Missed shift for symbol at end	n2s-last	0.2	0.1
Proximity errors	n/a	21.8	29.6
Transcription errors	n/a	3.0	3.3
Other errors	n/a	53.6	52.7

Fig. 2: The top categories of typos observed in our MTurk experiments. The "Corrector" column identifies an (easily applied) function that corrects the typo. The "Any" column is percentage of typos by category for the initial MTurk study in which workers could have used any browser. Of 97,632 passwords drawn from RockYou, 4,364 were mistyped. The "Mobile" column is the same for the 23,098 submitted passwords collected from devices with mobile browsers. Of these, 2,075 had a typo.

As can be seen, the distribution of typos is non-uniform. A few typo classes account for a large proportion of mistakes made. Caps-lock errors alone represent 9.2% of all mistakes made in our general MTurk experiments, and proximity errors for another 21.8% of all mistakes. For mobile, we see a proportionally larger number of keyboard proximity typos.

If a class of typo has a uniquely determined associated corrector, we refer to it as *easily correctable*. The typo that produces a flipped case in the first letter is an example: The corresponding corrector just flips the case of the first letter. Not all easily correctable typos have involutory correctors (the typo and corrector are the same function): consider the case of adding a character to the end of a password which is corrected by removing a character.

In contrast to easily correctable typos, a proximity error is hard to correct. Given a password with a proximity error, correction would require identification of the erroneous character as well as identification of the nearby character that was the original, true one. Thus the space of possible correctors for a proximity error is generally large. As we shall see later, both security and performance are adversely impacted by searching large spaces of correctors.

Our exploration culminates in the following two key results: *(1) Some typos are significantly more common than others* and *(2) Many common typos are easily correctable*. In the next section, we report on experiments at Dropbox that verify that common, easily correctable typos arise frequently in practice.

IV. Experiments at Dropbox

Our Mechanical Turk experiments in the last section show that there exists a small set of frequently observed typos. Those experiments, which asked users to type in passwords provided to them, may not simulate the kinds of typos users make when using their own passwords. We therefore turn to investigating typos in the production password authentication environment used at Dropbox. We will also assess the impact of typos on user experience. We emphasize that our experiments here did not change the effective login checks at Dropbox, but only recorded information about the frequency of typos.

The Dropbox authentication system. Dropbox is a file hosting service for consumers and enterprises with hundreds of millions of users. Each user must select a password during registration. Dropbox uses zxcvbn [44], a password strength estimator, to guide the user in choosing a strong password. The system requires that users choose a password of at least six characters, but it does not explicitly forbid users from choosing passwords that are considered to be weak by zxcvbn. Passwords are submitted over a standard HTTPS POST interface when logging in via the website or from within one of the native Dropbox applications. We call the submission of a password by a user a *password submission*. If the password is accepted by the Dropbox server, we call it a successful password submission, otherwise it is called a failed password submission. On a failed password submission, the user may resubmit his/her password. A *login attempt* is a sequence of password submissions by a user that either culminates in a successful login, in which case the login attempt is considered successful, or accumulates login failures until the study ends. If the user does not succeed in logging in during the scope of our study, we consider her sequence of password submissions to be a failed login attempt.

Dropbox, like most modern web companies, uses a number of fraud detection mechanisms in order to filter out spurious login attempts even before checking the password. An example of such fraud detection mechanisms is to refuse login attempts from IP addresses that appear on a blacklist for known bots. While some spurious login submissions may make it through these filtering mechanisms, we assume for simplicity below that our instrumentation is only monitoring legitimate login attempts. Note that this is a conservative assumption: if the data we collected contains illegitimate login attempts, then the true rate of correctable typos for legitimate users would be even higher. Our security analyses (Section VI) will not make such assumptions.

Instrumentation. We modified the Dropbox password checking code to perform additional checks on all legitimate login attempts on the web interface. This provided a vast amount of data, and it eliminated biases that could arise from selecting some small percentage of accounts. This also made visible multiple password submissions from a single user, which was necessary for timing re-tries.

During the period of measurement, every password submission was processed as follows. If the password check passed, do nothing. Otherwise, if it failed, apply one or more typo corrections from some predefined corrector function set $\mathcal{C} = \{f_1, f_2, \ldots, f_c\}$ where corrector functions were defined in the last section. We used slightly different sets of correctors in different experiments, as discussed below. One or more of the corrected version(s) of the password are checked. For failed login attempts, a log entry was generated that contained a time stamp, whether login would have been successful with a correction of the password, the type of correction f_i that was successful (if applicable), and the user agent string.

We emphasize that in our experiments login is *not* allowed based on the corrected passwords. We did not modify Dropbox's effective login checks; we only collected the data needed to evaluate whether doing so would be beneficial.

Typos and login failure rates. In an initial experiment we set out to measure the incidence rate of the top five corrections seen in the MTurk study of Section III. Thus for this experiment the set of corrector functions is $\mathcal{C}_{\text{top5}} = \{\textsf{swc-all}, \textsf{swc-first}, \textsf{rm-last}, \textsf{rm-first}, \textsf{n2s-last}\}$. For each instrumented failed password submission, one correction from $\mathcal{C}_{\text{top5}}$ was chosen uniformly at random and applied to the submitted password. The reason is that, in the current implementation, only sequential code is easily supported, and the password hashing scheme used at Dropbox is (by design) slow to compute. It was unclear a priori exactly what overhead the additional checks would have on Dropbox infrastructure, and so we conservatively only performed one additional check at a time. The success of this initial experiment suggested the performance impact was low, and later experiments applied multiple corrections (see below). We collected information over a 24-hour period.

We cannot report on the exact number of login attempts during this period, as this is considered confidential information by Dropbox. We will therefore report only rates of success and failure. In the following, we let c_f denote the number of times a corrector f was applied to an incorrect password during an experiment. We let r_f be the number of times f successfully corrected an incorrect password during the experiment. The ratio r_f/c_f gives the percentage of login failures correctable by f.

The left figure of Figure 3 reports the measured ratios r_f/c_f for each corrector in $\mathcal{C}_{\text{top5}}$ in during the 24-hour period. This reveals that 9.3% of failures are due to typos correctable by $\mathcal{C}_{\text{top5}}$, suggesting that typos indeed account for a significant number of failed (legitimate) password submissions[3]. By correction type, we see that the most common correction (switching the case of the first character) accounts for 60% of these, and the first three (switching the case of all characters, just the first character, dropping the last character) account for over 90% of these. Apparently capitalization errors are a significant source of errors, which provides evidence for why Facebook accepts these typos.

Some disparity with the MTurk results is apparent. While the top three of these five correctors are the same, the ordering is distinct, with caps-lock errors proportionally higher in MTurk then here. We believe this is due to the MTurk exper-

[3] We can add the fractions of typos because our correctors are mutually exclusive.

Corrected by (f)	r_f/c_f (%)
swc-all	1.13
swc-first	5.56
rm-last	2.05
rm-first	0.35
n2s-last	0.21
$\mathcal{C}_{\text{top5}}$	9.30

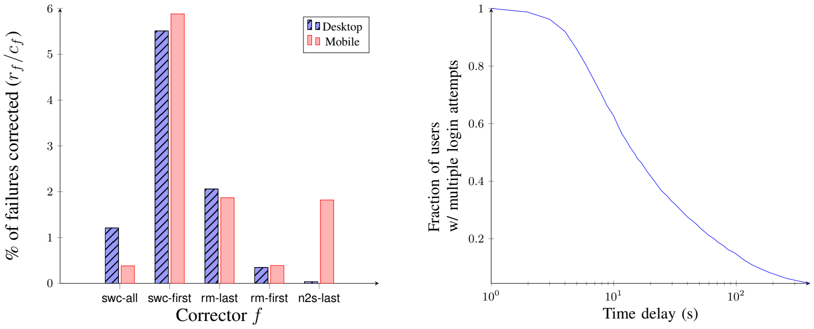

Fig. 3: **(Left)** The fraction of failed logins correctable by $\mathcal{C}_{\text{top5}}$ in a 24-hour study at Dropbox. **(Middle)** Performance of $\mathcal{C}_{\text{top5}}$ on mobile versus desktop. For each corrector in $\mathcal{C}_{\text{top5}}$ we plot the fraction of failures for each platform correctable by the corrector. **(Right)** CDF of time delay (in seconds) between the first failed login due to a typo and first successful login. Included are only users that had a failed login attempt and later a successful one.

iment design, and that the Dropbox numbers more accurately reflect rates in operational environments.

While collecting this data, we recorded the user agent for all password submissions, so we were able to analyze the performance of typo correction on mobile platforms versus desktop platforms. We found that the estimated correction rate for mobile was slightly higher at 10.5%, compared to 9.3% for desktop (calculated here with the denominator being the number of rejected password submissions for mobile and desktop, respectively). We show, in the middle figure of Figure 3, the estimated correction rates for each user agent broken down by corrector function. We see that n2s-last is a significantly more effective correction on mobile, which may be because mobile keyboards require switching to an alternate keyboard to reveal symbols. We also see that swc-all is a more effective correction on desktop, most likely because it's easier to leave caps lock enabled on conventional keyboards.[4] This dichotomy suggests the potential merit of applying different correction policies on the server based on the user agent. We leave the further analysis of this for future work.

Utility of the top three corrections. We perform a second study that restricts attention to just the overall top three correctors $\mathcal{C}_{\text{top3}} = \{\text{swc-all}, \text{swc-first}, \text{rm-last}\}$ observed in the previous study (and, in turn, the MTurk experiments). For this experiment, the instrumentation applied all three correctors to any password that failed to exactly match the registered password. So, now c_f is the number of failed login attempts for every $f \in \mathcal{C}_{\text{top3}}$. As before, we recorded data for 24 hours.

We additionally recorded the time duration for a login attempt to succeed. That is the time lag between the first failed submission and the first successful submission by each user in this 24-hour period. (Because Dropbox uses session cookies most users typically need to successfully login only once per 24-hour period.) This allowed us to quantify the time delay between failures and successes, a measure of how much utility is lost due to usability issues such as typos.

As we would expect, the success rate of corrections closely matched the results of the previous 24-hour experiment. Specifically, typos correctable by $\mathcal{C}_{\text{top3}}$ accounted for 9% of failed password submissions. This also attests the stability of these percentages over time.

We show in right figure of Figure 3 a CDF of the delay in logging in over all users who eventually succeeded at logging in (within the 24-hour period). Note that some small fraction of users did not log in for a very long time, suggesting they gave up and came back hours later. Even so, almost 20% of users that experienced a failed login would have been logged in a minute earlier should typo-tolerant checking have been enabled. Aggregated across all failed login attempts, typo-tolerance here would have *increased logged in time by several person-months just for this 24-hour experiment*. This represents a significant impact on user experience and a clear pain point for companies keen on making it easy for their users to log in.

In aggregate, of all users who attempted to log into Dropbox within the 24-hour measurement period, we discovered that 3% were turned away even though at least one of their submitted passwords was correctable by one of the correctors in $\mathcal{C}_{\text{top3}}$. This also represents a significant impact on user experience, with users being prevented from using the service.

V. Typo-tolerant Checking Schemes

In previous sections, we saw that typos account for a large fraction of login failures and that a simple set of typo corrector functions could significantly improve user experience. A natural follow-on question is whether we can achieve typo-tolerance in password authentication systems without a significant security loss. We address that question here.

We will show, by introducing what we call the "free corrections theorem," that for all natural settings there exist typo-tolerant checking schemes that correct typos with *no security loss* relative to exact checking for optimal attackers

[4] On Android devices, enabling caps lock requires pressing and holding the shift button, and on iPhone devices one has to double press the shift button to enable caps lock.

that (unrealistically) have exact knowledge of the distribution of passwords. We will also specify the optimality of the scheme underlying this theorem, i.e., showing that it achieves the maximum utility possible with no security loss.

We will define the notion of a "natural" setting formally below. Intuitively, it corresponds to the highly non-uniform, sparse (in the space of all strings) passwords chosen in practice. The schemes we analyze formally are not readily applied as is in practice because, among other things, they require exact knowledge of password and typo distributions. Nevertheless, combing our measurement studies with a theoretical perspective guides us towards the design of several concrete typo-tolerant checking schemes for which we give empirical security estimates in Section VI.

A. Password and Typo Settings

Let $\mathcal{S}$ be a set of all possible strings that could be chosen as passwords, e.g., ASCII strings up to some maximum length. We associate to $\mathcal{S}$ a distribution p that models the probability of user selection of passwords; thus $p(w)$ is the probability that some user selects a given string $w \in \mathcal{S}$ as a password. We let $\mathcal{PW} \subseteq \mathcal{S}$ be the set of possible passwords, which is formally just the support of p. We write $p(P)$ to denote the aggregate probability on a set $P \subseteq \mathcal{S}$ of strings. Following prior work (c.f., [11]), this model assumes for simplicity that the distribution of passwords is independent of the user selecting them, and that passwords are independently drawn from p.

A key feature of our formalization approach is that we do not appeal to a specific lexicographic notion of distance (e.g., Levenshtein distance) to model typos. Instead, we directly model typos as probabilistic changes to strings. Specifically, let $\tau_w(\tilde{w})$ denote the probability that upon authenticating, a user with password w types the string $\tilde{w}$. Thus τ is a family of distributions over $\mathcal{S}$, one distribution for each $w \in \mathcal{PW}$. If $\tilde{w} \neq w$ then $\tilde{w}$ is a typo; $\tau_w(w)$ is the probability that the user makes no typo. Note that $\tilde{w}$ may or may not itself be a password possibly chosen by a user, i.e., it may not be in $\mathcal{PW}$. We say that $\tilde{w}$ is a *neighbor* of w if $\tau_w(\tilde{w}) > 0$.

For all $w \in \mathcal{PW}$, then, $\tau_w(\cdot)$ defines a probability space over $\mathcal{S}$. That is, $\tau_w(\tilde{w}) \in [0,1]$ for any $\tilde{w}$ and $\sum_{\tilde{w}\in\mathcal{S}} \tau_w(\tilde{w}) = 1$. In practice, generally $\tau_w(w) > 0$, i.e., users will sometimes enter passwords correctly. Also, it will most often be the case that $\tau_w(\tilde{w}) \neq \tau_{\tilde{w}}(w)$ for $w \neq \tilde{w}$. For example, a user may mistype her password $w =$ "unlockme1" as $\tilde{w} =$"unlockme" as a result of accidentally dropping the last 1, while a user whose password is $\tilde{w} =$"unlockme" is less likely to type a 1 at the end of his password.

In our model we assume that typos depend only on a user's password w and not, for example, on the user that typed them, the time of day, or other factors. As we will see, this assumption simplifies operationalization of typo tolerance models. As one example, modeling individual users' typo habits would require a server to record the user's typo history. While higher-accuracy correction for the user might then be possible, this feature would, of course, result in a more complex system. It could also leak password information: recording the fact that a user fails to capitalize the first character in her password leaks the fact that character is a letter. From now on, a password and typo setting, or simply *setting*, is a pair (p, τ).

B. Password checkers

A password checker scheme consists of two algorithms:

- Reg is a randomized password registration algorithm. It takes as input a password w and outputs a string s that may, for example, be the output of a password hashing scheme like scrypt. These are randomized since one must choose a random salt value for each registration.
- Chk is a (possibly randomized) password verification algorithm. It takes as input a string $\tilde{w}$ and a stored string s, and outputs a Boolean value, either true or false.

In a modern, real-world service such as Dropbox, Chk is one input in a complex authentication system that combines multiple contextual, potentially probabilistic signals to make an authentication decision. A typo-tolerant checker could return a probabilistic estimate and/or combine with other contextual signals, but we focus our analysis only on deterministic checkers. Our techniques extend in natural ways to confidence values (e.g., by returning an estimate of $\tau_w(\tilde{w})$). In such a scenario, the security impact of a typo-tolerant Chk will be even lower. We also consider only *complete* checkers, meaning that for all w, $\mathsf{Chk}(w, \mathsf{Reg}(w)) \Rightarrow \mathsf{true}$.

An *exact checker* is one which never outputs true if $\tilde{w} \neq w$. In practice of course, exact checkers actually have a non-zero, but cryptographically small probability of false acceptance (for typical hash-function-based checkers, this small probability is equal to the probability of having found a collision in the hash function). We will throughout ignore this false acceptance probability. We will use ExChk to denote some secure exact checker, and assume the existence of one compatible with all password settings of interest.

Typo-tolerant checkers. We will focus our attention on building typo-tolerant checkers that *relax* the checks made by an existing exact checker construction. Let $\mathsf{Reg}, \mathsf{ExChk}$ be the algorithms of an exact checker. Then an associated relaxed checker has the same registration algorithm, but a different checking algorithm $\mathsf{Chk} \neq \mathsf{ExChk}$. Specifically, our approach will be to design relaxed checkers that enumerate some number of strings as candidates for the password and checks each with an exact checker.[5] The *ball* of a submitted string $\tilde{w}$ is the set $B(\tilde{w}) \subseteq \mathcal{S}$ of checked strings.

If balls are well constructed, the hope is that it often happens that when the user makes a typo, the true password w lies in the ball around the user submitted string $\tilde{w}$, and thus the typo can be corrected.

Relaxing an exact checker is a desirable approach to typo-tolerance for two main reasons. The first is *legacy compatibility*. Modifying a system to become typo-tolerant just requires deploying a new checking algorithm that works with

[5] We note that this can be viewed simply as the standard brute-force construction of an error correction code from an error detection code.

previously registered passwords. For example, registration may use a password hashing scheme like scrypt [36] or argon2 [8], or a password onion construction that combines password hashing with an off-system crypto service [18].

Second, relaxed checking offers *no security loss against offline, brute-force attacks* when the exact checker has, underlying it, a secure hash function. A compromise of the system or leak of the password hash database gives an attacker the registered string s, just as in the case of the exact checking system. When s is computed by applying a secure password hashing algorithm (e.g., [8], [25], [36]), an offline attacker's goal is to perform brute-force attacks to recover a password. Here one may worry that the attacker's goal is easier as it requires simply inverting s to a point that is in the ball of the target password, but for secure hash functions nothing will be revealed about the target password by s until the target password is found exactly. Thus, for a given user account, either an adversary: (1) Cracks a password hash and submits the correct password, in which case she obtains no advantage in an online attack from typo-tolerance or (2) Fails to crack a password hash, in which case she gains no benefit from her offline attack in mounting an online attack. Of course, should the typo-tolerant Chk algorithm be very complex to implement, it might increase the likelihood of software implementation vulnerabilities. For this reason, we consider simple-to-implement relaxed checkers.

Security degradation in a relaxed checker may still arise in *online* attacks. A poorly conceived relaxed checking system could diminish system security against remote brute-force guessing attacks. We will investigate this issue in detail below.

Before doing so, we note that relaxing an exact checker does circumscribe the space of possible checker designs. In particular, the size of a feasibly searchable ball $B(\tilde{w})$ is necessarily somewhat small: ExChk is designed to be computationally expensive to thwart offline brute-force guessing attacks, and relaxed checking involves running it for each string in $B(\tilde{w})$. Our measurement results in the prior sections show that even for balls of size three or four, however, significant utility improvements are possible.

Acceptance utility. We measure utility of a relaxed checker by the probability that the checker outputs true for entered passwords even when the submitted string is a typo of the true password. Formally, the acceptance utility is defined to be $\mathsf{Util}(\mathsf{Chk}) = \Pr[\text{ACC}(\mathsf{Chk}) \Rightarrow \mathsf{true}]$, where the event captures the probability that the experiment of Figure 4 outputs true. There $\leftarrow_p$ means sampling from the set according to p, and $\leftarrow_w$ means sampling from the set according to τ_w. The game is (implicitly) parameterized by the registration algorithms and the distribution pair (p, τ), and models a user's choice of password and first attempt to enter it.

The acceptance utility of an exact checker is $\mathsf{Util}(\mathsf{ExChk}) = \mathrm{E}\left[\,\tau_w(w)\,\right]$ where the expectation is over $w \leftarrow_p \mathcal{PW}$. For any non-trivial distribution τ, i.e., assuming a non-zero typo probability for some password, $\mathsf{Util}(\mathsf{ExChk}) < 1$.

```
ACC(Chk)
$w \leftarrow_p \mathcal{PW}$ ;
$\tilde{w} \leftarrow_w \mathcal{S}$
$s \leftarrow\$ \mathsf{Reg}(w)$
$b \leftarrow\$ \mathsf{Chk}(\tilde{w}, s)$
Return $b$
```

```
GUESS(Chk, $\mathcal{A}$, $q$)
$i \leftarrow 0; w \leftarrow_p \mathcal{PW}$
$\mathsf{win} \leftarrow \mathsf{false}$
$s \leftarrow\$ \mathsf{Reg}(w)$
$\mathcal{A}^{\text{Check}}$
Return $\mathsf{win}$
```

```
Check($\tilde{w}, s$)
$i \leftarrow i + 1$
$b \leftarrow \mathsf{Chk}(\tilde{w}, s)$
If $(b = \mathsf{true})$ and
   $(i \leq q)$ then
   $\mathsf{win} \leftarrow \mathsf{true}$
Ret $b$
```

Fig. 4: (**Left**) Experiment for defining acceptance utility for a checking scheme $\mathsf{Reg}, \mathsf{Chk}$. (**Right**) Security game for online guessing attacks against a checking scheme $\mathsf{Reg}, \mathsf{Chk}$ in which $\mathcal{A}$ may make q calls to its oracle Check. Both experiments are implicitly parameterized by a password and typo setting (p, τ).

C. Security definitions

As discussed above, since we focus on relaxed checkers, attacks due to compromise of an authentication server are not affected by a shift to typo-tolerance. The critical question is the effect of typo-tolerance on online guessing attacks.

Let us precisely define the notion of an online attack. In Figure 4 we give a simple guessing game played between an adversary $\mathcal{A}$ and a checker. The game GUESS is implicitly parameterized by p and the checker $\mathsf{Reg}, \mathsf{Chk}$. The success rate of the adversary $\mathcal{A}$ in guessing the password is measured as $\mathsf{Adv}(\mathsf{Chk}, \mathcal{A}, q) = \Pr[\text{GUESS}(\mathsf{Chk}, \mathcal{A}, q) \Rightarrow \mathsf{true}]$. This security game models a vertical attack, where the attacker tries to compromise a randomly chosen user account; changing this security game to model horizontal attacks is straightforward and our results extend to this setting as well.

Measuring security by this definition is quite conservative because it ignores the many countermeasures used in practice to thwart online guessing attacks. Most companies implement anomaly detection mechanisms that would, for example, block attackers that query too quickly, that use a known cracking tool or password leak to generate guesses, or that mount attacks from suspicious-looking IP addresses (those in the wrong country or on a botnet blacklist). Thus our evaluations here and in the remainder of the paper should be considered pessimistic upper bounds on true success rates.

Optimal and greedy attacks. Let $w_1, w_2, \ldots$ be a non-increasing order on passwords by probability, i.e., $p(w_1) \geq p(w_2) \geq p(w_3) \geq \ldots$. If a checker ExChk is exact, then $\mathsf{Adv}(\mathsf{ExChk}, \mathcal{A}, q) \leq \lambda_q$ for any $\mathcal{A}$ making at most q queries and where $\lambda_q = \sum_{i=1}^{q} p(w_i)$. Often λ_q is called the q-success rate. It was first defined as a measure of the unpredictability of a password distribution by Boztas [13].

Now consider a relaxed, deterministic checker. Let $B(\tilde{w})$ be the ball of a string $\tilde{w}$ for a checker Chk, which is the set of all passwords for which Chk will accept $\tilde{w}$. In submitting a guess $\tilde{w}$, an attacker induces checking on all of the strings in $B(\tilde{w})$. The adversary knows the design of the checker and so too can determine what ball will be associated with any given string submitted to the checking oracle.

Define λ_q^{fuzzy} to be the maximum guessing success probability of any adversary, namely

$$\lambda_q^{\text{fuzzy}} = \max_{\mathcal{A}} \ \mathsf{Adv}(\mathsf{Chk}, \mathcal{A}, q) \ .$$

The dependence of λ_q^{fuzzy} on Chk is left implicit in our notation but will be clear from context later. For $q = 1$, $\lambda_1^{\text{fuzzy}} = \text{argmax}_{\tilde{w} \in \mathcal{PW}}\, p(B(\tilde{w}))$. An optimal attacker simply guesses the password $\tilde{w}$ whose ball has the highest aggregate probability. This guessing strategy is analogous, in an exact-checking setting, simply to guessing the most probable password w_1. We observe that λ_1^{fuzzy} as defined here coincides conceptually with the fuzzy min-entropy notion of Fuller et al. [22], hence the fuzzy superscript in λ_q^{fuzzy}.

It turns out that implementing an optimal attack is, in general, NP-hard: finding the optimal set of queries is an instance of the weighted max cover problem. The formal reduction is shown in Appendix E. This is good news for security: it means that attackers cannot in general compute the optimal queries to make. That said, there exists a conceptually simple greedy algorithm that we now give.

Consider the following greedy adversary $\mathcal{A}^*$. At each step, it guesses the password $\tilde{w}$ whose residual ball $B(\tilde{w})$ has the highest aggregate probability. This ball is the one that maximizes $p(B(\tilde{w}) \cap P)$, where P is the set of residual passwords, those not yet checked by Chk as a result of previous adversarial queries.

More precisely, $\mathcal{A}^*$ does the following. Initialize a set $P = \mathcal{PW}$ of possible passwords. Then repeat the following q times. Guess a string $\tilde{w}$ that maximizes $p(B(\tilde{w}) \cap P)$. If the query succeeds, then the game is won; otherwise set $P \leftarrow P \setminus B(\tilde{w})$ and repeat. Let $\lambda_q^{\text{greedy}} = \mathsf{Adv}(\mathsf{Chk}, \mathcal{A}^*, q)$. As by the reduction of this problem to max cover we can claim using the classic result [23], that $\lambda_q^{\text{greedy}} \geq (1 - 1/e)\lambda_q^{\text{fuzzy}}$. Furthermore, Feige [20] has shown that this performance is indeed optimal, and no polynomial time approximation algorithm outperforms the performance of greedy.

All this gives us a way to measure security of a relaxed checker given an estimate of the password distribution p: simply compute $\lambda_q^{\text{greedy}}$ for the threshold q on online queries relevant to ones' system. This gives one, in all likelihood, the best attacker one will face in practice. One can also obtain a worst-case bound of λ_q^{fuzzy} by the formula above.

Computing even $\lambda_q^{\text{greedy}}$ in the most obvious way—a naive execution of $\mathcal{A}^*$—has time complexity on the order of $|\mathcal{S}|$ times the average ball size, and so will generally itself be intractable. We propose an alternative approach to restrict the search space and allow one to compute $\lambda_q^{\text{greedy}}$ for relevant q efficiently. The details appear in Appendix E.

Security loss. The above gives us a way to bound absolute security, but our concern will primarily be the gap between the security of today's current practice of exact checkers and the security of relaxed versions of them. This clearly depends on the password distribution and typo setting. We measure loss relative to the greedy attacker by $\Delta_q^{\text{greedy}} = \lambda_q^{\text{greedy}} - \lambda_q$ and worst-case loss by the difference $\Delta_q = \lambda_q^{\text{fuzzy}} - \lambda_q$. As $\lambda_q^{\text{fuzzy}} \leq \frac{e}{e-1} \cdot \lambda_q^{\text{greedy}}$, we can bound $\Delta_q \leq \frac{e}{e-1} \cdot \Delta_q^{\text{greedy}} + \frac{\lambda_q}{e-1} \approx 1.582\, \Delta_q^{\text{greedy}} + 0.582\lambda_q$.

By definition, $\lambda_q^{\text{fuzzy}} \geq \lambda_q$, meaning that $\Delta_q \in [0, 1)$. Moreover it holds that $\lambda_q^{\text{fuzzy}} \leq c\lambda_q$ for any tolerant checker that checks at most c strings for any input string $\tilde{w}$, i.e., $|B(\tilde{w})| \leq c$ for all $\tilde{w}$. This inequality is in fact an equality for some settings. Consider when p is uniform over $\mathcal{S}$ and that Chk is such that $|B(\tilde{w})| = c$ for all $\tilde{w}$. Then moving to the typo-tolerant checker will increase the probability of success of the optimal online brute-force attacker by a factor of c. Formally, $\lambda_q^{\text{fuzzy}} = c\lambda_q$ whenever $q \leq |\mathcal{S}|/c$.

This example seems to underlie the intuition for why typo-tolerance has been criticized as a security issue [2], [3]. Indeed, it is tempting to conclude that typo tolerance will *always* result in a factor c decrease in security. But this conclusion is too hasty: p is not uniform in reality and, in particular, passwords with high mass are sparse in the universe $\mathcal{S}$. Sparsity matters since a high λ_q^{fuzzy} depends intimately on finding strings whose balls include many passwords with high mass under p. In fact, we show next that for most natural settings one can actually obtain no security loss relative to an exact checker.

D. Free corrections theorem

We would ideally like to have typo-tolerant checkers that enjoy *free corrections*. This means that its security is equivalent to the security of an exact checker and so $\Delta_q = 0$ for any reasonable q. It is easy to come up with artificial distributions which admit free corrections of all typos. Specifically, a distribution for which no password's neighbor is in the ball of another password.

Unfortunately, just as the uniform setting, the dense setting discussed above is artificial, this completely sparse setting is also not realistic. For example, in the RockYou password leak and taking $\mathcal{C}_{\text{top5}}$ as the set of corrections to apply, one has significant overlap even among the top 50 passwords. We therefore ask: for the password distributions seen in practice, can one achieve checkers with free corrections? The answer is yes.

An optimal relaxed checker. We first give a construction of a relaxed checker that achieves free corrections for any given set of corrector functions $\mathcal{C} = \{f_0, f_1, \ldots, f_c\}$ one wants to consider, where $f_0(\tilde{w}) = \tilde{w}$ is the identity function. It achieves best-possible acceptance utility and no security loss relative to the best possible attack, assuming the checker has exact knowledge of the distribution pair (p,τ).

Fix some query budget q and recall that $p(w_q)$ is the probability mass of the q^{th} most probable password. Then let OpChk be the typo-tolerant checker that works as follows. Upon input $\tilde{w}$, generate a list of candidate typo corrections $\hat{B}(\tilde{w}) = \{w' \mid w' \leftarrow f_i(\tilde{w}),\ f_i \in \mathcal{C},\ \text{and}\ p(w') \cdot \tau_{w'}(\tilde{w}) > 0\}$. After this OpChk solves the following optimization problem to compute the set B,

$$\begin{array}{lll}
\underset{B \subseteq \hat{B}(\tilde{w})}{\text{maximize}} & \displaystyle\sum_{w' \in B} p(w') \cdot \tau_{w'}(\tilde{w}) & \text{/* Utility */} \\
\text{subject to} & p(\tilde{w}) > 0 \Rightarrow \tilde{w} \in B, & \text{/* Completeness */} \\
 & p(B) \leq p(w_q) \ \text{or} \ |B| = 1, & \text{/* Security */}
\end{array}$$

and checks all the passwords in B using ExChk. We let $B(\tilde{w})$ denote the solution of the optimization problem induced by the checker OpChk on input string $\tilde{w}$.

Observe that in addition to completeness, the constraints in the optimization problem enforce the condition that $p(B(\tilde{w})) > p(w_q)$ only if $|B(\tilde{w})| = 1$. Thus, OpChk ensures that the only balls with aggregate probability exceeding $p(w_q)$ are singletons (containing one high-probability password). The intuition here is that if we never allow a query to cover more probability mass than that of the q^{th} most popular password, then adversary $\mathcal{A}^*$ must select as its q queries the passwords $\{w_1, w_2, \ldots, w_q\}$. As these passwords define singleton balls, it follows that $\mathcal{A}^*$ will achieve exactly the same success probability as it would for an exact checker, and thus $\lambda_q^{\text{fuzzy}} = \lambda_q$.

We now give theorem statements showing that OpChk is indeed optimal in the sense that: (1) It achieves free corrections, meaning $\Delta_q = 0$ and, equivalently, $\lambda_q^{\text{fuzzy}} = \lambda_q$, for suitable q, and (2) Over all checkers with $\Delta_q = 0$, it achieves optimal utility, i.e., the highest possible probability of correcting a typo. The proofs of the following theorems appear in Appendix F.

Theorem 1 (**Free Corrections Theorem**). *Fix some password distribution p with support $\mathcal{PW}$, a typo distribution τ, $0 < q < |\mathcal{PW}|$ and an exact checker* ExChk. *Then for* OpChk *with any set of correctors $\mathcal{C}$, it holds that $\lambda_q^{\text{fuzzy}} = \lambda_q$.*

Theorem 2 (**Optimality of OpChk**). *Fix $q > 0$, a distribution pair (p, τ), and a corrector set $\mathcal{C}$. Define* OpChk *to work over $\mathcal{C}$ and let* Chk *work for a set of correctors $\mathcal{C}' \subseteq \mathcal{C}$. If* $\Delta_q(\mathsf{Chk}) = 0$, *then* $\mathsf{Util}(\mathsf{Chk}) \leq \mathsf{Util}(\mathsf{OpChk})$.

The free correction theorem applies with respect to an optimal attacker. We caution that it does not imply that for *any* attacker there is no security loss. Rather it is easy to give examples of password settings for which there exists some attack that achieves a speed-up due to tolerance. This attack, whatever it may be, cannot perform better than the optimal one. Finding good analogs to OpChk and the free correction theorem for non-optimal attackers is an interesting, open research problem. We empirically investigate in the next section the relative performance of some non-optimal attacks, showing that these achieve no meaningful speed-up due to typo tolerance.

VI. Practical Typo-Tolerant Checkers and their Security

In the previous section, we presented an optimal checker OpChk that achieves the maximum acceptance utility that is achievable with no loss in security (relative to optimal attacker). Unfortunately, OpChk is hard in general to implement, as it requires exact knowledge of the distribution pair (p, τ), which is not practically obtainable in most settings.

Here, we explore checkers that do not rely on exact distribution knowledge and are simple to implement. The first tries all corrections in some checker set. The latter two incorporate heuristics to try to avoid balls with high aggregate mass; these are directly inspired by the results regarding OpChk. As we show experimentally, our checkers can achieve high acceptance utility with minimal security degradation, and the heuristics help reduce security loss even against adversaries with exact knowledge of the probability distribution p. We also investigate the security of these checkers against more realistic adversaries that must themselves estimate the distribution p. For these adversaries our results here suggest that typo tolerance does not really help adversaries at all because of the difficulty of getting estimates right.

The tolerant checkers. For the following, let $\tilde{w}$ denote the input to the checker and $\hat{B}(\tilde{w})$ the ball of potential passwords to check as defined by the set of correctors $\mathcal{C}$ for the checker. Presented in increasing order of sophistication (and similarity to OpChk), the checkers are:

- *Check-always construction* (Chk-All): This checker checks all passwords in $\hat{B}(\tilde{w})$. Among the three checkers presented here, it achieves the greatest acceptance utility—and, conversely, the largest potential security degradation.
- *Blacklist construction* (Chk-wBL). This checker uses a blacklist L of (ostensibly high-probability) passwords. It checks $\tilde{w}$ and every other password $w \in \hat{B}(\tilde{w})$ such that $w \notin L$. Blacklisting in Chk-wBL aims to prune or eliminate non-singleton balls with high aggregate probability (as OpChk does). In our experiments, we use for the blacklist the 1,000 most popular passwords in RockYou, although one could use other blacklists as well, such as Twitter's banned password list [4].[6]
- *Approximately optimal construction* (Chk-AOp). This checker heuristically approximates OpChk. It estimates the distribution p of passwords using the empirical distribution of the RockYou password leak, and the distribution τ of typos using the empirical distribution learned from our MTurk study (see Figure 2). We denote these empirically derived distributions respectively by $\tilde{p}$ and $\tilde{\tau}$. Chk-AOp computes $B(\tilde{w})$ using the constraints used by OpChk (see the last section), but under the empirical distribution pair $(\tilde{p}, \tilde{\tau})$, rather than the (generally unknown) true distribution pair (p, τ). We set $q = 10^3$ for our experiments with Chk-AOp. We note that for the correction set sizes we consider, $c \leq 5$, solving the optimization problem is fast, as only 2^c possibilities for $B(\tilde{w})$ must be considered.

We will investigate these checkers for typo correction sets $\mathcal{C}_{\text{top2}} = \{\text{swc-all}, \text{swc-first}\}$, $\mathcal{C}_{\text{top3}} = \mathcal{C}_{\text{top2}} \cup \{\text{rm-last}\}$ and $\mathcal{C}_{\text{top5}} = \mathcal{C}_{\text{top3}} \cup \{\text{rm-first}, \text{n2s-last}\}$. In terms of utility, we know from the second Dropbox study (Section IV) the improvements obtained when using Chk-All with $\mathcal{C}_{\text{top3}}$. The other two constructions will obtain slightly less utility due to the fact that some corrections will not be checked.

[6] We emphasize that the blacklist is only used for typo corrections: we do not assume users are restricted from registering blacklisted passwords.

Our preliminary analysis, however, suggests that this utility reduction will be slight: both strategies, by design, prevent corrections only to popular passwords, which are rarely induced by typos in the first place (see Section III). For example, we can simulate acceptance utility for a given checker as defined in Section V by letting p be defined to be the RockYou empirical distribution and τ to be the empirical frequencies of typo types observed. Then for $\mathcal{C}_{\text{top3}}$ we have that the blacklist and approximately optimal strategies only reduce utility by 0.03 percentage points and 0.08 percentage points, respectively.[7]

Implementation considerations. The checkers above are all easy to implement, but care must be taken to optimize performance and ensure timing attacks do not arise. Generally, each checker should first run $\mathsf{ExChk}(\tilde{w})$ since this must always be computed. If that fails, then a constant-time check of the remainder of the ball should be performed. This involves running ExChk for the maximum number of checks that could occur for any $\tilde{w}$, i.e., $|\mathcal{C}|$. If implemented in this manner, timing and other side-channels will only potentially leak that a user made a typo, but nothing else about their password. Users that correctly input their passwords experience no performance degradation compared to existing systems.

If one instead does not use a constant time implementation, for example just running a check for each string in $B(\tilde{w})$, then timing side channels will arise that leak partial information about a user's password. For example, checking a singleton ball (which is induced by some inputs and not other inputs for $\mathsf{Chk\text{-}wBL}$ and $\mathsf{Chk\text{-}AOp}$) would be faster than checking a ball with multiple passwords. Thus the side-channel would reveal whether the user entered a high-probability password.

Security evaluation. In the remainder of this section we evaluate the security of our schemes against two types of attacker:

(A) **Exact-knowledge attackers:** We start by evaluating security of the constructions in the face of attackers that (unrealistically) know the precise distribution from which passwords are drawn. We will use a range of simulated password distributions and adversarial query budgets.

(B) **Estimating attackers:** We will then turn to more realistic attackers that do not have exact knowledge of the password distribution. Our evaluations will show that in this context an attacker attempting to take advantage of tolerant checking, even when they know the precise checker, can be quite error-prone: attackers can even do worse than naive approaches that just guess the most probable passwords in order.

Our approach for these analyses will be to utilize different password leaks to simulate true password selection. We will use the RockYou, phpBB, and Myspace leaks for these purposes. These leaks contain respectively the passwords of more than 32 million, 255,421, and 41,545 users of three different websites. Below when we say the RockYou, phpBB, or Myspace distribution we mean sampling according to the empirical distribution given by the indicated leak. Note that this means for some analyses we will use RockYou both within the designs of $\mathsf{Chk\text{-}wBL}$ and $\mathsf{Chk\text{-}AOp}$ as well as to test those designs' security, optimistically modeling that a "best-case" estimate of the distribution is known to the checker. While we could use a holdout set (sampled from RockYou without replacement, for example) to be more realistic, we instead simply perform analyses using the independent Myspace and phpBB data sets and report all of them for completeness.

A. Security against exact-knowledge attackers

We now evaluate the security of our constructions against attackers that have exact knowledge of the password distribution. Thus in this section we assume that the adversary knows not only the exact functioning of the checker being used (i.e., what typos it corrects for any submitted password), but also the precise distribution of passwords. The latter is a conservative assumption. Attackers in practice will lack such knowledge and we are therefore measuring worst-case security from this point of view.

We will focus on the greedy success rate increase $\lambda_q^{\text{greedy}} - \lambda_q$ for various values of the query budget q. We will also report on λ_q to put loss in context. To compute these values, we use the RockYou, Myspace, and phpBB distributions as a stand-ins to simulate a challenge distribution p. Since the optimal attacker is assumed to know the distributions exactly, in the exact checking setting she will simply guess the most probable q passwords. Here λ_q is straightforwardly computable (just sum the probabilities of the top q passwords in the challenge distribution). In the typo-tolerant settings, the attacker will construct a sequence of queries that achieves $\lambda_q^{\text{greedy}}$ using the algorithm given in Appendix E.

We start by comparing security for attackers given $q = 1,000$ queries across the various distributions, schemes, and corrector sets. We are here being conservative: a query budget of 1,000 is very generous to an attacker, as many websites will lock an account after tens of failed requests. Figure 5 reports the optimal success probability λ_q against an exact checker for each setting, as well as the improvements $\lambda_q^{\text{greedy}} - \lambda_q$ for each typo tolerant checker, correction set pair. All numbers are reported as percentages. The worst degradation occurs for correcting all top five errors in the Myspace setting, where the attacker's success probability increases by 3% (from 9.5% to 12.5%). To put this worst-case in perspective, consider the naive (and incorrect) assumption that seems to underlie the criticism of typo tolerance [2]: it suggests instead a fivefold increase in attacker success when correcting five errors and thus an increase to 47.5% in the Myspace setting.

Elsewhere the increase is much smaller. For example, with Rockyou, one can always correct all top five errors with increase only 1.6%: an attacker's probability of success goes from 11.2% to 12.8%, a small improvement. This means

[7] The absolute acceptance utilities for $\mathcal{C}_{\text{top3}}$ in these simulations are 0.9628, 0.9625, and 0.9620. But the low overall rate of typos in the MTurk experiments means that exact checking here obtains 0.9564 acceptance utility already, which is significantly less than what is implied by our Dropbox measurements.

Challenge		$q = 10$				$q = 100$				$q = 1000$			
Dist.	**Set**	All	wBL	AOp	Ex	All	wBL	AOp	Ex	All	wBL	AOp	Ex
RockYou	$\mathcal{C}_{\text{top2}}$	0.03	0.00	0.00	1.95	0.15	0.05	0.00	4.50	0.51	0.32	0.00	11.23
	$\mathcal{C}_{\text{top3}}$	0.22	0.03	0.00		0.56	0.14	0.00		1.41	0.86	0.00	
	$\mathcal{C}_{\text{top5}}$	0.25	0.06	0.00		0.63	0.18	0.00		1.57	0.87	0.00	
phpBB	$\mathcal{C}_{\text{top2}}$	0.03	0.00	0.00	2.75	0.12	0.02	0.00	5.50	0.38	0.19	0.15	12.71
	$\mathcal{C}_{\text{top3}}$	0.19	0.02	0.00		0.28	0.04	0.01		1.01	0.60	0.42	
	$\mathcal{C}_{\text{top5}}$	0.20	0.03	0.01		0.31	0.05	0.02		1.13	0.72	0.47	
Myspace	$\mathcal{C}_{\text{top2}}$	0.03	0.01	0.00	0.79	0.15	0.12	0.03	2.86	0.49	0.45	0.35	9.54
	$\mathcal{C}_{\text{top3}}$	0.17	0.06	0.02		0.62	0.46	0.32		2.46	2.21	1.59	
	$\mathcal{C}_{\text{top5}}$	0.27	0.15	0.04		0.87	0.68	0.52		3.00	2.66	1.94	

Fig. 5: Percentage improvements in an exact-knowledge adversary's success ($\lambda_q^{\text{greedy}} - \lambda_q$) for each setting (corrector strategy and correction set) and each of the challenge distributions, for $q \in \{10, 100, 1000\}$.

that the adversary's first 1,000 guesses against a typo-tolerant checker do not benefit much from high-probability balls.

Moving from $\mathcal{C}_{\text{top2}}$ to $\mathcal{C}_{\text{top3}}$ can result in a relatively big jump in security loss. The reason is that the rm-last typo corrector admits many higher-mass balls than only correcting the considered capitalization errors. For example, adding a character to many popular passwords results in another popular password: `password` and `password1`, `abc123` and `abc1234`. Fewer such pairs exist for capitalization errors since fewer users choose passwords with capital letters. Indeed in the worst case for $\mathcal{C}_{\text{top2}}$ we see a just a 0.5% improvement in adversarial success compared to the 2.42% worst-case jump for $\mathcal{C}_{\text{top3}}$. It is no coincidence, perhaps, that Facebook's policy seems to align with Chk-All for $\mathcal{C}_{\text{top2}}$. Our measurements are the first reported validation of this policy.

Even though security loss is low for Chk-All, one may want to do better. The blacklist and approximately optimal checkers help. When the challenge distribution is RockYou the approximately optimal checker Chk-AOp is, in this case, actually optimally secure by construction, hence it suffers no security loss at all. Also note that Chk-wBL may benefit unduly by knowing exactly the top 1,000 passwords from RockYou. Thus the more important analyses are when tested on independent distributions. Here we see some loss as one would expect given that the attacker in these cases has, after all, more information about the challenge distribution than the checker. But now the loss is small, and Chk-AOp reduces the security loss compared to Chk-All by 0.53% on average over the Myspace and phpBB settings. Chk-wBL also reduces loss compared to Chk-All by 0.27% on average over Myspace and phpBB, but never improves security more than Chk-AOp.

We now turn to what happens as q varies. In Figure 5 shows the attack success increases for the $q = 10$ and $q = 100$ cases. We note that the most realistic in practice is $q = 10$, since companies often will raise alarms after 10 consecutive failed login attempts. Here we see that attackers benefit little from typo-tolerance, and our Chk-AOp reduces loss to 0.04% or less. Often it is zero.

It is concievable that in some settings an attacker might be able to make more than $q = 1,000$ queries, which implies that our checker assumed too low of a bound on q. We focus for simplicity on $\mathcal{C}_{\text{top2}}$, a choice we expect many deployments to utilize, and show in Figure 6 the security loss using RockYou as the distribution for a range of $q \in \{1, 100, 200, 300, \ldots, 10{,}000\}$. We have drawn a vertical dotted line at $q = 1,000$, which was used by Chk-AOp as the expected query budget. As before, Chk-AOp has no loss below $q = 1,000$, and only after the attacker gets more than 1,000 queries does the attacker obtain an improvement over the exact checking case. Figure 7 shows the same type of chart but now for phpBB. This distribution leads to security loss seeing big discrete jumps for larger q, suggesting that at certain points the attacker can take advantage of new balls that just come in to play as higher mass than individual passwords. A chart for Myspace would exhibit similar trends as the one for phpBB, we omit it for the sake of brevity.

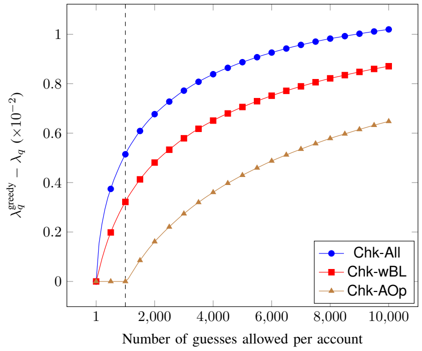

Fig. 6: The security loss as a function of q for challenge distribution RockYou and $\mathcal{C}_{\text{top2}}$.

Observe that as q gets larger, the improvement $\lambda_q^{\text{greedy}} - \lambda_q$ flattens out in both charts. The attacker in the typo-tolerant cases runs out of heavily-weighted balls to take advantage of and ends up just querying passwords that cover only one (high-probability) guess. For RockYou, we see that the improvement is never more than 1% and, for phpBB, never more than 0.8%. This all suggests that even as q grows to values unlikely ever to arise in practice, typo-tolerance nevertheless does not improve the attacker's rate of success by much. We note that our blacklisting and approximately optimal checkers can be made even more conservative should one desire, by blacklisting

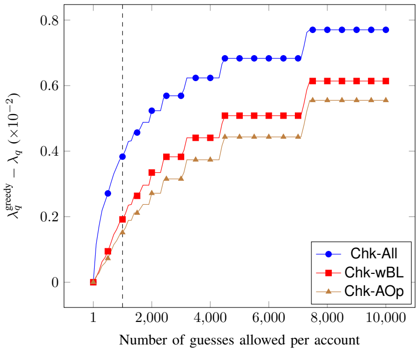

Fig. 7: Difference in exact-knowledge adversary success against typo-tolerant schemes and exact checking, as a function of q for challenge distribution phpBB and $\mathcal{C}_{\text{top2}}$.

more passwords or setting q larger, respectively.

As noted in the previous section, the greedy algorithm is known to provide a good approximation for the weighted max coverage problem. Given that the password probabilities range over a very dense space, and our correction sets are quite small, we expect $\lambda_q^{\text{greedy}} \approx \lambda^{\text{fuzzy}}$. We do not have any theoretical proof for this claim, and leave analysis as an important question for future work. We can of course always bound the actual value of Δ_q via $\Delta_q \leq 1.582\,\Delta_q^{\text{greedy}} + 0.582\lambda_q$. So, for example with the RockYou challenge distribution, $q = 10$, and the $\mathcal{C}_{\text{top5}}$ corrector set we have that $\Delta_q \leq 0.0153$ as compared to $\lambda_q^{\text{greedy}} - \lambda_q = 0.0063$. We expect this three-fold decrease in relative security to be quite pessimistic: the better the greedy algorithm approximates the problem the worse the adjustment to compute Δ_q becomes.

B. *Estimating attackers*

We have so far considered attackers that have exact knowledge of the password distribution (even when the system designer may not). In practice such attackers do not exist, and instead adversaries must try to estimate the distribution of passwords. We refer to these as estimating attackers. As before, we assume adversaries know the exact checking algorithm in use.

We started by considering an adversary that estimates the password distribution using the Weir et al. probabilistic context-free grammar (PCFG) [43], a trained model of password distributions used to build effective crackers. However, our experiments with this showed that it provides poor efficacy in online guessing attacks, doing significantly worse than the approaches we describe below and, importantly, it did equally poorly against the typo-tolerant checkers in all settings.

We therefore turn to a different adversarial strategy for estimating the password distribution. We measure the success rate of an attacker that uses one of the password leaks as its estimate of the distribution. This is a typical strategy in practice. We test these attacks against the other two distributions and for each of the exact checking, Chk-All, Chk-wBL, and Chk-AOp. The latter three use $\mathcal{C}_{\text{top2}}$. The security loss for all combinations are tabulated in Figure 8. (Note that the left-to-right diagonals reflect some of the results already shown for the exact-knowledge attacker in Figure 5.)

	Attacker distribution	**Challenge distribution**		
		RockYou	phpBB	Myspace
ExChk	RockYou	11.23	3.21	9.34
	phpBB	8.10	12.71	1.81
	Myspace	3.57	3.32	9.54
Chk-All	RockYou	+0.51	+0.28	+0.25
	phpBB	+0.25	+0.38	+0.11
	Myspace	**-0.15**	**-0.02**	+0.49
Chk-wBL	RockYou	+0.32	+0.11	+0.20
	phpBB	+0.06	+0.19	+0.05
	Myspace	**-0.26**	**-0.20**	+0.46
Chk-AOp	RockYou	0.00	0.00	0.00
	phpBB	**-0.11**	+0.15	**-0.04**
	Myspace	**-0.27**	**-0.14**	+0.35

Fig. 8: The top table shows the success rate of an attack against the exact checking scheme for the attacker-estimated distribution (row) used against the challenge distribution (column). The remaining tables show the *difference* between success rate of an attacker against the tolerant scheme and the exact checking scheme, for the indicated attacker-estimated and actual challenge distribution pairs. All values are in percentages.

The improvement the attacker obtains when one switches to a tolerant checking system is never greater than 0.28%. More interestingly, in some cases the difference is negative, which means that the attacker did *worse* against the typo-tolerant scheme. This may be counterintuitive, but here the estimates the attacker makes about the distribution can often be wrong. This can lead her to choose a set of guesses that maximizes the total success probability according to her estimate but not according to the challenge distribution. We give an example for the curious reader in Appendix G.

In summary, our simulations here suggest that a carefully designed typo-tolerant checker will result in little to no security loss against realistic adversaries.

VII. Conclusion

We presented the first treatment of typo-tolerant password authentication. We demonstrated, with large-scale, real-world experiments, that password typos are a real and common source of user errors in authentication systems. We found that a few types of typo-corrections account for an overwhelming number of password typos. We provided a formal framework for exploring typo-tolerant password checkers, and focused on a class of them called relaxed checkers that are backwards-compatible with existing password hashing schemes. We showed, via what we call the free corrections theorem, that there exist relaxed checkers against which the best attack performs no better than the best attack against an exact checker. Unfortunately the construction requires exact knowledge of the password distribution. We therefore gave a number of practical typo-tolerant checkers inspired by it, and analyzed their security empirically, showing that one can easily obtain significant utility improvement with minimal or no security degradation.

In future work, we plan to investigate whether typo-tolerance will actually serve to improve overall security. Because allowing for password typos increases login success rates in benign scenarios, it may help to make adversarial login attempts stick out. This would strengthen the signals used to detect online password attacks as used in Internet-scale authentication systems.

Acknowledgements

This work was funded in part by NSF grants CNS-1330308 and CNS-1514163 as well as a generous gift by Microsoft.

References

[1] "Amazon login may accept password variants," http://www.ghacks.net/2011/01/31/amazon-login-may-accept-password-variants/, accessed: 2015-11-06.

[2] "Facebook passwords are not case sensitive," http://www.zdnet.com/article/facebook-passwords-are-not-case-sensitive-update/, accessed: 2015-11-12.

[3] "Is Vanguard making it too easy for cybercriminals to access your account?" http://www.thestreet.com/story/13213265/4/is-vanguard-making-it-too-easy-for-cybercriminals-to-access-your-account.html?startIndex=0, accessed: 2015-11-06.

[4] "Twitter's list of 370 banned passwords," http://www.businessinsider.com/twitters-list-of-370-banned-passwords-2009-12, accessed: 2015-11-06.

[5] "Uniq Turker," https://uniqueturker.myleott.com/, accessed: 2015-15-10.

[6] S. Antilla, "Vanguard group fires whistleblower who told TheStreet about flaws in customer security." *TheStreet*, 18 Sept. 2015.

[7] G. V. Bard, "Spelling-error tolerant, order-independent pass-phrases via the Damerau-Levenshtein string-edit distance metric," in *Proceedings of the fifth Australasian Symposium on ACSW Frontiers-Volume 68*. Australian Computer Society, Inc., 2007, pp. 117–124.

[8] A. Biryukov, D. Dinu, and D. Khovratovich, "Argon and argon2: password hashing scheme," Technical report, Tech. Rep., 2015.

[9] J. Bonneau, "Guessing human-chosen secrets," Ph.D. dissertation, University of Cambridge, May 2012. [Online]. Available: http://www.cl.cam.ac.uk/~jcb82/doc/2012-jbonneau-phd_thesis.pdf

[10] J. Bonneau, "The science of guessing: analyzing an anonymized corpus of 70 million passwords," in *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2012, pp. 538–552.

[11] J. Bonneau, C. Herley, P. C. van Oorschot, and F. Stajano, "The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes," in *2012 IEEE Symposium on Security and Privacy*, May 2012.

[12] J. Bonneau and S. Schechter, "Towards reliable storage of 56-bit secrets in human memory," in *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX, 2014.

[13] S. Boztas, "Entropies, guessing, and cryptography," *Department of Mathematics, Royal Melbourne Institute of Technology, Tech. Rep*, vol. 6, pp. 2–3, 1999.

[14] M. Buhrmester, T. Kwang, and S. D. Gosling, "Amazon's mechanical turk a new source of inexpensive, yet high-quality, data?" *Perspectives on psychological science*, vol. 6, no. 1, pp. 3–5, 2011.

[15] F. J. Damerau, "A technique for computer detection and correction of spelling errors," *Communications of the ACM*, vol. 7, no. 3, pp. 171–176, 1964.

[16] Y. Dodis, "On extractors, error-correction and hiding all partial information," in *Theory and Practice in Information-Theoretic Security, 2005. IEEE Information Theory Workshop on*, 2005, pp. 74–79.

[17] Y. Dodis, L. Reyzin, and A. Smith, "Fuzzy extractors: How to generate strong keys from biometrics and other noisy data," in *Eurocrypt 2004*, C. Cachin and J. Camenisch, Eds. Springer-Verlag, 2004, pp. 523–540, lNCS no. 3027.

[18] A. Everspaugh, R. Chatterjee, S. Scott, A. Juels, T. Ristenpart, and C. Tech, "The Pythia PRF service," in *Proceedings of the 24th USENIX Conference on Security Symposium*. USENIX Association, 2015, pp. 547–562.

[19] S. Fahl, M. Harbach, Y. Acar, and M. Smith, "On the ecological validity of a password study," in *Proceedings of the Ninth Symposium on Usable Privacy and Security*. ACM, 2013, p. 13.

[20] U. Feige, "A threshold of ln n for approximating set cover," *Journal of the ACM (JACM)*, vol. 45, no. 4, pp. 634–652, 1998.

[21] D. Florencio and C. Herley, "A large-scale study of web password habits," in *Proceedings of the 16th International Conference on World Wide Web*, ser. WWW '07. New York, NY, USA: ACM, 2007, pp. 657–666. [Online]. Available: http://doi.acm.org/10.1145/1242572.1242661

[22] B. Fuller, A. Smith, and L. Reyzin, "When are fuzzy extractors possible?" Cryptology ePrint Archive, Report 2014/961, 2014, http://eprint.iacr.org/.

[23] D. S. Hochbaum, "Approximating covering and packing problems: set cover, vertex cover, independent set, and related problems," in *Approximation algorithms for NP-hard problems*. PWS Publishing Co., 1996, pp. 94–143.

[24] M. Jakobsson and R. Akavipat, "Rethinking passwords to adapt to constrained keyboards," *Proc. IEEE MoST*, 2012.

[25] B. Kaliski, "PKCS #5: Password-based cryptography specification version 2.0," 2000, RFC 2289.

[26] M. Keith, B. Shao, and P. Steinbart, "A behavioral analysis of passphrase design and effectiveness," *Journal of the Association for Information Systems*, vol. 10, no. 2, p. 2, 2009.

[27] M. Keith, B. Shao, and P. J. Steinbart, "The usability of passphrases for authentication: An empirical field study," *International journal of human-computer studies*, vol. 65, no. 1, pp. 17–28, 2007.

[28] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8, 1966, pp. 707–710.

[29] M. L. Mazurek, S. Komanduri, T. Vidas, L. Bauer, N. Christin, L. F. Cranor, P. G. Kelley, R. Shay, and B. Ur, "Measuring password guessability for an entire university," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 173–186.

[30] A. Mehler and S. Skiena, "Improving usability through password-corrective hashing," in *String Processing and Information Retrieval*. Springer, 2006, pp. 193–204.

[31] R. Morris and K. Thompson, "Password security: a case history," *Commun. ACM*, vol. 22, no. 11, pp. 594–597, Nov. 1979. [Online]. Available: http://doi.acm.org/10.1145/359168.359172

[32] A. Muffet, "Facebook: Password hashing & authentication," Presentation at Real World Crypto, 2015.

[33] R. Munroe, "Password strength," https://xkcd.com/936/, accessed: 2015-11-13.

[34] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher, "An analysis of approximations for maximizing submodular set functionsi," *Mathematical Programming*, vol. 14, no. 1, pp. 265–294, 1978.

[35] R. Ostrovsky and Y. Rabani, "Low distortion embeddings for edit distance," *Journal of the ACM (JACM)*, vol. 54, no. 5, p. 23, 2007.

[36] C. Percival and S. Josefsson, "The scrypt password-based key derivation function," 2015.

[37] R. Shay, P. G. Kelley, S. Komanduri, M. L. Mazurek, B. Ur, T. Vidas, L. Bauer, N. Christin, and L. F. Cranor, "Correct horse battery staple: Exploring the usability of system-assigned passphrases," in *Proceedings of the Eighth Symposium on Usable Privacy and Security*. ACM, 2012, p. 7.

[38] R. Shay, S. Komanduri, A. L. Durity, P. S. Huh, M. L. Mazurek, S. M. Segreti, B. Ur, L. Bauer, N. Christin, and L. F. Cranor, "Can long passwords be secure and usable?" in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2014, pp. 2927–2936.

[39] M. Siegler, "One of the 32 million with a RockYou account? you may want to change all your passwords. like now." *TechCrunch*, 14 Dec. 2009.

[40] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.

[41] B. Ur, P. G. Kelley, S. Komanduri, J. Lee, M. Maass, M. L. Mazurek, T. Passaro, R. Shay, T. Vidas, L. Bauer *et al.*, "How does your password measure up? the effect of strength meters on password creation." in *USENIX Security Symposium*, 2012, pp. 65–80.

[42] B. Ur, F. Noma, J. Bees, S. M. Segreti, R. Shay, L. Bauer, N. Christin, and L. F. Cranor, ""I added '!' at the end to make it secure": Observing

password creation in the lab," in *Eleventh Symposium On Usable Privacy and Security (SOUPS 2015)*, 2015, pp. 123–140.

[43] M. Weir, S. Aggarwal, B. de Medeiros, and B. Glodek, "Password cracking using probabilistic context-free grammars," in *IEEE Symposium on Security and Privacy (SP)*, 2009, pp. 162–175.

[44] D. Wheeler, "zxcvbn: Realistic password strength estimation," Dropbox Tech Blog, Apr. 2012, https://blogs.dropbox.com/tech/2012/04/zxcvbn-realistic-password-strength-estimation/.

[45] Y. Zhang, F. Monrose, and M. K. Reiter, "The security of modern password expiration: an algorithmic framework and empirical analysis," in *ACM Conference on Computer and Communications Security (ACM CCS)*, 2010, pp. 176–186. [Online]. Available: http://doi.acm.org/10.1145/1866307.1866328

APPENDIX

A. Secure Sketches

Secure sketches and fuzzy extractors, explored by Dodis et al. [16], [17], are designed to generate consistent, cryptographically strong keys from noisy secrets, such as biometric data. They may also be applied to passwords, as typographical errors in passwords can be modeled as noise. Dodis et al. proposed two ways to construct secure sketches for the edit-distance metric space; see Section 7 of [17]. They show how to use a low-distortion embedding for the edit-distance metric given by Ostrovsky and Rabani [35], and also describe a relaxed embedding for the edit-distance metric using c-shingles. The security losses for these constructions, as given in Proposition 7.2 and Theorem 7.5 of [17], are $t(\log F)2^{O\left(\sqrt{\log(n\log F)\log\log(n\log F)}\right)}$ and $\lceil \frac{n}{c} \rceil \log(n-c+1)-(2c-1)t\lceil \log(F^c+1)\rceil$ respectively. Here, n is the size of the password, F is the alphabet size, t is the number of errors/edits tolerated, and c is a construction parameter denoting the size of the shingles. In our setting, typical values would be $n = 8$, $t = 1$, and $F = 96$. The value of c, according to Theorem 7.4, should be 1 in our setting (and the loss is an increasing function in c). Given these parameters, the entropy loss of the two secure sketches would be ≈ 91 bits and ≈ 31 bits respectively. The min-entropy of real world password distributions is only about ≤ 8 bits [10]. Thus known constructions provide no security guarantees in our context, and providing proven constructions that do would seem to require new techniques.

B. Sanitizing Caps-Lock Errors

As mentioned in the paper body, preliminary analysis of the data revealed that a large fraction of errors was caused by accidental pressing of the caps-lock key. Measuring the rate of caps-lock errors is more challenging than for other typos, for two reasons. First, caps-lock key presses are not recordable via keystroke logging, and thus not directly detectable remotely. Second, if the user engages the caps-lock key while typing one password, there's a chance that it will remain on (erroneously) while the next one is entered. In MTurk, if an individual worker is to be permitted to enter more than one password—even across multiple HITs—propagation of caps-lock typos across passwords is therefore methodologically unavoidable. This second issue accounts for the (artificially) high rate of caps-lock typos observed in our experiments. We found that 76 HITs contributed to 1120 caps-lock errors.

To adjust for the effect of such propagation errors in determining the rate of caps-lock typos, we do the following. We define a caps-lock error as an incorrect password which, when the cases of all the letters are inverted, becomes correct. In sequentially processing the passwords in a HIT, we use a variable $\mathsf{CL\text{-}ERR} \in \{\texttt{0}, \texttt{1}\}$ to denote a heuristic determination as to whether the caps lock is in an error state when the user entered a password in a HIT. (An error state could either be that caps lock is on and the user should have typed lower-case letters, or caps lock is off and the user should have typed upper case letters.) We initially let $\mathsf{CL\text{-}ERR} = \texttt{0}$. When we detect a caps-lock error in a password, we record it and set $\mathsf{CL\text{-}ERR} = \texttt{1}$. If it is already the case that $\mathsf{CL\text{-}ERR} = \texttt{1}$ when we reach a password in a HIT, we discard the password. That is, in such cases, we do not count it in our computation of error rates for any typo. Additionally, for every password in a HIT, we determine (heuristically) whether the caps lock has been turned off during entry of the password. If the password contains at least one letter and the password was submitted correctly, then we set $\mathsf{CL\text{-}ERR} = \texttt{0}$.

In general, the intuition here is that we keep track heuristically of whether the caps-lock key appears to be engaged erroneously. If the entry of a password in a HIT has been affected by the state of the erroneously engaged caps-lock key, we treat it as "tainted," and thus discard it from our experiment. (We assume heuristically that caps-lock errors are independent of other typos. The global effect of discarding "tainted" passwords and not recording typos they contain in addition to caps-lock errors is small in any case.)

C. Complexity and Typo Likelihood

Our MTurk experiments revealed a significant initial finding regarding the frequency of typos. Typo rates in our study increased under the following three distinct metrics relating to password complexity.

Lexical diversity in passwords: One might suspect that more lexically diverse passwords—ones that include symbols, letters with different cases, numbers or some combination thereof—would be more prone to typos. We define four character classes: upper case letters, lower case letters, digits, and symbols. Now, based on how many of the four classes of characters are present within a password we can partition passwords into four buckets. For example, passwords containing characters from only one of the four classes are binned as bucket 1, passwords containing exactly two different classes of characters are bucket 2, etc. In our first sample of 100,000 passwords, there were very few lexically diverse passwords. RockYou has $< 0.2\%$ passwords with characters from all of the four character classes. So we sample with replacement 5,000 passwords for each bucket from the empirical distribution of passwords in RockYou restricted to the passwords corresponding to the bucket. We performed the same typing experiment as described before but with new HITs created from these newly sampled passwords. In the left graph

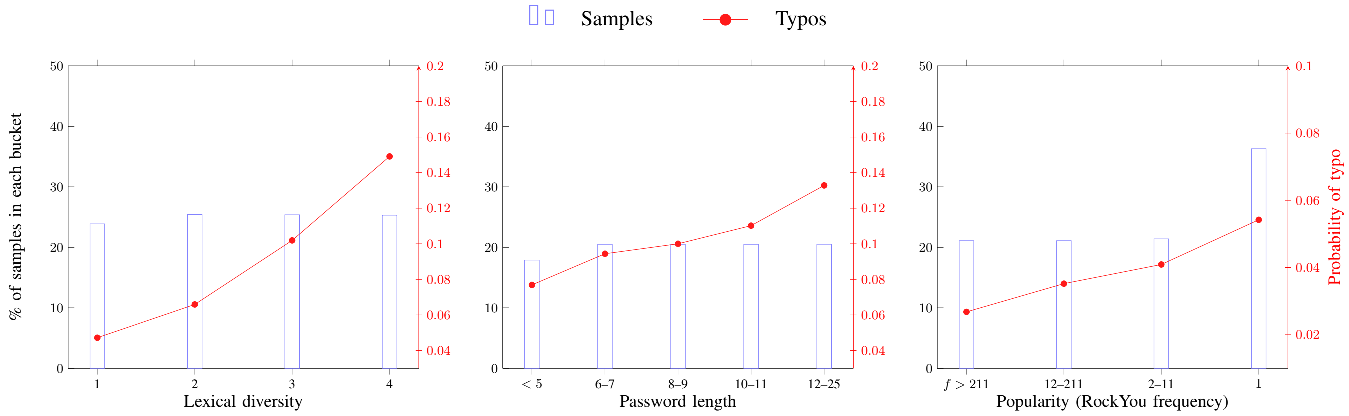

Fig. 9: Three experiments showing typo frequency relative to various partitions of passwords into buckets. Bucket size is indicated on the left of each figure, and corresponding typo rates on the right. **(Left)** Passwords are partitioned into four buckets based on diversity of character types. For each bucket we report the percentage of samples (blue bars) that fall in that bucket and what fraction of those samples are mistyped (red line). **(Middle)** Passwords are categorized into buckets by increasing order of length. **(Right)** Passwords are assigned to buckets by decreasing frequency (increasing unpopularity) in RockYou. Bucket frequency ranges are selected so that each bucket has roughly an equal number of samples.

of Figure 9, we present the percentage of passwords in each bucket that were mistyped.

Password length: We divide passwords into five groups based on their lengths, namely ≤ 5, 6–7, 8–9, 10–11, and ≥ 12. For each class, we compute the percentage of samples that lie in that class, along with the percentage of passwords in those samples that were mistyped. In the middle graph of Figure 9 we show these numbers for each of the length groups. As one might expect, typo likelihood grows with password length.

Password popularity: We sort the list of sampled passwords for our MTurk experiment based on their frequency counts in the RockYou leak. (Ties were broken alphabetically.) We then split the passwords into four buckets, adjusting their corresponding frequency ranges to ensure that buckets are of roughly equal size. (Some unevenness was unavoidable, as many passwords occur only once in the Rockyou leak.) For each bucket, we present the number of mistyped passwords in the right graph of Figure 9. We can see the clear trend that passwords that are popular among RockYou users are more likely to be typed correctly. For example, passwords used by more than 211 users are 1.5 times more likely to mistyped than those used by only one user.

Discussion: password typing complexity. As noted above, lexical diversity, length, and popularity are related metrics. Inspection of the passwords within the various buckets used in the charts of Figure 9 reveals that there is significant overlap between them. As one example, 18% of the passwords with lexical-diversity bucket 4 *both* have length $\geq$ 12 *and* are unpopular ($f = 1$).

The three metrics together highlight different aspects of the underlying and intuitive trend: some passwords are more difficult to type than others. It appears, moreover, that typos are more likely to surface in harder-to-guess passwords. Consequently, typo correction could help encourage users to adopt stronger passwords by easing the use of such passwords. We leave rigorous study of this hypothesis to future work, but note that it offers further potential motivation for our work.

D. Typist Speed and Typo Rate

As an enhancement of our experimental results in Section III, we report on two experiments that provide further illumination of password features that lead to typos. These experiments further emphasize our observation that typo rates appear to increase with password complexity.

Typist and typo likelihood. In our MTurk experiments, we timestamped each character as it was typed during the experiments. We sorted the workers based on their average typing speed (characters-per-minute) and binned workers into four quartiles. For each quartile, we consider the subset of passwords that were typed by the typists whose speed falls in that quartile, and we compute the fraction of passwords that were mistyped in that subset. The data is reported in Figure 10. We found that slow typists make more mistakes than faster typists. It could be that faster typists are also more skilled and so less likely to make mistakes.

Password entry time. We binned the passwords into four quartiles based on the time required to type those passwords. The fraction of typos in each of that quartile is reported in the middle chart of Figure 11. Passwords that required more time on average to type are more likely to be mistyped.

E. Computing $\lambda_q^{\text{greedy}}$

We start by showing that computing the optimal q guesses to make against a relaxed checker is NP-hard in general. Later, we present an efficient approximate algorithm for the problem.

Definition 1. Best-q-guess. *Given a function $B : \mathcal{S} \to \mathcal{PW}^*$, a password distribution p over $\mathcal{PW} \subseteq \mathcal{S}$, and a query budget*

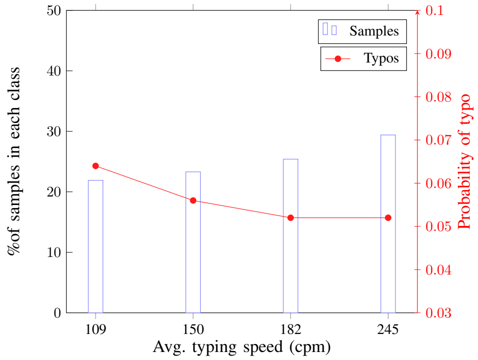

Fig. 10: The workers are divided into four quartiles based on their typing speed, and for each quartile we report the percentage of passwords that were mistyped.

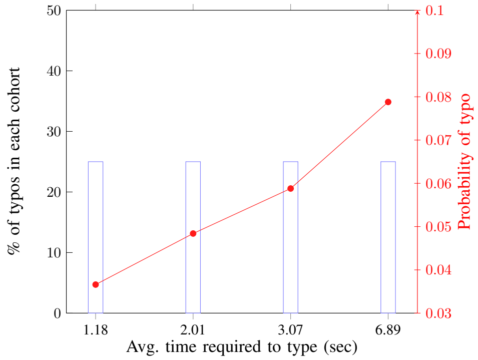

Fig. 11: Passwords are divided into four quartiles based on the amount of time spent by workers to type them, and then compute the fraction of passwords mistyped in each quartile.

q, find a q-size subset $P \subseteq \mathcal{S}$, such that $\sum_{w\in C'} p(w)$ is maximized, where $C' = \bigcup_{\tilde{w}\in P} B(\tilde{w})$.

Definition 2. Maximum coverage problem. *Given a ground set $E = \{e_1, e_2, \dots e_n\}$, a collection of m subsets of E, $S = \{S_1, S_2, \dots, S_m\}$, and a weight function $\gamma : E \to \mathbb{R}^+$ that assigns weights to each element $e \in E$, find a subset $C \subseteq S$ of size k, such that the following quantity is maximized, $\sum_{e\in C'} \gamma(e)$, where $C' = \bigcup_{S_i\in C} S_i$.*

The maximum coverage problem is known to be NP-hard [20]. We can thus prove the following theorem.

Theorem 3. *If there is a polynomial time algorithm for best-q-guess, there exists a polynomial time algorithm for maximum coverage problem.*

Proof: We shall show a polynomial time reduction from the maximum coverage problem to the best-q-guess problem. To start with, we are given an instance of maximum coverage problem with (E, S, γ, k), and we want to construct an instance of best-q-guess problem. To do so, we set $\mathcal{PW} = E$ and probability p as proportional to γ. (We might have to normalize γ to make it a probability distribution.) The function $B : \mathcal{S} \to \mathcal{PW}^*$ is defined as follows. First add to $\mathcal{S}$ a set $W^* = \{\tilde{w}_i^*\}_{i=1}^m$, with $p(\tilde{w}_i^*) = 0$, and for each $S_i \in S$, set $B(\tilde{w}_i^*) = S_i \bigcup \{\tilde{w}_i^*\}$. For all other $\tilde{w} \in \mathcal{S} \setminus W^*$, let $B(\tilde{w}) = \{\tilde{w}\}$. This is a valid instance of the best-q-guess problem, and if we can find an polynomial time computable solution to this, we can solve the maximum weighted set cover problem in polynomial time. ▮

```
nextPw()
Returns the passwords in $\mathcal{PW}$ in decreasing order of
their probability $p$.

FindGuesses($q$)
/* $B(\tilde{w})$ = ball around $\tilde{w}$, and $b = \max_{\mathcal{S}} |B(\tilde{w})|$ */
/* $N(w) = \{\tilde{w} \mid w \in B(\tilde{w})\}$ */
$P \leftarrow \mathcal{PW}$
$A \leftarrow$ MaxHeap()     /* val($\tilde{w}$) = $p(B(\tilde{w}) \cap P)$ */
$g \leftarrow \phi$;
do {
    $w \leftarrow$ nextPw()
    $\tilde{w}_m \leftarrow A$.popmax()
    while  $p(B(\tilde{w}_m) \cap P) \geq b \cdot p(w)$ {
        $g \leftarrow g \cup \{\tilde{w}_m\}$
        $P \leftarrow P \setminus B(\tilde{w}_m)$
        foreach  $\tilde{w} \in \{\tilde{w} \in A \mid B(\tilde{w}) \cap B(\tilde{w}_m) \cap P \neq \phi\}$
            $A$.updateweight ($\tilde{w}$)
        $\tilde{w}_m \leftarrow A$.popmax()
    }
    $A$.heappush($\tilde{w}_m$)
    foreach  $\tilde{w} \in (N(w) \setminus A)$
        $A$.heappush($\tilde{w}$)
} while ($|g| < q$)
return $g$
```

Fig. 12: Figure presents a greedy algorithm to compute the best q guesses and thereby compute $\lambda_q^{\text{greedy}}$, for an attacker who estimates the the password distribution with p.

Nevertheless, a greedy algorithm can achieve $(1 - \frac{1}{e})$ times the optimal λ^{fuzzy} value and, as shown by Feige [20], [34], yields an optimal approximation for the maximum coverage problem. Naïve implementation of this greedy algorithm for best-q-guess, however, requires searching over exponentially many strings in $\mathcal{S}$. We can exploit the fact that in our setting, a small number of correctors are used, and these correctors are efficiently invertible. Additionally, password weights are highly non-uniform. Thus it is efficient to enumerate all balls above a certain threshold weight, yielding the efficient implementation of Feige's greedy approximation algorithm specified in Figure 12. The algorithm intuition is this: as an invariant, in any iteration of the external while loop, all new balls (balls of $\tilde{w}$) pushed onto the heap have max-weight password w, and hence total weight $\leq b \cdot p(w)$. This observation enables a global selection of balls in weighted order.

F. Proofs

We restate and then prove the free correction theorem from Section V-D.

Theorem 1 (Free Corrections Theorem). *Fix some password distribution p with support $\mathcal{PW}$, a typo distribution τ, $0 <$*

$q < |\mathcal{PW}|$ and an exact checker ExChk*. Then for* OpChk *with any set of correctors $\mathcal{C}$, it holds that $\lambda_q^{\text{fuzzy}} = \lambda_q$.*

Proof: Let $\hat{S}$ be the optimal set of q strings which maximizes the total acceptance rate for the given checker OpChk. (Note that the order in which the queries are made does not change the success probability.) Let $B(S) = \cup_{\tilde{w} \in S} B(\tilde{w})$ for some set S of strings in $\mathcal{S}$. Recall that $\lambda_q = \sum_{i=1}^{q} p(w_i)$ is the sum of the probabilities of q most probable passwords in $\mathcal{PW}$. On the other hand, $\lambda_q^{\text{fuzzy}} = p\left(B(\hat{S})\right)$. The above holds because $B(\tilde{w})$ is the set of passwords checked by OpChk for a given string $\tilde{w}$.

The checker OpChk ensures that the cumulative probability mass of any ball is less than or equal to $p(w_q)$ whenever the size of the ball is more than 1, but, if the size is one, the cumulative probability can be more than $p(w_q)$. So, we split $\hat{S}$ into two distinct groups $\hat{S}_1$ and $\hat{S}_{>1}$, where $\hat{S}_1$ is the set of all strings in S whose ball sizes are exactly one, and $\hat{S}_{>1} = \hat{S} \backslash \hat{S}_1$. We can claim following two inequalities.

$$p\left(B(\hat{S}_1)\right) \leq \sum_{i=1}^{|\hat{S}_1|} p(w_i) \tag{1}$$

$$p\left(B(\hat{S}_{>1})\right) \leq |\hat{S}_{<1}| p(w_q) \leq \sum_{i=|\hat{S}_1|+1}^{q} p(w_i) \tag{2}$$

Equation (1) is true because the $|B(\hat{S}_1)| = |\hat{S}_1|$, and the right hand side is the highest cumulative probability that any set of that size can achieve under p. Equation (2) is true because of the facts that $p(B(\tilde{w})) \leq p(w_q)$ for all $\tilde{w} \in \hat{S}_{>1}$, and $p(w_i) \geq p(w_q)$ for all $i \geq q$. So, by a union bound over $B(\hat{S}_{>1})$, we can achieve that inequality. We can add the two inequalities to obtain our desired result.

$$p\left(B(\hat{S})\right) = p\left(B(\hat{S}_1)\right) + p\left(B(\hat{S}_{>1})\right) \leq \sum_{i=1}^{q} p(w_i)$$

To show strict equality, simply observe that an attacker against OpChk can always choose the q most probable passwords to guess and achieve a success rate of λ_q. Thus $\lambda_q^{\text{fuzzy}} = \lambda_q$. ▌

Theorem 2. *Fix $q > 0$, a distribution pair (p, τ), and a corrector set $\mathcal{C}$. Define* OpChk *to work over $\mathcal{C}$ and let* Chk *work for a set of correctors $\mathcal{C}' \subseteq \mathcal{C}$. If $\Delta_q(\mathsf{Chk}) = 0$, then* $\mathsf{Util}(\mathsf{Chk}) \leq \mathsf{Util}(\mathsf{OpChk})$.

Proof: First recollect utility of any checker Chk is defined as

$$\begin{aligned} \mathsf{Util}(\mathsf{Chk}) &= \Pr[\mathrm{ACC}(\mathsf{Chk}) \Rightarrow \mathsf{true}] \\ &= \sum_{\tilde{w} \in \mathcal{S}} \sum_{w \in B(\tilde{w})} p(w) \cdot \tau_{\tilde{w}}(w), \end{aligned}$$

where $B(\tilde{w})$ is the ball of $\tilde{w}$ under Chk.

Let assume for contradiction that there exists a checker Chk which uses only the correctors in $\mathcal{C}$, achieves a $\Delta_q(\mathsf{Chk}) = 0$ and still beats the OpChk in utility, that is, $\mathsf{Util}(\mathsf{Chk}) > \mathsf{Util}(\mathsf{OpChk})$. Let denote a ball of OpChk by $B(\cdot)$ and a ball of Chk by $\tilde{B}(\cdot)$. So, if $\mathsf{Util}(\mathsf{Chk}) > \mathsf{Util}(\mathsf{OpChk})$, then there exists at least one $\tilde{w} \in \mathcal{S}$ such that

$$\sum_{w \in \tilde{B}(\tilde{w})} p(w) \cdot \tau_w(\tilde{w}) > \sum_{w \in B(\tilde{w})} p(w) \cdot \tau_w(\tilde{w}). \tag{3}$$

Now, by construction, the optimal checker OpChk selects the $B(\tilde{w})$ that maximizes the utility under the constraint that no ball of size 1 has higher cumulative mass than $p(w_q)$. Here by utility we mean the sum $\sum_{w \in B(w)} p(w) \cdot \tau_w(\tilde{w})$. (See Eqn. V-D.) The checker Chk can achieve higher utility only if it violates one of the two constraints in (V-D). The first constraint, required for completeness, is inviolable. The second constraint determines security; if $p(\tilde{B}(\tilde{w})) > p(w_q)$ when $|\tilde{B}(\tilde{w})| > 1$, then the security loss $\Delta_q(\mathsf{Chk}) > 0$ according to Lemma 1. Thus there cannot exist any $\tilde{w} \in \mathcal{S}$ fulfilling Eqn. 3. Thus the assumption $\mathsf{Util}(\mathsf{Chk}) > \mathsf{Util}(\mathsf{OpChk})$ is false. ▌

Lemma 1. *For any password and typo distribution pair (p, τ), checker* Chk*, and parameter $0 < q < |\mathcal{PW}|$, if there exists a string $\tilde{w} \in \mathcal{S}$, s.t. $|B(\tilde{w})| > 1$ and $p(B(\tilde{w})) > p(w_q)$, then $\Delta_q > 0$.*

Proof: Security loss $\Delta_q > 0$ implies that $\lambda_q^{\text{fuzzy}} > \lambda_q$. Let $\mathcal{PW}_q = \{w_1, \ldots, w_q\}$ and recall that $\lambda_q = p(\mathcal{PW}_q)$. Recall too that:

$$\lambda_q^{\text{fuzzy}} = \max_{S \subseteq \mathcal{S}} p(B(S)), \quad \text{where } |S| = q.$$

First set $S \leftarrow (\mathcal{PW}_q \backslash B(\tilde{w})) \cup \{\tilde{w}\}$. Clearly $\lambda_q^{\text{fuzzy}} \geq p(B(S))$. If we look at the union of balls of the strings in the set S,

$$\begin{gathered} B(S) \supseteq \mathcal{PW}_q \cup B(\tilde{w}) \\ \Rightarrow p(B(S)) \geq p(\mathcal{PW}_q) + p(B(\tilde{w}) \setminus \mathcal{PW}_q) \end{gathered}$$

Now, if $B(\tilde{w}) \setminus \mathcal{PW}_q \neq \phi$, then clearly $\lambda_q^{\text{fuzzy}} \geq p(B(S)) > p(\mathcal{PW}_q)$, and so $\Delta_q > 0$.

If $B(\tilde{w}) \setminus \mathcal{PW}_q = \phi$, then $|S| < q$, as $|B(\tilde{w})| > 1$. Thus as long as there exists a password $w' \in \mathcal{PW} \backslash S$ such that $p(w') > 0$, we can add w' to S, resulting in $p(B(S)) > p(\mathcal{PW}_q)$. This concludes the proof. ▌

G. Toy Example of Poor Ball Estimation

Consider the following toy example of the attacker's estimated distribution $\hat{p}$ and the actual challenge distribution p:

Attacker's estimate		Actual distribution	
w	$\hat{p}(w)$	w	$p(w)$
`123456`	$1/3$	`123456`	$1/2$
`password`	$1/4$	`password`	$1/5$
`Password`	$1/4$	`Password`	$1/5$
`qwerty`	$1/6$	`asdffghj`	$1/10$

The best guess of the attacker against ExChk is `123456`, which yields success rate $1/2$. If the attacker wants to optimize her guess in the presence of a typo tolerant checker, e.g., $\mathsf{Chk\text{-}All}$ with correctors $\mathcal{C}_{\text{top2}}$, the she select as her first guess is `password` (in whose ball `Password` lies), yielding success probability only $2/5$.