Secure and Efficient Password Typo Tolerance

Nikola K. Blanchard

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. We propose an alternative framework which corrects up to 57% of typos without affecting user experience, at no computational cost to the server. We also provide an algorithm for the more general problem of computing an edit distance between two strings without having direct access to those strings, and corresponding impossibility results.

CCS CONCEPTS

• **Security and privacy** → **Authentication**; *Software security engineering*; Hash functions and message authentication codes; • **Human-centered computing** → *User centered design.*

KEYWORDS

Passwords, Hashing functions, Usable security, Discrete logarithm

ACM Reference Format:
Nikola K. Blanchard. 2019. Secure and Efficient Password Typo Tolerance. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Despite recent advances in biometric authentication [39] and account linking [4], 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 [28, 36, 54], initially focusing on low security, with users creating bad passwords [9, 42, 52] and repeatedly dodging security measures [35, 49, 53], but also service providers ignoring best practices on how to secure password databases[23]. More recently, research on how to make them more usable has made advances [8, 38], and some of the effects of bad password policies are being reversed [18, 47], to focus on longer passwords. Unlike random passwords with special characters which suffer from low memorability [37], long and simple passwords and passphrases [10, 30, 34, 57] can benefit from humans' superior ability to memorise strings that make sense, improving both security and usability [13, 41, 48]. 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 [15]. 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 [33, 45].

In an innovative paper in 2016 [16], 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 [17], 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. 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 [56].

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, 51]. 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 multiple simple typo-tolerant frameworks, building up to one complete system that satisfies the different constraints mentioned. It improves usability by correcting up to 57.7% 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.

We also introduce a simple metric called the *keyboard distance*, and a protocol that can compute this distance (as well as the Hamming distance) between a queried string and a secret string, without it being possible to find the secret string in polynomial time (assuming the security of the discrete logarithm).

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Finally, we show some simple lower bounds: for a large set of edit distances, given an oracle to that distance between a secret string and a queried string, it is possible to get the secret string in a polynomial number of queries. This holds even in the case of probabilistic and approximate distances. This means that any general typo-correction system that can compute arbitrary distances is vulnerable in at most a polynomial number of queries. The previous protocol's resistance to this method comes from having queries of non-uniform — and potentially exponential — complexity.

2 TYPO-TOLERANT FRAMEWORKS

The goal of this section is to provide an analysis of the most frequent user errors before introducing three frameworks. Each framework builds on the previous ones, correcting more errors at the cost of increasing storage and computation, to obtain typo-tolerant password checkers that can handle substitutions, transpositions, and finally insertions. A final complete framework then integrates all the features while being efficient, secure and easy to implement.

Here, a 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*). Frameworks should work 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").

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. Around 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.

Throughout this paper, n will be the length of the passwords considered. To make things simpler and more adaptable, the frameworks are built to work using two primitive functions which they call multiple times, HASH and PRNG — for deterministic Pseudo Random Number Generator. The security analyses will focus on Argon2 as HASH and SHA-3 as a PRNG, although 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.

2.1 Typology of errors

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 [16] and made the data publicly accessible. In a follow-up paper [17], they also ran a second study where users were asked to repeatedly enter a password over time (at intervals of at least one hour).

In the original study, the authors chose to only look at strings whose *Damerau-Levenshtein distance* was less than 2 [19], 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 1. Some of the errors considered in [16] would probably not happen in the real world, mostly inserting spaces and transcription errors — such as confusing "1" and "l". Although this creates differences between analyses, one common finding is 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	29.7
QWERTYnumpad neighbour	14.0
Single shift	8.5
Single deletion	19.4
Caps lock	14.7
Single insertion	13.1
Space	2.0
Duplicated letter	3.8
Single transposition	3.9
Other	19.0

Table 1: Types of typos recomputed on the original data-set from [16], over all passwords at distance at most 6 from the correct version, 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 5, 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).

2.2 General intuition

The first framework 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. As with all subsequent algorithms, it relies 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").

All the frameworks are similarities, relying on architectures similar to the following :

- The password of length n is split into n partial passwords, each missing one character.
- The partial passwords are concatenated with a salt[1] before being hashed.
- Pseudorandom permutations within the set of character codes are computed (generally [0; 255], based on the hashes.

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

- Each excluded character and all the adjoining ones on the keyboard are encoded using the corresponding permutation.
- The user sends the authentication message, a list of n pairs of (hash, number list).
- If one hash is correct, and the stored number is in the corresponding list, the server authenticates the user.

2.3 Substitution tolerance

2.3.1 Intuition for the general scheme. The substitution-tolerant key-setting algorithm (Algorithm 2, and Figure 1) works by creating hashes of every substring of length $n-1$ — which we denote as the P_i, where $i \in [1; n]$ is the position of the missing — or, rather, extracted — character. This means that, for any substitution of a single character, one hash — not containing that character — will be correct. We could stop the algorithm here and send the list of hashes, but this would allow specific kinds of targeted attacks examined in Section 3. Sending — or storing — the remaining character in plaintext would give a potential adversary too much information, and even the whole password if this is done for all characters independently. For this reason, the remaining character is also sent, although in an encrypted fashion, with the key depending on the other characters. Here, this encryption is just a pseudorandom permutation, entirely determined by the other characters and computed lazily through Brassard's virtually initialised array algorithm [11].

The key-sending algorithm (Algorithm 2) works in a similar fashion to Algorithm 2. However, instead of sending the hashes of every P_i and the image of the extracted character through the permutation, it sends the image of the extracted character, its shifted version (inverted case) as well as its neighbours on the keyboard. Finally, Algorithm 3 checks that the full hash (H_0) is correct. If it isn't, it checks that at least one partial hash is correct and that the corresponding extracted character is among the ones allowed.

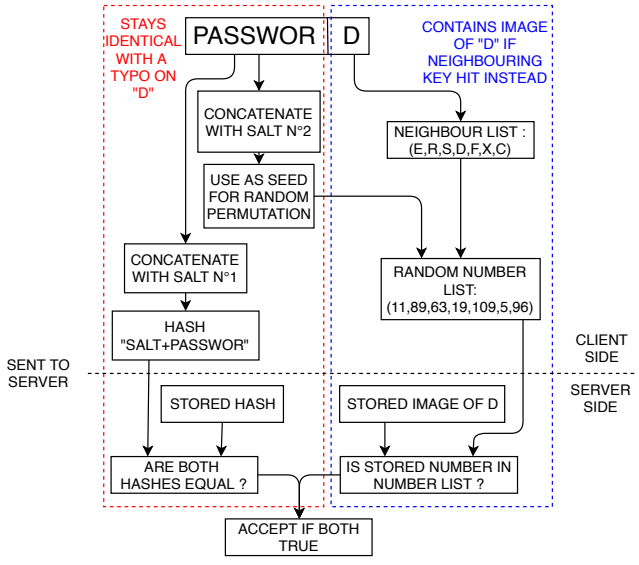

Figure 1: Diagram for the substitution tolerant framework. The real framework removes each letter of the password in parallel, and accepts only if one execution accepts.

Data: Salts S_0, S_1, S_2, Password P of length n
Keyboard map M: Keys $\to [0; 255]$
Hash function HASH
Pseudorandom number generator PRNG
Result: Main hash and list of n (hash / integer) pairs

begin
  $H_0 \longleftarrow$HASH(Concatenate(S_0, P))
  for i from 1 to n **do**
    $P_i \longleftarrow P \setminus P[i]$ /* Removing the i-th letter from the string */
    $H_i \longleftarrow$HASH(Concatenate(S_1, P_i))
    Random_bits $\longleftarrow$ PRNG(Concatenate(S_2, P_i))
    $\pi_i \longleftarrow$ Brassard(Random_bits)
    /* The (pseudo) random bits are used lazily each time the permutation is called for a new number */
    $K_i \longleftarrow \pi_i(M(P[i]))$
  return $(H_0, (H_i, K_i)_{1 \le i \le n})$

Algorithm 1: Key-setting substitution-tolerant algorithm

Data: Salts S_0, S_1, S_2, Password P of length n
Keyboard map M: Keys $\to [0; 255]$
Hash function HASH, Pseudorandom number generator PRNG
Result: Main hash and list of n (hash / integer list) pairs

begin
  $H_0 \longleftarrow$ HASH(Concatenate(S_0, P))
  for i from 1 to n **do**
    $P_i \longleftarrow P \setminus P[i]$
    $H_i \longleftarrow$HASH(Concatenate(S_1, P_i))
    Random_bits$\longleftarrow$ PRNG(Concatenate(S_2, P_i))
    $\pi_i \longleftarrow$ Brassard(Random_bits)
    $L_i \longleftarrow [\pi_i(M(P[i]))]$
    L_i.append($\pi_i(M(\text{SHIFT}(P[i])))$
    foreach $a \in$ Neighbours($P[i]$) **do**
      L_i.append($\pi_i(M(a))$
    L_i.sort()
  return $(H_0, (H_i, L_i)_{1 \le i \le n})$

Algorithm 2: Key-sending substitution-tolerant algorithm

Data: Length n, Original list (H_i, K_i) of (hash/integer) pairs received list (H'_i, L_i) of (hash / integer list) pairs
Result: ACCEPT if and only if the password corresponds to the correct one or a version with an acceptable typo.

begin
  if $H_0 = H'_0$ **then**
    return ACCEPT
  else
    for i from 1 to n **do**
      if $H_i = H'_i$ **then**
        for j from 1 to $|L_i|$ **do**
          if $L_i[j] = K_i$ **then**
            return ACCEPT
  return REJECT

Algorithm 3: Key-checking substitution-tolerant algorithm

2.3.2 Design choices and security. Two approaches could be used to compare the extracted character. The first, shown here, 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[2]. 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". A solution to this is proposed in the complete algorithm.

Instead of a permutation, a function that goes 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 probability of success 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 already low probability of getting a correct number in the list ($\leq \frac{7}{255}$).

Our algorithms 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, through the Fisher-Yates algorithm [7] for example). 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[3], which could be attainable using a longer salt for the seed (hundreds of bits) and a PRNG with variable output. Using Brassard's algorithm [11], 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. In all cases, the algorithm used to get the random bits should not be too efficient, as seen in Section 3.

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.

2.4 Transposition tolerance

2.4.1 Problem and intuition. On top of single substitution errors, the second framework also corrects transposition errors, where the string "correction" becomes "correctoin". In the first framework, a single letter was extracted before hashing, but this method couldn't detect or correct transpositions. This second framework works similarly but extracts two adjacent letters each time and encodes the extracted characters through four different permutations. 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. Instead of sending a hash and a list of neighbours for each character, the key-sending algorithm sends a hash and a set of lists of neighbours for each pair of adjacent characters.

2.4.2 Inner workings. In the transposition-tolerant framework[4], two lists, LA_i and LB_i, represent the sets of neighbours of the first and second extracted characters for $0 \leq i \leq n-1$. The checking algorithm accepts if three conditions are true for one index: the hash is correct, one of the two extracted characters is in the set of allowed characters and, finally, the second extracted character is correct (which is true if it has the first index in LA_i or LB_i).

2.4.3 Design choices. In this framework, the use of different permutations to encode the two excluded characters is essential, as otherwise the security would be inadvisably lowered. The first two permutations have to be different to prevent an adversary from finding out whether the two characters are neighbours, or have common neighbours. The other two have to be different to prevent adversaries from finding whether a character is present twice in a row.

2.4.4 Optimisation. 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 commucation costs by at most 19%.

2.5 Insertion tolerance

2.5.1 Problem and intuition. Insertions can be handled by combining the previous two frameworks: the key-setting algorithm sends information corresponding to both previous frameworks, and the key-sending algorithm is the same as Algorithm 2. Using both kinds of hashes, we can detect and evaluate insertions as well as proximity substitutions and transpositions.

Suppose that the password typed has one extra character. We can then check the hashes created by Algorithm 2, as two of them will correspond to hashes computed for a single removed character. For one of the two, the corresponding extracted character is then the same as the one received.

2.5.2 Security. The question of which insertions should be allowed is non-trivial. For example, duplicated letters or added spaces seem like good candidates, whereas letters far from the 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. All those can be corrected by this framework, depending on what is chosen to be the list of neighbours. To address all the comments up to here, we then show the complete framework.

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

[3]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.

[4]The pseudocode for this framework and the following two is shown in the appendix.

2.6 Complete framework

The previous framework can easily be made into a complete framework that is ready for implementation by tweaking a few things. The first is to add an additional hash to account for the inverted case. The second is to make small passwords indistinguishable in the database, padding[5] 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. Finally, we also replace the arbitrary hash function and PRNG by SHA3-256 and use Argon2 as a key stretcher. The parameters on Argon2 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.

2.7 Performance summary

The frameworks shown differ in terms of the proportion of typos handled, computational costs, communication requirements and storage space needed. Those can also vary with the paremeters of each framework's implementation, such as the list of allowed typos. Table 2 sums up the performance on each front. The difference between the conservative and tolerant methods corresponds to the strategy on accepting random insertions, the first one accepting only duplicated letters or spaces, and the second accepting all insertions. To note, the additional 15.5% of corrected typos from handling caps lock in the complete framework could also be added to any of the other frameworks. The final algorithm only handles 55.7% of passwords with typo, a proportion mostly bounded by our decision to ignore any typo that includes a deletion (at least 22.7% of mistyped passwords). If we consider all the typos we do not want to accept as they do not seem legitimate or pose a security risk — deletions, substitutions which are not with adjacent characters, or more than 2 typos — the forbidden typos represent 36.7% of all mistyped passwords. The complete framework can then handle up to 91.2% of acceptable typos.

Algorithm	Substitution	Transposition	Insertion	Complete
Computation				
Permutations #	n	$4n-4$	$4n-4$	$\max(4(n-1), 60)$
Hashes #	$n+1$	n	n	$\max(n+1, 17)$
Numbers #	$n \times k$	$(n-1) \times 4k$	$(n-1) \times 4k$	$\max(4(n-1)k, 60k)$
Storage				
Hashes #	$n+1$	n	$2n$	$\max(2n+1, 33)$
Numbers #	n	$4n$	$5n$	$\max(5n, 80)$
Typos handled				
Conservative	24.2 %	28.4 %	34.5 %	50.2 %
Tolerant	24.2 %	28.4 %	42.2 %	57.7 %

Table 2: Performance comparison of the four frameworks, showing the computation, communication and storage requirements of each, as well as the proportion of total typos handled, depending on the strategy used.

3 SECURITY ANALYSIS

Our frameworks seek to improve authentication systems, which 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 [44].

3.1 Preventing access

As the frameworks considered seek to tolerate certain typos, they inevitably cause an 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 [42, 55]. 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[6]. 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 is exactly that of the added typos. In other words, those frameworks 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. The proof is at the end of subsection 3.2.

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 $\frac{1}{114}$ sounds 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 as these passwords are generally not vulnerable to the attacks considered.

3.1.1 Intuition. There are two ways an adversary could obtain access if they have no prior information besides the username.

[5] 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.

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

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 in the complete framework 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[7]. 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. 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 iteslf 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 uper bound of at most 114 acceptance chances. □

3.2 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.

3.2.1 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 this kind of attack. 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, having a few numbers on the client-side reserved for non-existent keys and filling up the neighbour list with those prevents this information leak.

3.2.2 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[8] 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 [50]. 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.

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

[8] Essentially, key stretching is the process of running the hash function on itself a fixed number of times to make computation slower — and bruteforce more expensive.

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[9]). As stated in [5], 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.

3.2.3 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 [29] — or rather Argon2 [6], 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[10]. 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 [31, 54], 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 3, but stayed below 1.5.

[9]It would be interesting to check whether tist 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.

[10]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.

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 3: 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. 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 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 [14], 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.

□

4 ALTERNATIVE ALGORITHM BASED ON THE DISCRETE LOGARITHM AND LOWER BOUNDS

Although the algorithms shown previously should be sufficient for practical applications, it is worth wondering whether better solutions could exist, at least from a theoretical standpoint. Taking inspiration from homomorphic encryption, can we safely compute some forms of edit distances between two strings on encrypted data? The techniques involved here are related to fuzzy extractors [21], which have been used to tolerate errors in biometric authentication systems without revealing information [12].

4.1 Impossibility results and lower bounds

The following trivial lemma poses a first obstacle to the existence of such frameworks:

Lemma 3 (Folklore). *Let A be an n-character string on an alphabet of size m. Knowing n and having access to an oracle that can compute the Hamming distance between A and B for any n-character string B, one can find A in $O(m \times n)$ queries to the oracle.*

Proof. By taking any B as an initial string and iteratively changing one character repeatedly until the oracle indicates a lower distance before moving to the next character, we can converge to the correct A in at most $m \times n$ queries. □

Remark 4. The bound of $O(m \times n)$ can be reduced to $O((m+n)\log m)$ by cutting the string into bits of size $< m$, finding which letters are present in each substring and getting their positions by dichotomy.

Lemma 3 is but a first obstacle, as this reasoning applies to other metrics, shown in the following lemma and its corollaries. Let $\mathcal{F}$ be a set of operations that take a single string, a set of indices in that string, and a set of characters in an alphabet of size m and return a string. Let $\mathcal{F}$ be such that for any operation that can transform A into B, there is a symmetric[11] operation transforming B into A. Let the $\mathcal{F}$-edit distance d between two strings be defined as the minimal number of operations from $\mathcal{F}$ required to transform one into the other.

Lemma 4. *Let A be an n-character string on an alphabet of size m. Knowing n and having access to an oracle that can compute the $\mathcal{F}$-edit distance between A and B for any n-character string B, one can find A in $O(D \times (\max(m,n)^k))$ queries to the oracle, where k is the maximum arity of operations in $\mathcal{F}$, and D is the maximum $\mathcal{F}$-edit distance between two strings of length n.*

Proof. As before, we start with an arbitrary string B and query its distance to the target. We then run each possible operation in $\mathcal{F}$ on all possible operands, querying the oracle with each result. As there are at most $(\max(m,n)^k)$ operand combinations, and at least one of them reduces $d(A,B)$, we need at most $O(D \times (\max(m,n)^k))$ queries to find A. □

Corollary 1. *Finding A without n takes at most $O(m + n\log n)$ queries with an oracle for the Levenshtein distance, $O(n(\max(m,n)))$ for Damerau-Levenshtein [19], and $O(n^4)$ for Kendall's tau [32].*

[11] Through the definition, we have the properties of non-negativity, identity of indiscernibles and subadditivity, thus we only need to add symmetry to get a metric.

Proof. The second and third assertions are obtained by direct application of the previous lemma, but we need details for the first.

We can start by obtaining n in $O(n)$ queries (for any of the considered distance and including when only queries of at least a minimal size are accepted). Starting with the smallest query size, we increase the query at each step while keeping all the letters equal to any constant letter. The distance stays constant or decreases before eventually increasing — corresponding to a deletion — at which point n is equal to the current length minus one.

Once we have n, we can find how many occurrences of each letter we have by querying words composed of a single repeated letter, which takes $O(m)$ queries. In $O(n)$ queries we can get the exact location of all occurrences of one given letter (by putting it successively in each position). Using this letter as a background, we can do a dichotomic search for the positions of each remaining letter, which takes $O(\log n)$ queries per letter, or $O(n\log n)$ total. □

Corollary 1 also holds when the answer given by the oracle is imprecise. For example, instead of computing the exact distance, it could indicate $f(d(A,B))$, for a function f like $f(x) = \lceil \frac{x}{4} \rceil$.

Corollary 2. *Let f be a non-decreasing function, and let*

$$D = \max_{B,\,|B|\leq|A|} d(A,B) \text{ and } p = \max_{i,j<D,\,f(i)=f(j)}(i-j)$$

If the oracle answers queries by returning $f(d(A,B))$, finding A takes at most $O(D \times (\max(m,n)^k)^p)$ queries.

Proof. Following the proof of Lemma 4, we can try all sequences of at most i $\mathcal{F}$-operations, for $1 \leq i \leq p$. This requires at most $(\max(m,n)^k)^p)$ each time, and has to be done at most D times. □

Corollary 3. *For a given A and B, let the oracle have a probabilistic outcome which follows a distribution with values in $[0,T]$ and with expectation $d(A,B)$. There is a probabilistic algorithm that computes A with probability $\Omega(\frac{1}{2})$ with a number of queries at most $O(D \times (\max(m,n)^k) \times k \times \ln(D \times \max(m,n)) \times 2T^2)$.*

Proof. The proof follows that of Lemma 4, except that instead of querying for a single value from the oracle, we make $2kT^2 \times ln(4D \times \max(m,n))$ queries and take the integer closest to the mean. By Hoeffding's inequality [26], we have a probability at most $\frac{1}{2D\times(\max(m,n)^k)}$ of getting a wrong value for the distance. Using the union bound [27], the probability of getting at least one distance wrong is at most $\frac{1}{2}$. Thus, with probability $\Omega(\frac{1}{2})$, all the distances are accurate and the correct string is computed. □

The proofs of those corollaries can also be combined to prove that the lemma holds even against an oracle that answers a probabilistic approximation of a function of the distance. We can then see that any simple system that allows computation of an edit distance between a secret string and arbitrary queried strings can be used to find the original secret string in a polynomial number of queries. Any method that seeks to prevent the discovery of the secret string must then be able to overcome this, for example — as is done hereafter — by having variable costs for queries. Proposition 1 in the Appendix also shows the space-optimality of the last framework we will now introduce.

4.2 Discrete logarithm method

Before the algorithm, we must first introduce a distance between strings which, although simple, is not generally used. Let's consider a keyboard, with a standard QWERTY layout. The 48 main keys of the keyboard and the different characters they can create can easily be modelled by a 3-dimensional coordinate system. The first dimension corresponds to the horizontal position of the key (or the row), the second dimension to the vertical (the diagonal column), and the third dimension to the *modifiers*, here only considering Shift although it could easily be extended. This forms a subset of a $14 \times 4 \times 2$ lattice[12].

Definition 1. Let s be a string of length n. The string coordinates of s are defined as the sequences $(x_i)_{1\leq i\leq n}$,$(y_i)_{1\leq i\leq n}$ and $(z_i)_{1\leq i\leq n}$, where (x_i, y_i, z_i) are the coordinates of the i-th letter in the previous coordinate system.

Definition 2. Let s and s' be strings of identical length n. Let the keyboard distance between s and s' be defined as the L^1-distance between their string coordinates, that is,

$$d(s, s') = \sum_{1\leq i\leq n} \left(\left|x_i - x'_i\right| + \left|z_i - z'_i\right| + \left|z_i - z'_i\right|\right)$$

By this definition, the distance between *homomorphic* and *homimorphic* is 1, but the distance between *homomorphic* and *Bomomorphic* is 3, the same as the distance between *homomorphic* and *homomor;jkc*. The expected distance between two random n-character strings is then $\frac{59707}{10296} \times n$, or about 58 for 10-character keymashes.

Definition 3. Let s be a string of length n, and let $(x_i)_{1\leq i\leq n}$,$(y_i)_{1\leq i\leq n}$ and $(z_i)_{1\leq i\leq n}$ be its string coordinates. Let p_i be the i-th prime number. We define the integral representation $X(s)$ of s as

$$X(s) = \prod_{1\leq i\leq n} p_i^{x_i} \times p_{i+n}^{y_i} \times p_{i+2n}^{z_i}$$

Remark 5. Alternatively, we could have used a more intuitive definition, with $X(s) = \prod_{1\leq i\leq n} p_{3i-2}^{x_i} \times p_{3i-1}^{y_i} \times p_{3i}^{z_i}$. This means that strings that include others as prefixes have integral representations that are multiples of the prefixes' integral representations. As we only consider strings of constant length, this leads to higher values of $X(s)$ with no real advantage. On a standard keyboard, for a string s of length 10, $X(s) < 2^{966}$ with the second definition whereas $X(s) < 2^{768}$ with the first (and $X(s) < 2^{853}$ for length 12). In all cases, they are in expectation quite above 2^{250}, which is enough to prevent discrete logarithm attacks on small exponents [24].

With the coordinate system, the associated distance and an integral representation, we can now define the key-setting and the typo-checking algorithms, inspired by the Diffie-Helman key exchange. Intuitively, we take a random element in a group and put it to the X-th power, where X is dependent on the password. Because of the function's structure, it is easy to compare the elements corresponding to two closely related strings. The security lies in the assumed hardness of computing the discrete logarithm.

Here, the key-setting and key-sending algorithm are identical: the stored key is exactly the one that is sent the first time the client creates the password.

[12]One could also add the space key, in which case the following proofs still work although with a slightly different structure. Similarly, adding the Alt key would only make it a 4-dimensional coordinate system.

Remark 6. As before, the PRNG in the algorithm could be a derivative of Keccak but we do not not require a high level of security here. Any algorithm to get an element from a set of pseudorandom bits — such as a PCG one [40] — would be appropriate.

Data: Username string U, Salt string S, Password string P
Group G, Pseudorandom number generator f
Result: An element $g_0 \in G$ corresponding to the "hashed" password, that is sent to the server

1 **begin**
2 Compute the string coordinates $(x_i, y_i, z_i)_{1\leq i\leq |P|}$ of P
3 $X \longleftarrow \prod_{1\leq i\leq n} p_i^{x_i} \times p_{i+n}^{y_i} \times p_{i+2n}^{z_i}$
4 $N \longleftarrow f(U + S)$
5 Let g be a pseudorandom element g of G computed from N
6 Transfer g^X to the server

Algorithm 4: Key-setting/sending discrete logarithm algorithm

Data: Group G, constant D, maximum length n
Stored element $g_0 \in G$, received element $g_1 \in G$
Result: Keyboard distance between the passwords if it's less than D.

1 **begin**
2 **for** i from 1 to D **do**
3 **for** j from 0 to i **do**
4 $L_0 \longleftarrow [\,], L_1 \longleftarrow [\,]$
5 **foreach** $1 \leq a_1 \leq a_2 \leq \ldots \leq a_j \leq 3n$ **do**
6 $X_0 \longleftarrow \prod_{a_k} p_{a_k}$
7 $L_0 \longleftarrow$ Concatenate$(L_0, {g_0}^{X_0})$
8 **foreach** $1 \leq b_1 \leq b_2 \leq \ldots \leq b_{i-j} \leq 3n$ **do**
9 $X_1 \longleftarrow \prod_{b_k} p_{b_k}$
10 $L_1 \longleftarrow$ Concatenate$(L_1, {g_1}^{X_1})$
11 **foreach** $g' \in L_0$ **do**
12 **if** $g' \in L_1$ **then return** i
13 **return** REJECT

Algorithm 5: Distance-checking discrete logarithm algorithm

Remark 7. The reason why we compute two lists of elements is that computing errors where a_i is greater than expected is easy, as $g^{Xp_i} = (g^X)^{p_i}$. Computing errors the other way around is actually akin to computing a discrete logarithm in the group. As such, the distance computation in this algorithm always goes from the "smaller" to the "bigger" password, which can thankfully be mixed when the keyboard distance is greater than 1.

The comparison on lines 14-16 is not optimal, but it is simple and should not affect the running time as the main factor is the computation of the powers of the elements in G.

4.3 Security and performance

The security of this algorithm directly comes from the discrete logarithm assumption: computing P from g_0 corresponds exactly to solving the discrete logarithm with the promise that the solution is a $3n$-smooth number — for potentially high n in case of added padding. To implement it in practice, one would have to be careful

to choose an appropriate group [1]. A cyclic group of order p with p a 2048-bit prime should be sufficient against current computational capabilities. A similar algorithm could be adapted to elliptic curves.

With this framework, the login queries are all of the same format: a single element of the group. In a way, as shown in Proposition 1 in the Appendix, this could lead to a proof of optimality in terms of space and communication bits required, depending on the group used in practice. It also means that faking an id is is not easier than the hardest typo-tolerant framework that accepts the same typos. As the size of the group is much greater than the general password space, the discrete logarithm assumption also implies that bruteforcing the password is once again the best avenue of attack. As previously, this can be slightly optimised by not trying passwords that are close to each other.

Besides the fact that it only allows the correction of substitution errors, the main downside of this algorithm is the time needed to compute the distance. This is still acceptable on the client side, where the main hurdle is squaring an element at most 1600 times in a large group. Using efficient libraries, this can be done in less than 10ms. However, the server-side computation is where the cost becomes prohibitive. For strings of length 12, checking whether they are at distance 1 takes at most 72 exponentiation operations, or less than 500 squaring operations, doable in a few ms. At distance 2, computation already takes 35 times more operations, which is on the edge of noticeable from the client-side. Checking whether they are at distance 3 (probably the highest reasonable distance for typos) is, alas, prohibitive, taking at least a few seconds. Using the trinomial revision, the number of expected exponentiations at distance $D \leq n$ is on average

$$\frac{1}{2}\sum_{i=0}^{D}\left(\binom{3n}{i}\binom{3n-i}{D-i}\right) = 2^{D-1} \times \sum_{i=0}^{D}\binom{3n}{D} \geq \frac{1}{2}\left(\frac{6n}{D}\right)^{D}$$

This illustrates why the method is not concerned by the lower bounds shown previously: although a linear number of queries might be enough to find the original string from the computed distances, most of those couldn't be computed because of the exponential cost.

Remark 8. There is, in fact, one potential risk that requires investigating with this method. The discrete logarithm assumption concerns normal elements of the group. However, the elements considered here are not random elements but X-th powers, with B-smooth X, for $101 \leq B \leq 181$. Although B-smooth numbers are essential in discrete logarithm problems [43], this does not seem to be a situation where X being B-smooth is an issue.

5 CONCLUSION

The main contribution of this paper is a set of frameworks that can be combined in a complete system with the following properties:

- It corrects 57.7% of all typos, or 91.2% of legitimate 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 [20].
- Assuming optimised code that runs on specialised hardware 15× faster than an average client's browser's hashing ability, bruteforcing a single password from the database still takes more than a year[13].
- 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[14], it has greater usability — correcting about twice as many typos — and lowered computing requirements, at the cost of an increased storage and slightly lowered security guarantees.

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.

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.

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.

[13] This assumes that the client interface runs fast hashing algorithms, for example, in a WebAssembly or PNaCl environment, which can have a 20× speedup over asm.js [2, 22, 46].

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

REFERENCES

[1] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. 2015. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 5–17. https://doi.org/10.1145/2810103.2813707

[2] Antelle. 2018. Argon2 in browser. https://web.archive.org/web/20180119222301/http://antelle.net/argon2-browser/

[3] Axel Bacher, Olivier Bodini, Hsien-Kuei Hwang, and Tsung-Hsi Tsai. 2017. Generating Random Permutations by Coin Tossing: Classical Algorithms, New Analysis, and Modern Implementation. *ACM Transactions on Algorithms – TALG* 13, 2 (2017), 24.

[4] G. C. Batista, C. C. Miers, G. P. Koslovski, M. A. Pillon, N. M. Gonzalez, and M. A. Simplicio. 2018. 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. 920–927. https://doi.org/10.1109/AINA.2018.00135

[5] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. 2008. On the indifferentiability of the sponge construction. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 181–197.

[6] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. 2016. Argon2: new generation of memory-hard functions for password hashing and other applications. In *IEEE European Symposium on Security and Privacy – EuroS&P*. IEEE, 292–302.

[7] Paul E. Black. 2005. Fisher-yates shuffle. *Dictionary of algorithms and data structures* 19 (2005).

[8] Manuel Blum and Santosh Srinivas Vempala. 2015. Publishable humanly usable secure password creation schemas.. In *3rd AAAI Conference on Human Computation and Crowdsourcing*.

[9] J. Bonneau. 2012. The Science of Guessing: Analyzing an Anonymized Corpus of 70 Million Passwords. In *IEEE Symposium on Security and Privacy*. 538–552. https://doi.org/10.1109/SP.2012.49

[10] Joseph Bonneau and Ekaterina Shutova. 2012. Linguistic properties of multi-word passphrases. In *International Conference on Financial Cryptography and Data Security*. Springer, 1–12.

[11] Gilles Brassard and Sampath Kannan. 1988. The Generation of Random Permutations on the Fly. *Inform. Process. Lett.* 28, 4 (July 1988), 207–212. https://doi.org/10.1016/0020-0190(88)90210-4

[12] J. Bringer, H. Chabanne, and Q. Tang. 2007. An Application of the Naccache-Stern Knapsack Cryptosystem to Biometric Authentication. In *2007 IEEE Workshop on Automatic Identification Advanced Technologies*. 180–185. https://doi.org/10.1109/AUTOID.2007.380616

[13] Sacha Brostoff and M. Angela Sasse. 2000. Are Passfaces More Usable Than Passwords? A Field Trial Investigation. In *People and Computers XIV — Usability or Else!: Proceedings of HCI*, Sharon McDonald, Yvonne Waern, and Gilbert Cockton (Eds.). Springer London, London, 405–424. https://doi.org/10.1007/978-1-4471-0515-2_27

[14] William E. Burr, Donna F. Dodson, and Timothy W. Polk. 2004. Nist special publication 800-63-2. *Electronic Authentication Guideline* 1 (2004).

[15] Centrify. 2014. *Centrify Password Survey: Summary*. Technical Report. Centrify. https://www.centrify.com/resources/5778-centrify-password-survey-summary/ Accessed: 2017-12-20.

[16] Rahul Chatterjee, Anish Athayle, Devdatta Akhawe, Ari Juels, and Thomas Ristenpart. 2016. pASSWORD tYPOS and how to correct them securely. In *IEEE Symposium on Security and Privacy*. IEEE, 799–818.

[17] Rahul Chatterjee, Joanne Woodage, Yuval Pnueli, Anusha Chowdhury, and Thomas Ristenpart. 2017. The TypTop System: Personalized Typo-Tolerant Password Checking. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 329–346. https://doi.org/10.1145/3133956.3134000

[18] Lorrie Faith Cranor. 2016. Time to rethink mandatory password changes. https://web.archive.org/web/20190302102425/https://www.ftc.gov/news-events/blogs/techftc/2016/03/time-rethink-mandatory-password-changes

[19] Fred J. Damerau. 1964. A technique for computer detection and correction of spelling errors. *Commun. ACM* 7, 3 (1964), 171–176.

[20] S. Deering and R. Hinden. 2014. RFC 2460-Internet Protocol, Version 6 (IPv6) Specification, 1998. http://www.ietf.org/rfc/rfc2460.txt

[21] Y. Dodis, R. Ostrovsky, L. Reyzin, and A. Smith. 2008. Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data. *SIAM J. Comput.* 38, 1 (2008), 97–139. https://doi.org/10.1137/060651380 arXiv:https://doi.org/10.1137/060651380

[22] Alan Donovan, Robert Muth, Brad Chen, and David Sehr. 2010. Pnacl: Portable native client executables. *Google White Paper* (2010).

[23] Seena Gressin. 2017. The Equifax Data Breach: What to Do. https://web.archive.org/web/20190304122541/https://www.consumer.ftc.gov/blog/2017/09/equifax-data-breach-what-do

[24] Aurore Guillevic and François Morain. 2016. Discrete Logarithms. In *Guide to pairing-based cryptography*, Nadia El Mrabet and Marc Joye (Eds.). CRC Press - Taylor and Francis Group, 42. https://hal.inria.fr/hal-01420485

[25] Ben Herzog and Yaniv Balmas. 2016. Great Crypto Failures. In *Virus Bulletin Conference* (2016-10-01).

[26] Wassily Hoeffding. 1963. Probability Inequalities for Sums of Bounded Random Variables. *J. Amer. Statist. Assoc.* 58, 301 (3 1963), 13–30. http://www.jstor.org/stable/2282952?

[27] David Hunter. 1976. An upper bound for the probability of a union. *Journal of Applied Probability* 13, 3 (1976), 597–603. https://doi.org/10.2307/3212481

[28] Brent Jensen. 2013. 5 Myths of Password Security. https://web.archive.org/web/20180528052512/https://stormpath.com/blog/5-myths-password-security Accessed: 2017-12-18.

[29] B. Kaliski. 2000. *PKCS# 5: Password-Based Cryptography Specification Version 2.0*. Technical Report. RFC Editor.

[30] Mark Keith, Benjamin Shao, and Paul John Steinbart. 2007. The usability of passphrases for authentication: An empirical field study. *International journal of human-computer studies* 65, 1 (2007), 17–28.

[31] Patrick Gage Kelley, Saranga Komanduri, Michelle L. Mazurek, Richard Shay, Timothy Vidas, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, and Julio Lopez. 2012. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In *IEEE Symposium on Security and Privacy*. IEEE, 523–537.

[32] Maurice G. Kendall. 1938. A new measure of rank correlation. *Biometrika* 30, 1/2 (1938), 81–93.

[33] Patrick Lambert. 2012. The case of case-insensitive passwords. https://web.archive.org/web/20190310221858/https://www.zdnet.com/article/the-case-of-case-insensitive-passwords/

[34] Micah Lee. 2015. Passphrases that you can memorize – but that even the NSA can't guess. https://web.archive.org/web/20180115133823/https://theintercept.com/2015/03/26/passphrases-can-memorize-attackers-cant-guess/

[35] Peter Lipa. 2016. The Security Risks of Using "Forgot My Password" to Manage Passwords. https://web.archive.org/web/20170802185615/https://www.stickypassword.com/blog/the-security-risks-of-using-forgot-my-password-to-manage-passwords/ Accessed: 2017-12-18.

[36] W. Ma, J. Campbell, D. Tran, and D. Kleeman. 2010. Password Entropy and Password Quality. In *4th International Conference on Network and System Security*. 583–587. https://doi.org/10.1109/NSS.2010.18

[37] Jim Marquardson. 2012. Password Policy Effects on Entropy and Recall: Research in Progress. In *Americas Conference on Information Systems*.

[38] William Melicher, Darya Kurilova, Sean M. Segreti, Pranshu Kalvani, Richard Shay, Blase Ur, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, and Michelle L. Mazurek. 2016. Usability and Security of Text Passwords on Mobile Devices. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, New York, NY, USA, 527–539. https://doi.org/10.1145/2858036.2858384

[39] Nasir Memon. 2017. How biometric authentication poses new challenges to our security and privacy [in the spotlight]. *IEEE Signal Processing Magazine* 34, 4 (2017), 196–194.

[40] Melissa E. O'Neill. 2014. PCG: A family of simple fast space-efficient statistically good algorithms for random number generation. *ACM Trans. Math. Software* (2014).

[41] Denise Ranghetti Pilar, Antonio Jaeger, Carlos F. A. Gomes, and Lilian Milnitsky Stein. 2012. Passwords Usage and Human Memory Limitations: A Survey across Age and Educational Background. *PLoS One* 7, 12 (05 12 2012). https://doi.org/10.1371/journal.pone.0051067 PONE-D-12-21406[PII].

[42] Melanie Pinola. 2014. Your Clever Password Tricks Aren't Protecting You from Today's Hackers. https://web.archive.org/web/20190203093823/https://lifehacker.com/your-clever-password-tricks-arent-protecting-you-from-t-5937303

[43] Carl Pomerance. 1994. The role of smooth numbers in number theoretic algorithms. In *International Congress of Mathematicians*. Citeseer.

[44] Ponemon Institute. 2017. *The Cost of Credential Stuffing*. Technical Report. Ponemon Institute.

[45] Emil Protalinski. 2011. Facebook passwords are not case sensitive. https://web.archive.org/web/20180422141217/https://www.zdnet.com/article/facebook-passwords-are-not-case-sensitive-update/

[46] Andreas Rossberg. 2016. WebAssembly: high speed at low cost for everyone. In *ML16: Proceedings of the 2016 ACM SIGPLAN Workshop on ML*.

[47] Sean M. Segreti, William Melicher, Saranga Komanduri, Darya Melicher, Richard Shay, Blase Ur, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, and Michelle L. Mazurek. 2017. Diversify to Survive: Making Passwords Stronger with Adaptive Policies. In *13th Symposium on Usable Privacy and Security – SOUPS*. USENIX Association, Santa Clara, CA, 1–12. https://www.usenix.org/conference/soups2017/technical-sessions/presentation/segreti

[48] Richard Shay, Saranga Komanduri, Adam L. Durity, Phillip (Seyoung) Huh, Michelle L. Mazurek, Sean M. Segreti, Blase Ur, Lujo Bauer, Nicolas Christin, and

Lorrie Faith Cranor. 2014. Can Long Passwords Be Secure and Usable?. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 2927–2936. https://doi.org/10.1145/2556288.2557377

[49] Richard Shay, Saranga Komanduri, Patrick Gage Kelley, Pedro Giovanni Leon, Michelle L. Mazurek, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. 2010. Encountering Stronger Password Requirements: User Attitudes and Behaviors. In *Proceedings of the 6th Symposium on Usable Privacy and Security (SOUPS '10)*. ACM, New York, NY, USA, Article 2, 20 pages. https://doi.org/10.1145/1837110.1837113

[50] Martijn Sprengers. 2011. *GPU-based Password Cracking*. Master's thesis. Radboud University Nijmegen.

[51] San-Tsai Sun and Konstantin Beznosov. 2012. 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 (CCS '12)*. ACM, New York, NY, USA, 378–390. https://doi.org/10.1145/2382196.2382238

[52] Aaron Toponce. 2011. Strong Passwords NEED Entropy. https://web.archive.org/web/20180223215746/https://pthree.org/2011/03/07/strong-passwords-need-entropy/ Accessed: 2017-12-18.

[53] Blase Ur, Fumiko Noma, Jonathan Bees, Sean M. Segreti, Richard Shay, Lujo Bauer, Nicolas Christin, and Lorrie F. Cranor. 2015. 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*.

[54] Ashlee Vance. 2010. If your password is 123456, just make it hackme. https://web.archive.org/web/20181023160454/https://www.nytimes.com/2010/01/21/technology/21password.html

[55] Rick Wash, Emilee Rader, Ruthie Berman, and Zac Wellmer. 2016. Understanding Password Choices: How Frequently Entered Passwords Are Re-used across Websites. In *12th Symposium on Usable Privacy and Security – SOUPS*. USENIX Association, Denver, CO, 175–188. https://www.usenix.org/conference/soups2016/technical-sessions/presentation/wash

[56] Joanne Woodage, Rahul Chatterjee, Yevgeniy Dodis, Ari Juels, and Thomas Ristenpart. 2017. A New Distribution-Sensitive Secure Sketch and Popularity-Proportional Hashing. In *Advances in Cryptology – CRYPTO*, Jonathan Katz and Hovav Shacham (Eds.). Springer International Publishing, Cham, 682–710.

[57] Weining Yang, Ninghui Li, Omar Chowdhury, Aiping Xiong, and Robert W. Proctor. 2016. An Empirical Study of Mnemonic Sentence-based Password Generation Strategies. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 1216–1229. https://doi.org/10.1145/2976749.2978346

6 APPENDIX

We start by showing the pseudocode for the previous frameworks before proving a lower bound for the last framework.

6.1 Transposition tolerance

Data: Salts S_0, S_1, $\ldots S_5$, Password P of length n
Keyboard map M: Keys $\longrightarrow [0; 255]$
Hash function HASH,
Pseudorandom number generator PRNG
Result: Main hash and list of $n-1$ (hash / integer list) pairs

begin
  $H_0 \longleftarrow$HASH(Concatenate(S_0, P))
  for i from 1 to $n-1$ **do**
    $P_i \longleftarrow P \setminus \{P[i] \bigcup P[i+1]\}$
    $H_i \longleftarrow$HASH(Concatenate((S_1, P_i)
    for j from 1 to 4 **do**
      Random_bits[j] $\longleftarrow$ PRNG(Concatenate(S_2, 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 $(H_0, (H_i, KA_i, KB_i, KC_i, KD_i)_{1 \leq i \leq n-1})$

Algorithm 6: Key-setting transposition-tolerant algorithm

Data: Salts S_0, S_1, S_2, Password P of length n
Keyboard map M: Keys $\longrightarrow [0; 255]$
Hash function HASH
Pseudorandom number generator PRNG
Result: Set of n (hash / integer list) pairs

begin
  $H_0 \longleftarrow$HASH(Concatenate((S_0, P))
  for i from 1 to $n-1$ **do**
    $P_i \longleftarrow P \setminus \{P[i] \bigcup P[i+1]\}$
    $H_i \longleftarrow$HASH($S_1 + P_i$)
    for j from 1 to 4 **do**
      Random_bits $\longleftarrow$PRNG(Concatenate(S_{j+2}, P_i))
      $\pi_{i,j} \longleftarrow$ Brassard(Random_bits)
    $LA_i \longleftarrow [\pi_{i,1}(M(i))]$
    LA_i.append($\pi_{i,1}(M($SHIFT$(P[i])))$)
    foreach $a \in$ Neighbours($P[i]$) **do**
      LA_i.append($\pi_{i,1}(M(a))$
    LA_i.sort()
    $LB_i \longleftarrow [\pi_{i,2}(M(i+1))]$
    LB_i.append($\pi_{i,2}(M($SHIFT$(P[i+1])))$)
    foreach $a \in$ Neighbours($P[i+1]$) **do**
      LB_i.append($\pi_{i,2}(M(a))$
    LB_i.sort()
    $LC_i \longleftarrow [\pi_{i,3}(M(P[i+1]))]$
    $LD_i \longleftarrow [\pi_{i,4}(M(P[i]))]$
  return $(H_0, (H_i, LA_i, LB_i, LC_i, LD_i)_{1 \leq i \leq n-1})$

Algorithm 7: Key-sending transposition-tolerant algorithm

Data: Length n, H_0 Original list (H_i, KA_i, KB_i, KC_i, KD_i) received hash H'_0 and list (H'_i, LA_i, LB_i, LC_i, LD_i) of (hash / integer lists) pairs
Result: ACCEPT if the password corresponds to the correct one or a version with an acceptable typo.

begin
  if $H_0 = H'_0$ **then**
    return ACCEPT
  else
    for i from 1 to $n-1$ **do**
      if $H_i = H'_i$ **then**
        for j from 1 to $|LA_i|$ **do**
          if $LA_i[j] = KA_i$ AND $LB_i[1] = KB_i$ **then**
            return ACCEPT
        for j from 1 to $|LB_i|$ **do**
          if $LB_i[j] = KB_i$ AND $LA_i[1] = KA_i$ **then**
            return ACCEPT
        if $LC_i[1] = KC_i$ AND $LD_i[1] = KD_i$ **then**
          return ACCEPT
  return REJECT

Algorithm 8: Key-checking transposition-tolerant algorithm

6.2 Insertion tolerance

Data: Salts S_0, S_1, $\ldots S_5$, Password P of length n
Keyboard map M: Keys $\longrightarrow [0; 255]$
Hash function HASH
Pseudorandom number generator PRNG
Result: Main hash and lists of (hash / integer) and (hash / integer list) pairs

begin
  $H_0 \longleftarrow$HASH(Concatenate(S_0, P))
  for i from 1 to n **do**
    $PA_i \longleftarrow P \setminus P[i]$ /* Removing the i-th letter from the string */
    $HA_i \longleftarrow$HASH(Concatenate(S_1, PA_i))
    Random_bits $\longleftarrow$ PRNG(Concatenate(S_2, P_i))
    $\pi_i \longleftarrow$ Brassard(Random_bits)
    $K_i \longleftarrow \pi_i(M(P[i]))$
  for i from 1 to $n-1$ **do**
    $PB_i \longleftarrow P \setminus \{P[i] \bigcup P[i+1]\}$
    $HB_i \longleftarrow$HASH(Concatenate(S_1, PB_i))
    for j from 1 to 4 **do**
      Random_bits[j] $\longleftarrow$ PRNG(Concatenate(S_2, 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
    $(H_0, (HA_i, K_i)_{1\le i\le n}, (HB_i, KA_i, KB_i, KC_i, KD_i)_{1\le i\le n-1})$

Algorithm 9: Key-setting insertion-tolerant algorithm

Data: Length n, Original hash H_0, Original list (HA_i, K_i)
Original list $(HB_i, KA_i, KB_i, KC_i, KD_i)$
Received hash 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 $H_0 = H_0'$ **then**
    return ACCEPT
  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
  return REJECT

Algorithm 10: Key-checking insertion-tolerant algorithm

6.3 Complete framework

Data: Username $NAME$, Password P of length n
Keyboard map M: Keys $\longrightarrow [0; 255]$
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* $(P) \ge 16$ **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($S[2]$, P_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
      $(H_0, (HA_i, K_i)_{1\le i\le n}, (HB_i, KA_i, KB_i, KC_i, KD_i)_{1\le i\le n-1})$

Algorithm 11: Key-setting complete 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 typo correction on very 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. This also concerns the padding characters from line 12. */
    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 12: Key-sending complete 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 13: Key-checking complete algorithm

6.4 Lower bound for the storage

Proposition 1. *Let f be a function from $\{0,1\}^*$ to $\{0,1\}^n$, such that finding x from $f(x)$ takes in expectation $\Omega(2^{\alpha \times n})$ operations, with $\alpha > 0$. Let there be a three-party system where the first party has a secret A, and the second party must check whether the third knows A or A' with $d(A, A') \leq E$ for a given distance d and a constant E. Let's also assume that the second party receives a single message from the first, followed by a single message from the second before deciding. Then for any deterministic algorithm that guarantees that a good message gets accepted with probability 1, that a random message has probability $O(n \times 2^{-n})$ of getting accepted, and that finding A takes $\Omega(2^{\alpha \times n})$ operations, the second party must store at least $n - g(n)$ bits, and the third must send at least $n - g(n)$ bits, where $g(n) \in o(n)$.*

Proof. Let us suppose that there exists $g(n) \geq C \times n$ for a constant C such that a protocol satisfying the assumptions exists, which requires sending only $n - g(n)$ bits in the first step. Then the server can only have at most $2^{n-g(n)} \in o(2^n)$ internal states. Assuming the adversary knows the protocol, as there is no other secret data, they can craft one message corresponding to each of these internal states. As there is at least one message that leads to acceptance, sending a random message from the crafted set leads to acceptance with probability $\omega(n \times 2^{-n})$. The proof follows the same idea when the message in the second step takes only $n - g(n)$ bits.

□