Making more extensive and efficient typo-tolerant password checkers

Anonymous authors
Redacted institutes

***Abstract*—As passwords remain the main online authentication method, focus has shifted from naive entropy to how usability improvements can increase security. Chatterjee et al. recently introduced the first two typo-tolerant password checkers, their second being usable in practice while being able to correct up to 32% of typos, with no real security cost.**

We propose an alternative framework which corrects up to 57% of typos without affecting user experience, at no computational cost to the server.

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

Despite recent advances in biometric authentication [1] and account linking [2], passwords are still the main method of authentication used online and will probably remain so in the near future. Countless studies have been written on the pitfalls of password-based authentication [3], [4], initially focusing on low security, with users creating bad passwords [5] and repeatedly dodging security measures [6]–[8], but also service providers ignoring best practices on how to secure password databases [9]. More recently, research on how to make them more usable has made advances [10], [11], and some of the effects of bad password policies are being reversed [12], to focus on longer passwords. Unlike random passwords with special characters which suffer from low memorability [13], long and simple passwords and passphrases [14]–[16] can benefit from humans' superior ability to memorise strings that make sense, improving both security and usability [17], [18]. As authentication becomes an omnipresent task, being refused access is increasingly frustrating, with forgetting one's password being perceived about as frustrating as forgetting one's keys [19]. Moreover, just as users sometimes forget their passwords, they often mistype them. To prevent some of this frustration and improve usability, some services like Facebook have discreetly adopted typo correction for the 2-3 most frequent typos, such as forgetting the caps lock or capitalising the first character of a password on a mobile device [20].

In an innovative paper in 2016 [21], Chatterjee et al. discovered that a vast majority of authentication failures comes from a few simple typos, and that it could turn 3% of the users away. They developed a first typo-tolerant password checker which was highly secure (and computationally intensive) but could only correct about 20% of typos. The same team developed a second system called TypTop [22], which is efficient both computationally and memory-wise, and corrects up to 32% of typos. This system works by keeping a cache of allowed password hashes corresponding to the frequent typos made by the user, and updates this cache at each successful authentication. Using a different approach, Blanchard also proposed a simple theoretical method based on homomorphic encryption that is too computationally expensive to be usable in practice [23]. Finally, Woodage and some of the original authors created a new distribution-sensitive scheme that adjusted the error rate and hashing time, improving the resistance to certain attacks and providing better time/security trade-offs [24].

Those systems can actually have a positive impact on security as they make long passwords — which are more error-prone — much more usable, lowering the cost of using highly secure passwords. The issue with the schemes proposed is that they are technically complex, which often creates difficulties in the implementation [25], [26]. As such, we wondered whether similar performances could be attained with simpler designs, and how to create a system that increased the usability even further, while satisfying the following constraints:

- *Usability*: the system should make it easier to log into a service (by correcting as many legitimate typos as possible).
- *Security*: the system should have similar resistance to present frameworks against known attacks on passwords.
- *Efficiency*: the system should require as little computation, communication and storage as possible.

Main results: Based on a completely different design, we introduce a simple typo-tolerant framework that satisfies the different constraints mentioned. It improves usability by correcting up to 57.8% of total typos, or up to 91.2% of acceptable typos. It is efficient, requiring limited client-side and next to no server-side computation, as well as low communication bandwidth and limited storage. It is simple, being easily implementable and compatible with other systems, as well as being retro-compatible with other frameworks. Finally, it is secure, limiting the risks of both credential spoofing and credential theft.

Structure of the paper: This paper starts by an analysis of the typos shown in Chatterjee et al.'s initial study [21], which we recompute as the classification is not directly compatible with the analyses in our model (although they remain mostly comparable). We then describe the intuition behind the framework we propose, followed by the algorithms themselves and the security analyses. We conclude and discuss those results, and the appendix features an example run of the algorithm on each potential error.

II. Typology of errors

Before introducing our framework, we want to provide an analysis of the most frequent user errors, as studied in [21], [22]. As motivation for the first error-tolerant password checker, Chatterjee et al. ran an experiment using Mechanical Turk to look at the types of errors committed by users typing other people's password. They published a summary analysis with their algorithm in [21] and made the data publicly accessible.

In the original study, the authors chose to only look at strings whose *Damerau-Levenshtein distance* [27] was less than 2, as well as errors where the caps lock was inverted for the whole string. We decided to run a more detailed analysis of the first data-set, shown in Table I. Some of the errors considered in [21] would probably not happen in the real world, mostly inserting spaces and transcription errors — such as confusing "1" and "l". This creates differences between analyses, but both agree that handling caps lock as well as single substitution, transposition, insertion and shifting errors would handle 65% to 73.9% of errors.

Typo category	Wrong password %
Single substitution	31.3
QWERTY neighbour	14.7
Numpad neighbour	0.6
Single shift	9.0
Single deletion	20.5
Caps lock	15.5
Single insertion	13.9
Space	2.1
Duplicated letter	4.0
Single transposition	4.2
Other	15.7

TABLE I
Types of typos recomputed on the original data-set from [21], over all passwords at distance at most 6 from the original, plus complete capitalisation errors.

Two main questions arise when looking at such data: which errors are legitimate typos, and which legitimate typos should be corrected. Considering the length of passwords in the database, we chose to look at Levenshtein distances up to 4, discounting transcription errors. . From this, the set of *acceptable typos* will correspond to typos at distance at most 2, except ones involving deletions or substitutions by a distant character. We chose to exclude both, as deletions would greatly increase the risk of targeted attacks as shown in the next section, and to only allow proximity substitutions. Such a substitution happens when the key pressed is one of the six keys closest to the original one (two above, two below, and one on each side).

Chatterjee et al.'s model generally does not seek to correct transcription errors, so our set of errors is almost a superset of theirs. Thus, although we use slightly different metrics and proportions, comparing proportions directly between their model and ours can only reduce the difference in corrected proportions (i.e., we correct 57% in our model, and they correct 32% in their model, which would be less than 32% in our model). The model we present then corrects the following: substitutions of adjacent characters (15.3%), single capitalisation (9.0%), full capitalisation (15.5%), single transposition (4.2%), insertion of spaces or duplicate characters (2.1% and 4.0%). We finally also choose to correct the remaining arbitrary insertions (7.7%) as it does not have a large impact on security, for a total of 57.8%. This is when compared to the total number of typos that does not exclude deletions and arbitrary substitutions. The *unacceptable typos* represent 36.6% of all typos, leaving only 5.6% of typos left that are neither corrected nor immediately dangerous to correct.

III. A typo-tolerant framework

A. Definitions and general intuition

This framework is a set of three algorithms: one to create a password (*key-setting*), one for the user to compute and send their password to the server when asked their credentials (*key-sending*), and the last one for the server to check whether the credentials received should be accepted (*key-checking*). The framework works with a variety of typo tolerance policies, such as only accepting capitalisation errors, or only certain forms of keyboard proximity errors (accepting an "r" instead of "e", but not a "d").

There are of course many different potential frameworks. For example, the simplest efficient typo-checking framework would consist in storing the value for both the hash of the normal password and the hash of the string corresponding to the same password in caps lock. More than 15% of typos could be handled this way, at the cost of storing and comparing a single additional hash. The simplest complete system is to store — or send — hashes corresponding to all possible typos. The problem is that, depending on the typos corrected, this system requires the storage or communication of hundreds of hashes, making the system less efficient and more vulnerable to random collisions.

The framework shown is in fact the third iteration of a process where each step corresponds to a framework that handles more typos than the previous one. The first and simplest step only addresses the most frequent typo: single adjacent substitution errors, where one character in the password is replaced by another neighbouring character. For example, an "e" could be replaced by an "r", a proximity error that should be accepted, whereas replacing "e" by "m" should lead the algorithm to reject. We also rely on the agreement over a canonical *keyboard map*, assigning every key-press to an integer. For example, one could use the JavaScript key codes, whose main list goes from 8 till 255, but less than 100 of those numbers correspond to usual keys. Instead of the layout, the keyboard map depends on a map from key-presses (such as "a" or "SHIFT+a"). The initial framework of the series broadly works as follows:

1) The password of length n is split into n partial passwords, each missing one character.

2) The partial passwords are concatenated with a salt[1] before being hashed.
3) Pseudorandom permutations within the set of character codes are computed (generally $[0, 255]$), based on the hashes, using Brassard's algorithm [28].
4) Each excluded character and all the adjoining ones on the keyboard are encoded using the corresponding permutation.
5) The user sends the login message, a list of n (hash, number list) pairs.
6) If one hash is correct, and the stored number is in the corresponding list, the server authenticates the user.

The framework shown here extends this idea and adds insertion and transposition tolerance, by removing two adjacent characters and computing the hashes, and sending the images of the missing characters through different permutations. The three algorithms of the framework are shown in the following pages: key-setting in Algorithm 1, key-checking in Algorithm 2 and key-sending in Algorithm 3. Then comes some reflections on the design choices, and security properties in the following section. A more detailed explanation of the algorithms is found in the Appendix.

Data: Username $NAME$, Password P of length n
Keyboard map M: Keys $\rightarrow [0, 255]$
Result: Main hash and lists of (hash / integer) and (hash / integer list) pairs
begin
  $S[0] \longleftarrow$SHA3-256$(NAME)$
  for i from 1 to 5 **do**
    $S[i] \longleftarrow$SHA3-256$(S[i-1])$
  $H_0 \longleftarrow$Argon2(Concatenate$(S[0], P)$)
  if $n < 10$ **then**
    return H_0 /* Preventing general typo correction on very short passwords. */
  else
    while *Length* $(P) \geq 16$ **do**
      P.append(S[0][0]) /* Making the passwords have uniform minimum length of 16. */
    for i from 1 to n **do**
      $PA_i \longleftarrow P \setminus P[i]$
      $HA_i \longleftarrow$Argon2(Concatenate$(S[1], PA_i)$)
      Random_bits $\longleftarrow$ SHA3-256(Concatenate$(S[2], PA_i)$
      $\pi_i \longleftarrow$ Brassard(Random_bits)
      $K_i \longleftarrow \pi_i(M(P[i]))$
    for i from 1 to $n-1$ **do**
      $PB_i \longleftarrow P \setminus \{P[i] \bigcup P[i+1]\}$
      $HB_i \longleftarrow$Argon2(Concatenate$(S[1], PB_i)$)
      for j from 1 to 4 **do**
        Random_bits[j] $\longleftarrow$ SHA3-256(Concatenate$(S[j+1], P_i)$)
        $\pi_{i,j} \longleftarrow$Brassard(Random_bits[j])
      $KA_i \longleftarrow [\pi_{i,1}(M(P[i]))]$
      $KB_i \longleftarrow [\pi_{i,2}(M(P[i+1]))]$
      $KC_i \longleftarrow [\pi_{i,3}(M(P[i]))]$
      $KD_i \longleftarrow [\pi_{i,4}(M(P[i+1]))]$
    return $(H_0, (HA_i, K_i)_{1 \leq i \leq n}, (HB_i, KA_i, KB_i, KC_i, KD_i)_{1 \leq i \leq n-1})$

Algorithm 1: Key-setting algorithm

Data: Length n, Original hash H, Original list (HA_i, K_i)
Original list $(HB_i, KA_i, KB_i, KC_i, KD_i)$
Received hashes H_0 and H'_0 and list $(H'_i, LA_i, LB_i, LC_i, LD_i)$
Result: ACCEPT if and only if the password has at most one acceptable typo.
begin
  if $H = H_0$ *OR* $H = H'_0$ **then**
    return ACCEPT
  else
    if $n < 10$ **then**
      WAIT(RANDOM(0.1-1))/* in ms, against timing attacks */
      return REJECT
    else
      for i from 1 to $n-1$ **do**
        if $HB_i = H'_i$ **then**
          for j from 1 to $|LA_i|$ **do**
            if $(LA_i[j] = KA_i$ AND $LB_i[1] = KB_i)$ OR $(LB_i[j] = KB_i$ AND $LA_i[1] = KA_i)$ **then**
              return ACCEPT
          if $LC_i[1] = KC_i$ AND $LD_i[1] = KD_i$ **then**
            return ACCEPT
        else
          if $HA_i = H'_i$ AND $LB_i[2] = KB_i$ **then**
            return ACCEPT
        if $HA_n = H'_n$ AND $LB_n[2] = KB_n$ **then**
          return ACCEPT
      WAIT(RANDOM(0.1-1))
      return REJECT

Algorithm 2: Key-checking algorithm

Data: Username $NAME$, Password P of length n, Keyboard map M: Keys $\rightarrow [0, 255]$
Result: Two main hashes and list of (hash / integer list) pairs
begin
  $S[0] \longleftarrow$SHA3-256$(NAME)$
  for i from 1 to 5 **do**
    $S[i] \longleftarrow$SHA3-256$(S[i-1])$
  $P' \longleftarrow$Invert_caps_lock(P)
  $H_0 \longleftarrow$Argon2(Concatenate$(S[0], P)$), $H'_0 \longleftarrow$Argon2(Concatenate$(S[0], P')$)
  if $n < 10$ **then return** (H_0, H'_0) /* Only sending caps lock for short passwords. */
  else while $|P| < 16$ **do** P.append(S[0][0])
  for i from 1 to $n-1$ **do**
    while |Neighbours$(P[i])| <$ MAX_NEIGHBOURS **do**
      Neighbours$(P[i]) \longleftarrow$ any k with $k > \max_l(M(l))$
      /* Making the neighbours lists have uniform length by adding dummy characters. */
  for i from 1 to $n-1$ **do**
    $P_i \longleftarrow P \setminus \{P[i] \bigcup P[i+1]\}$
    $H_i \longleftarrow$Argon2(Concatenate$(S[1], P_i)$)
    for j from 1 to 4 **do**
      Random_bits[j] $\longleftarrow$SHA3-256(Concatenate$(H_i + S[j+1])$)
      $\pi_{i,j} \longleftarrow$Brassard(Random_bits[j])
    $LA_i \longleftarrow [\pi_{i,1}(M(i)), \pi_{i,1}(M(SHIFT(P[i])))]$
    foreach $j \in$ Neighbours$(P[i])$ **do**
      LA_i.append$(\pi_{i,1}(M(j))$
    LA_i.sort()
    $LB_i \longleftarrow [\pi_{i,2}(M(i+1)), \pi_{i,2}(M(SHIFT(P[i+1])))]$
    foreach $j \in$ Neighbours$(P[i+1])$ **do**
      LB_i.append$(\pi_{i,2}(M(j))$
    LB_i.sort()
    $LC_i \longleftarrow [\pi_{i,3}(M(P[i+1]))]$, $LD_i \longleftarrow [\pi_{i,4}(M(P[i]))]$
  return $(H_0, H'_0, (H_i, LA_i, LB_i, LC_i, LD_i)_{1 \leq i \leq n-1})$

Algorithm 3: Key-sending algorithm

B. Design choices and optimisations

The question of which proximity errors should be allowed is quite simple, but when it comes to inserted letters it becomes non-trivial. For example, duplicated letters or added spaces seem like good candidates, whereas letters far from the

[1]The salt here can be any arbitrary string, using the login plus a number works, the main goal being to avoid precomputed tables.

nearby keys might not be legitimate typos. Additionally, some insertions go with other typos, especially with shift errors. This happens when, instead of hitting the shift key followed by the targeted letter, the user hits a key next to the shift key, committing a double typo.

Instead of a permutation, a function from $[0, 255]$ to a greater set could also be used, as it could increase the security by reducing the probability that an adversary could guess the correct number. This is a trade-off between simplicity, efficiency, and security. The main advantage is that it would lower the success probability of attacks with hashes of different dictionary words. This is not relevant as the advantage of this type of attacks over dictionary attacks is limited in scope by the low probability of getting a correct number in the list ($\leq \frac{7}{255}$).

If we don't allow double proximity errors, LB_i is redundant with LA_{i+1}, and all single-character typos that are not on the last letter of the password could be corrected using only LA_{i+1}. We still include it as it only marginally increases computing costs client-side and increases communication costs by at most 19%.

We call Brassard's algorithm to lazily get the permutation by computing the image of an element only when it is needed (instead of computing all images at the initialisation, e.g. through the Fisher-Yates algorithm [29]). In our case, we require 8 pseudorandom bits per element. We need the images of $k = |\text{Neighbours}(P[i])|$ random element chosen uniformly among all possible permutations in a deterministic way dependent on the seed. Fisher-Yates' algorithm would require about 713 random bits if implemented correctly[2], which could be attainable using a longer salt for the seed (hundreds of bits) and a PRNG with variable output. Using Brassard's algorithm [28], we require at most 8 bits per call, and at most 80 bits in the calls made by the key-sending algorithm. This allows us to use most PRNG with fixed output length.

The presence of the full hash H_0 is not strictly necessary, but it allows the server to check if everything is right in one comparison. An alternative would be to check $(H_1, L[1])$ and $(H_2, L[2])$, thus detecting the presence of an error, in which case at least one of the hashes would be incorrect. The other hashes can be checked lazily if both tests lead to rejection.

IV. Security analysis

As we seek to improve authentication systems, we have two goals: preventing people without correct credentials from logging in, and preventing people with — potentially illegitimate — access to the database from getting the credentials of other users. This second point is crucial, as credential stuffing attacks — where an adversary steals a list of login/password pairs on an unsecured website and tests them systematically on other websites — are increasingly frequent, with up to 91% of login attempts coming from credential stuffing, of which on average 0.50% are successful [30].

A. Preventing access

As we tolerate certain typos, we have an inevitable increase in the probability of a successful login attempt by an adversary. Which typos are allowed is then a crucial decision. For example, allowing single deletions might seem like a good idea: it corresponds to many typos, and only reduces the entropy by a limited amount (around 5 bits on average). However, this would be extremely detrimental in one important case: partial password re-use. As users become aware of credential stuffing, some make small variations to prevent such automated attacks [31], [32]. Accepting deletions makes such attacks much more likely to succeed, which is why a substitution — being very similar to a deletion in terms of security — should only be accepted if the substituted letter is a neighbour of the original. As long as the adversary follows the protocol, the security loss entirely comes from the fact that more passwords are allowed. With a generally lax typo-tolerance system this means that the set of acceptable strings goes from 1 to around 100 for a 12-character password[3]. This makes bruteforce and dictionary attacks somewhat easier, but as countermeasures are shifting the online setting away from those and towards more refined attacks, this should not be a risk for users with passwords of reasonable strength. Typo correction also makes it easier to use safer, longer passwords — which come with a higher risk of typos.

The goal here is to prove that the security loss mostly comes from the added typos, without creating additional security risks. In other words, the framework should not reduce the security much beyond accepting the allowed typos. This is done by proving the following lemma in which *smart bruteforce* means that the bruteforce follows the frequent password list by decreasing frequency.

Lemma 1. *Using only the username and knowledge of the framework, finding a correct authentication message for a password of length ≤ 16 takes in expectation at least $\frac{1}{114}$ times as many queries as a smart bruteforce attack against a system without typo correction.*

Remark 1. *Although the bound of $\frac{1}{114}$ seems bad, there are two reasons that explain and compensate for this. The first is that a query in this system corresponds to a set of queries in a standard system, so the number of queries naturally goes down (but the bound on the number of queries accepted by the server before triggering an alarm should go down accordingly). The second point is that for this bound to be reached, the bruteforce algorithm must be able to distribute queries in an optimal way to make full use of the complex query.*

Remark 2. *The lemma here could also be applied to passwords of length strictly greater than 16, but this is unnecessary*

[2]The information lower bound is 373 bits, but low-efficiency implementations that require a new random integer at each call would require up to 6400 random bits.

[3]This discounts insertions as the benefit from testing longer passwords is anecdotal.

as these passwords are generally not vulnerable to the attacks considered.

1) Intuition: There are two ways an adversary could obtain access if they have no prior information besides the username. The first is to take a set of passwords and send each through the key-sending algorithm, to gain access with either the password itself or a version with an allowed typo. The second is to fake the algorithm's outputs and send at least partially incorrect messages to the server, in an attempt to attack the hash directly.

Let's suppose that an adversary decides to send partially inauthentic login queries. Each query is composed of a main hash, and a set of (hash, number lists) pairs. All the hashes are salted, and the hash space — using for example SHA3-256 — is much greater than the usual password space. This means that sending a random string instead of a real hash can be made to have a lower probability of success (per time unit) than computing a real password hash. For example, assuming a very generous bound of 160-bit passwords (uniformly random password on 20 ASCII characters), it would still take at least 10^{26} login queries before having a reasonable chance of getting a correct hash, evidently costing more than computing one of the correct hashes[4]. Taking a more realistic bound on passwords would only decrease the success probability. As the limiting factor lies in the number of queries, an adversary trying to maximise their chance of success would accurately compute all the hashes in the query.

Because sending random hashes is not efficient, an adversary could instead send the same hash in multiple positions, with different additional letters each time. This way, they could cover all possibilities for a single missing letter in only two or three login queries. The checking system couldn't easily prevent this, as common hashes would be possible (for example, the password "*encoded*" has two identical hashes at the end corresponding to removing either "de" or "ed"). Moreover, $n-1$ correct hashes could be computed and then checked in parallel through interweaving.

This effectively increases the efficiency of an adversary by testing multiple passwords per login query. The main deterrent against such attacks is a limit on the number of queries accepted by the service provider (or rate limiting). As the method proposed greatly increases the probability of a user logging in successfully when they make a typo, the maximum number of queries allowed can be reduced accordingly without lowering the usability. Additionally, one could make a counter for a given hash to prevent bruteforcing them: if the server receives a correct partial hash with a wrong additional character, they could temporarily reject all typoed submissions from the user. Essentially, this would be equivalent to typo correction on the first try, and normal password checking on all subsequent tries.

Proof of Lemma 1. Any authentication message that doesn't follow the correct structure can be discarded. A message is deemed correct if at least one of the hashes is correct, and the corresponding numbers are also correct. A message must either contain a correct hash/number pair, or a correct number and a hash collision. As the hash space is much greater than the space of 16-character passwords, using random hashes to find collisions has a probability of success so low ($< 2^{-128}$) that it is irrelevant. As the checking algorithm prevents timing attacks, finding the hash by itself is not possible. The adversary must then have at least one (hash, number list) pair correct. Every query they make has 18 possibilities of getting an acceptance: one for the first two hashes, and one for each of the 16 (hash, number list) pairs. Each query has 7 chances, hence an upper bound of at most 114 acceptance chances. □

[4]This assumes that the adversary knows the salt, which is reasonable as it could, for example, be computed from the login.

B. Obtaining credentials from the database

The second attack can be performed by an adversary with access to the database and focuses on obtaining correct pairs of password and email/login credentials for use against other targets. The goal is then to prove the following lemma:

Lemma 2. *Let's consider an adversary with access to the usernames, corresponding (password hash, number) lists and transcripts of successful login interactions. Using generic attacks, they require at least $\frac{1}{16}$ as much computing power to get a password of length < 16 from a single user as if the database only stored simple hashes of the passwords without typo-correction.*

Remark 3. *Once again, the bound of $\frac{1}{16}$ corresponds to a worst case analysis. Empirical data shows that the real speedup is close to* 1.5.

1) Securing structural information: The first step to prevent credential theft is to make sure that the database itself doesn't give structural information on the passwords through the way it stores them. For example, storing hash lists of varying lengths would reveal the length of the stored passwords, indicating to adversaries the ones that would be easiest to crack.

For the users with passwords of length < 10, exactly two hashes are stored, and the adversary gains at most a factor 2 in the bruteforcing (less in practice due to non-uniform distribution). Let's now consider users with passwords of lengths ≥ 10.

Deterministically adding extra characters to the end of the password to reach a common length prevents attacks that seek to find the easiest passwords to crack. However, we should avoid compromising users with already long passwords by imposing length upper limits. Adding characters only if the passwords are of length less than 16 seems a good compromise, with only a few passwords standing out from the database as being extra-hard. Despite the uniformity of the database, a successful attack could still happen if an adversary also has access to the messages received by the database. In messages received, the length of the allowed key list — the list of numbers — is also important as it can give a lot of information on the position of the keys on the keyboard. To avoid this, the framework reserves few numbers on the client-side reserved for non-existent keys and fills up the neighbour list with those to prevent this information leak.

2) Cracking the hashes: We are left with the problem of computing passwords from a set of list of hashes and numbers, with each list having a single salt. The adversary has three avenues of attack. The first is by bruteforce: enumerate all the possible passwords and check when they are correct by comparing with the recorded hashes. To prevent this attack, key stretching is central but must be used wisely, to make the computation of each hash expensive and prevent the adversary from bruteforcing billions of passwords per second [33]. The second attack uses hashes directly and computes their preimages. The third attack uses the recorded numbers to get information on specific letters of the password and simplify the rest of the work. We will start by the second and third attacks.

With the second attack, considering each list independently, finding the preimage of a single hash is enough for the attacker, as the number of possibilities left for the missing letter becomes trivial. We are then looking at multi-target preimage attacks with a promise on the structure of the targets (that their preimages are close together[5]). As stated in [34], however, the resistance of even SHA3-256 against generic attacks is much stronger than the security requirements for passwords. This means that the main weakness doesn't come from finding the preimage of the password hashes.

When it comes to the third avenue of attack, collisions are frequent, as opposed to hashes, as the image space of each permutation is small. If computing the permutations were more efficient than computing the hashes, it would be possible for the adversary to eliminate lots of potential passwords quickly. Two methods can be used to prevent this. The first is to run the key stretching method on each random bit computation. The second goes by using the same key stretcher for both the PRNG and the hashing. This can be done by first using the key stretcher on PB_i, hashing the output with different salts to get the random permutations and finally the hash itself. This could slightly affect preimage resistance but makes bruteforce attacks to find the permuted characters at most as efficient as the bruteforce attacks against the hash itself. Indeed, if an adversary wants to eliminate possibilities for the k-th character, they must compute the permuted character for each password, and then eliminate all the impossible ones. If they don't run the procedure for the correct password they can't reliably eliminate passwords or characters, and if they do they automatically get the correct hashes (and the answer) at no additional cost.

3) Bruteforcing the passwords: The main attack left is then to use bruteforce from the password side, testing every password until the adversary finds one with the correct hash. The traditional way to prevent this is to use key stretching methods such as PBKDF2 [35] — or rather Argon2 [36], which also has security guarantees against generic attacks. This is where our frameworks have a security flaw, as we have at least 16 different hashes instead of one to create and send the password, but the adversary only has to find one. Making all of them go through key stretching methods either takes more time or lowers the number of iterations on each of them[6]. Two factors mitigate this flaw: first, even running a key stretching method for a few milliseconds is enough to make bruteforce attacks very costly. Assuming we use Argon2 — which prevents efficient large parallelisation — for 2ms on each hash, cracking a 48-bit password would naively take an average of sixteen billion seconds, or 544 years, on the same machine. This does not use the fact that it is enough to guess one of the hashes containing a typo. Assuming a 5-bit loss of entropy — which requires a well-optimised bruteforcing algorithm — the expected time is still more than 17 years. We simulated the use of this method on the Rockyou leaked password data-set [37], [38], bruteforcing until we obtained hashes for the 50% most frequent passwords of length > 10. The speedup varied depending on which two characters were removed, as shown in Table II, but stayed below 1.5.

Characters removed	none	0-1	1-2	2-3	3-4	4-5	5-6	6-7	7-8	8-9
Unique passwords ($\times 10^6$)	4.40	4.26	4.33	4.29	4.29	4.28	4.26	4.22	4.12	3.96
Proportion for 50%	33.1	29.9	31.4	30.6	30.7	30.4	30.0	29.0	26.7	23.0
Speedup	1	1.11	1.05	1.08	1.08	1.09	1.10	1.14	1.24	1.44

TABLE II
SPEEDUP GAINED FOR DICTIONARY ATTACKS BY REMOVING 2 CHARACTERS FROM ROCKYOU PASSWORDS OF LENGTH > 10. THE FIRST LINE HAS THE NUMBER OF UNIQUE PASSWORDS (IN MILLIONS), AND THE SECOND INDICATES THE PROPORTION OF PASSWORDS NEEDED TO GET THE 50% MOST FREQUENT PASSWORDS IF WE REMOVE THE CHARACTERS IN THE i-TH POSITION.

As we can see, even among a list notorious for containing many bad passwords with lots of redundancy, removing two characters only reduced the average number of hashes to compute by about 31% when setting the character position in advance — and dynamically removing the best 2 characters would improve this by at most a few percentage points. Even with efficient hardware, the attack would be prohibitively costly. Moreover, a smart user interface could compute the key stretching before the user submits the password, recomputing from scratch each time a new character is typed. This can guarantee an additional 10ms of free key stretching per hash without the user noticing. We can now prove Lemma 2. We only consider users with passwords at least 10-characters long, as otherwise the proof is immediate due to the trivial typo correction.

Proof of Lemma 2. The hashes are all computed with different salts, so rainbow tables can't help, and cracking a single user's credentials doesn't help the attacker with the credentials of another user. The data under each user is composed of the same number of hashes and corresponding numbers, except for the users with increased security, and the transcripts are also structurally identical, so finding the users with passwords of

[5]It would be interesting to check whether this kind of promise problem makes preimage computation any easier, but in any case, they could also be made irrelevant by the use of different salts for each of the $(n-1)$ password hashes.

[6]Using a key stretcher on the central salts that are used afterwards by the rest of the algorithm centralises this proof of work but does not provide any extra security.

lower lengths is as easy as finding out that the last characters of those passwords are made of padding. Knowing that they are made of padding requires knowing that they are the image of a non-existent character, which is equivalent to finding that they are the image of a given character.

However, finding whether the number stored corresponds to a given character through bruteforce is not easier than finding the password itself, as a large set of passwords with different characters in its stead will yield the correct number. If the actual password wasn't in the set tested, the adversary can't guess the extracted character with probability much bigger than uniform, whereas if the correct password was in the set the adversary already knows a correct hash.

As the preimages under the hashing functions considered are much harder to compute than bruteforcing from the password side, and as the additional numbers give no information unless the adversary knows the rest of the password, the only viable generic attack goes through bruteforcing from the password side.

An adversary can then consider one position, ignore the two letters concerned, and bruteforce all the others. In a best case scenario, this method could remove close to 14 bits of entropy, or improve by a factor 15000 the speed of the bruteforce. However, using NIST estimates [39], at best 4 bits of entropy would be lost, corresponding to a factor 16 speedup — much higher than the 45% speedup observed on the data.

□

V. Conclusion and Discussion

The main contribution of this paper is a typo-correction system with the following properties:

- It corrects 57.7% of all typos, or 91.2% of acceptable typos.
- It stores 32 hashes and 90 integers on the server. Using lazy evaluation — only checking the remaining hashes when the main one is incorrect — this does not require any extra computation on the server's side.
- It requires no additional waiting time for computation on the user side, as it can run between the moment the user presses the last key and the moment they submit the password.
- It creates little extra communication cost as the additional data can still fit in an average packet (420 bytes for the numbers, 544 bytes for the hashes), well below the IPv6 MTU [40].
- Assuming optimised code that runs on specialised hardware $15\times$ faster than an average client's browser's hashing ability, bruteforcing a single password from the database still takes more than a year[7].
- Faking a correct authentication message is at best 114 times more efficient than normal bruteforce, but this can be compensated or eliminated by having stricter constraints on the number and frequency of queries while still having a positive impact on usability.

When compared to TypTop, the best typo-correction system today[8], it has greater usability — correcting about twice as many typos — and lowered computing requirements. There is, however, a cost, in that our security guarantees are slightly weaker (but not directly comparable as the models are different).

Multiple practical improvements could still be added to the system considered. For example, as the system can detect typos, it might be interesting to let the user know when they've made one (although this might lower usability). Looking in another direction, it would be possible to associate given (hash / number) pairs with frequencies and allow typos probabilistically, with the system being more forgiving when the typo is repeated.

Combining both approaches, if a typo happens with great frequency, it would be possible for the system to ask if the user wants to make that their new password. It would also be possible to use some secret sharing system to combine the different hashes and simplify the computations, but this seems to require a challenge system with at least two rounds of communication.

Naturally the schemes proposed depend on the service providers' will to implement them. Thankfully, we can easily address this. Switching from a system where passwords are simply hashed requires two things to be changed: the database must be transformed, and the client's code must also be made to compute the new kinds of hashes. The first part is relatively simple and can be done by adding an extra column that points to the new complete hashing information and is accessed only when the main hash is not correct. Each time a user correctly logs in, the database uses the occasion to add the relevant data (which is sure to be correct as the main hash matches). This allows the service provider to maintain compatibility with a legacy system and lazily upgrade the security of all users. In the context of long-term maintainability, we focused on Argon2 SHA-3 as primitive functions. That said, other cryptographic hash functions and PRNGs could be used if vulnerabilities were found in the ones mentioned. The main constraint is that the PRNG should be secure on correlated and non-uniform inputs. The parameters on Argon2 also require fine-tuning depending on the assumed client hardware and the estimated abilities of adversaries, as they create a direct trade-off between usability (in login delay) and resistance to credential theft attacks.

The client's code must also be transformed so that it transfers not just the main hash but all the necessary information. This can be done without requiring redeployment or updating clients when considering web services. Indeed, the service provider is also the one providing the Javascript code for the web page, and can update this centrally without directly implicating the users.

[7]This assumes that the client interface runs fast hashing algorithms, for example, in a WebAssembly environment, which can have a $20\times$ speedup over asm.js [41], [42].

[8]This title of best is easily attributed as the only competitors — to our knowledge — are previous systems by the same authors.

An important change is that hashes are computed on the client's side, but there are nowadays next to no reason to compute them on the server's side — unlike two decades ago when they could be necessary to assure compatibility with legacy systems.

References

[1] N. Memon, "How biometric authentication poses new challenges to our security and privacy [in the spotlight]," *IEEE Signal Processing Magazine*, vol. 34, no. 4, pp. 196–194, 2017.

[2] G. C. Batista, C. C. Miers, G. P. Koslovski, M. A. Pillon, N. M. Gonzalez, and M. A. Simplicio, "Using Externals IdPs on OpenStack: A Security Analysis of OpenID Connect, Facebook Connect, and OpenStack Authentication," in *IEEE 32nd International Conference on Advanced Information Networking and Applications – AINA*, vol. 00, 5 2018, pp. 920–927.

[3] B. Jensen, "5 myths of password security," 2013, accessed: 2017-12-18. [Online]. Available: https://web.archive.org/web/20180528052512/https://stormpath.com/blog/5-myths-password-security

[4] W. Ma, J. Campbell, D. Tran, and D. Kleeman, "Password entropy and password quality," in *4th International Conference on Network and System Security*, 9 2010, pp. 583–587.

[5] J. Bonneau, "The science of guessing: Analyzing an anonymized corpus of 70 million passwords," in *IEEE Symposium on Security and Privacy*, 5 2012, pp. 538–552.

[6] R. Shay, S. Komanduri, P. G. Kelley, P. G. Leon, M. L. Mazurek, L. Bauer, N. Christin, and L. F. Cranor, "Encountering stronger password requirements: User attitudes and behaviors," in *Proceedings of the 6th Symposium on Usable Privacy and Security*, ser. SOUPS '10. New York, NY, USA: ACM, 2010, pp. 1–20.

[7] 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 *Proceedings of the 11th symposium on usable privacy and security*, 2015.

[8] P. Lipa, "The security risks of using "forgot my password" to manage passwords," 2016. [Online]. Available: https://web.archive.org/web/20170802185615/https://www.stickypassword.com/blog/the-security-risks-of-using-forgot-my-password-to-manage-passwords/

[9] S. Gressin. (2017) The equifax data breach: What to do. [Online]. Available: https://web.archive.org/web/20190304122541/https://www.consumer.ftc.gov/blog/2017/09/equifax-data-breach-what-do

[10] M. Blum and S. S. Vempala, "Publishable humanly usable secure password creation schemas." in *3rd AAAI Conference on Human Computation and Crowdsourcing*, 2015.

[11] W. Melicher, D. Kurilova, S. M. Segreti, P. Kalvani, R. Shay, B. Ur, L. Bauer, N. Christin, L. F. Cranor, and M. L. Mazurek, "Usability and security of text passwords on mobile devices," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, ser. CHI '16. New York, NY, USA: ACM, 2016, pp. 527–539.

[12] S. M. Segreti, W. Melicher, S. Komanduri, D. Melicher, R. Shay, B. Ur, L. Bauer, N. Christin, L. F. Cranor, and M. L. Mazurek, "Diversify to survive: Making passwords stronger with adaptive policies," in *13th Symposium on Usable Privacy and Security – SOUPS*. Santa Clara, CA: USENIX Association, 2017, pp. 1–12. [Online]. Available: https://www.usenix.org/conference/soups2017/technical-sessions/presentation/segreti

[13] J. Marquardson, "Password policy effects on entropy and recall: Research in progress," in *Americas Conference on Information Systems*, 2012.

[14] W. Yang, N. Li, O. Chowdhury, A. Xiong, and R. W. Proctor, "An empirical study of mnemonic sentence-based password generation strategies," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 1216–1229.

[15] J. Bonneau and E. Shutova, "Linguistic properties of multi-word passphrases," in *International Conference on Financial Cryptography and Data Security*. Springer, 2012, pp. 1–12.

[16] 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.

[17] 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*, ser. CHI '14. New York, NY, USA: ACM, 2014, pp. 2927–2936.

[18] D. R. Pilar, A. Jaeger, C. F. A. Gomes, and L. M. Stein, "Passwords usage and human memory limitations: A survey across age and educational background," *PLoS One*, vol. 7, no. 12, 12 2012, pONE-D-12-21406[PII]. [Online]. Available: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3515440/

[19] Centrify, "Centrify password survey: Summary," Centrify, Tech. Rep., 2014. [Online]. Available: https://www.centrify.com/resources/5778-centrify-password-survey-summary/

[20] P. Lambert. (2012, 6) The case of case-insensitive passwords. [Online]. Available: https://web.archive.org/web/20190310221858/https://www.zdnet.com/article/the-case-of-case-insensitive-passwords/

[21] R. Chatterjee, A. Athayle, D. Akhawe, A. Juels, and T. Ristenpart, "pASSWORD tYPOS and how to correct them securely," in *IEEE Symposium on Security and Privacy*. IEEE, 2016, pp. 799–818.

[22] R. Chatterjee, J. Woodage, Y. Pnueli, A. Chowdhury, and T. Ristenpart, "The typtop system: Personalized typo-tolerant password checking," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 329–346.

[23] N. K. Blanchard, "Password typo correction using discrete logarithms," in *8th International Conference on Computer Science and Communication Engineering*, 2019.

[24] J. Woodage, R. Chatterjee, Y. Dodis, A. Juels, and T. Ristenpart, "A new distribution-sensitive secure sketch and popularity-proportional hashing," in *Advances in Cryptology – CRYPTO*, J. Katz and H. Shacham, Eds. Cham: Springer International Publishing, 2017, pp. 682–710.

[25] S.-T. Sun and K. Beznosov, "The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 378–390.

[26] B. Herzog and Y. Balmas, "Great crypto failures," in *Virus Bulletin Conference*, 2016.

[27] 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.

[28] G. Brassard and S. Kannan, "The generation of random permutations on the fly," *Information Processing Letters*, vol. 28, no. 4, pp. 207–212, Jul. 1988.

[29] P. E. Black, "Fisher-yates shuffle," *Dictionary of algorithms and data structures*, vol. 19, 2005.

[30] Ponemon Institute, "The cost of credential stuffing," Ponemon Institute, Tech. Rep., 2017.

[31] R. Wash, E. Rader, R. Berman, and Z. Wellmer, "Understanding password choices: How frequently entered passwords are re-used across websites," in *12th Symposium on Usable Privacy and Security – SOUPS*. Denver, CO: USENIX Association, 2016, pp. 175–188. [Online]. Available: https://www.usenix.org/conference/soups2016/technical-sessions/presentation/wash

[32] M. Pinola, "Your clever password tricks aren't protecting you from today's hackers," 2014. [Online]. Available: https://web.archive.org/web/20190203093823/https://lifehacker.com/your-clever-password-tricks-arent-protecting-you-from-t-5937303

[33] M. Sprengers, "Gpu-based password cracking," Master's thesis, Radboud University Nijmegen, 2011.

[34] G. Bertoni, J. Daemen, M. Peeters, and G. van Assche, "On the indifferentiability of the sponge construction," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2008, pp. 181–197.

[35] B. Kaliski, "Pkcs# 5: Password-based cryptography specification version 2.0," RFC Editor, Tech. Rep., 2000.

[36] A. Biryukov, D. Dinu, and D. Khovratovich, "Argon2: new generation of memory-hard functions for password hashing and other applications," in *IEEE European Symposium on Security and Privacy – EuroS&P*. IEEE, 2016, pp. 292–302.

[37] A. Vance. (2010) If your password is 123456, just make it hackme. [Online]. Available: https://web.archive.org/web/20181023160454/https://www.nytimes.com/2010/01/21/technology/21password.html

[38] P. G. Kelley, S. Komanduri, M. L. Mazurek, R. Shay, T. Vidas, L. Bauer, N. Christin, L. F. Cranor, and J. Lopez, "Guess again (and again and

again): Measuring password strength by simulating password-cracking algorithms," in *IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 523–537.
[39] W. E. Burr, D. F. Dodson, and T. W. Polk, "Nist special publication 800-63-2," *Electronic Authentication Guideline*, vol. 1, 2004.
[40] S. Deering and R. Hinden. (2014) Rfc 2460-internet protocol, version 6 (ipv6) specification, 1998. [Online]. Available: http://www.ietf.org/rfc/rfc2460.txt
[41] Antelle. (2018) Argon2 in browser. [Online]. Available: https://web.archive.org/web/20180119222301/http://antelle.net/argon2-browser/
[42] A. Rossberg, "Webassembly: high speed at low cost for everyone," in *ML16: Proceedings of the 2016 ACM SIGPLAN Workshop on ML*, 2016.

VI. Appendix: a detailed explanation of the algorithms

To explain how the algorithms work, we'll go through the process of correcting each potential error, starting with the proximity error.

a) Proximity errors.: The framework is composed of the key-setting and key-sending algorithms, which are very similar, and the key-checking algorithm which is built on a different model. The first two are executed on the client's side, and compute a list of hashes, and for each hash a set of numbers. The key-setting algorithm actually sends two hash lists. The first corresponds to the HA_i, hence the hashes of the passwords from which we removed one letter. The second corresponds to the HB_i, the hashes of the passwords from which we removed two adjacent letters. In the key-setting algorithm, the number associated to each HA_i is the image of the missing character through the permutation π_i. The four numbers associated to each HB_i are the images of each two characters with different permutations $\pi_{i,j}$ computed from the same data with different salts.

Let's now suppose that there is a single proximity error on one character in position $i > 1$ (the case $i = 1$ is similar). Then the hash computed by removing this character and the previous one is still correct, and the permutations $\pi_{i,j}$ are also identical to the correct $\pi_{i,j}$. The key-setting algorithm only sends the image of the correct characters through $\pi_{i,j}$, but the key-sending computes a list of neighbours (as well as the original letter with different capitalisation), passes all of them through $\pi_{i,j}$, and sends the resulting list. We know that the previous character is correct, hence $KA_i = LA_i[1]$. If the i-th key pressed is a neighbour of the correct key, then the correct key will be in the neighbour list $LB_i[j]$. The key-checking algorithm will find a correct hash $HB_i = H'_i$, and will verify that $KA_i = LA_i[1]$ and $KB_i = LB_i[j]$ for exactly one j.

There is an alternative approach that could be used to compare the extracted character. Our framework sends the list of neighbours of the typed character. If the typed character is a neighbour of the correct one, then the image of the correct one through the permutation will be among the numbers sent. The other way is to store the list of neighbours during the key-setting and only send the image of the typed character. This has the advantage of slightly lowering the amount of data transferred but makes the system less adaptable to different keyboards. For example, we could consider a user that sets their key on a QWERTY keyboard layout and then uses an AZERTY layout. If instead of typing an "E" in their password they type a 'Z', the password would get rejected, as it is not in the list of neighbours on the initial keyboard. With the version shown in the algorithm, only the present set of neighbours counts[9]. The size of the set of neighbours varies which can lead to vulnerabilities if it isn't addressed, as "1" has fewer neighbours than "g".

[9]This does, however, require the system to know which keyboard layout the user is using, which is not always easy in practice.

b) Transposition errors.: Transposition errors — such as when the string "correction" becomes "correctoin" — are quite easy to correct with what we already have. We only need to check that the images are exchanged, hence that $LC_i[1] = KC_i$ and $LD_i[1] = KD_i$. We use four permutations for a simple reason: the first two permutations are used to identify the neighbours of each character and prevent single proximity errors. The other two are used to check whether the two characters are transposed. Although we could use two permutations, this would allow an adversary to obtain information when the sequential characters are identical, or are each other's neighbours.

c) Insertion errors.: Suppose that the password typed has one extra character. We can then check the stored hashes HA_i, which correspond to the original password with one missing letter. Two such hashes will correspond to hashes computed from the erroneous password with the inserted character removed, as well as the character either before or after it. For one of the two, the corresponding extracted character is then the same as the one stored as K_i.

d) Other errors.: The algorithm features a few other choices. The first is to add an additional hash to account for the inverted case (H_0). The second is to make small passwords indistinguishable in the database, padding[10] them all to length 16. The letter lists can be made of equal length by adding dummy numbers to prevent an adversary from gaining information through the number of neighbours.

[10]This "16" is an arbitrary parameter that is a good compromise to prevent revealing small passwords while not costing too much in storage and time.