tPAKE: Typo-Tolerant Password-Authenticated Key Exchange

Thitikorn Pongmorrakot and Rahul Chatterjee

University of Wisconsin–Madison
{pongmorrakot,rahul.chatterjee}@wisc.edu

Abstract. Password-authenticated key exchange (PAKE) enables a user to authenticate to a server by proving the knowledge of the password without actually revealing their password to the server. PAKE protects user passwords from being revealed to an adversary who compromises the server (or a disgruntled employee). Existing PAKE protocols, however, do not allow even a small typographical mistake in the submitted password, such as accidentally adding a character at the beginning or at the end of the password. Logins are rejected for such password submissions; the user has to retype their password and reengage in the PAKE protocol with the server. Prior works have shown that users often make typographical mistakes while typing their passwords. Allowing users to log in with small typographical mistakes would improve the usability of passwords and help users log in faster. Towards this, we introduce tPAKE: a typo-tolerant PAKE, that allows users to authenticate (or exchange high-entropy keys) using a password while tolerating small typographical mistakes. tPAKE allows edit-distance-based errors, but only those that are frequently made by users. This benefits security, while still improving usability. We discuss the security considerations and challenges in designing tPAKE. We implement tPAKE and show that it is computationally feasible to be used in place of traditional PAKEs while providing improved usability. We also provide an extension to tPAKE, called adaptive-tPAKE, that will enable the server to allow a user to log in with their frequent mistakes (without ever learning those mistakes).

Keywords: passwords · authentication · password-authenticated key exchange (PAKE) · typo-tolerant password checking

1 Introduction

Authenticating users on the Internet is still primarily done using passwords. Passwords are user-chosen short secrets. A user picks a password during registering an account with a service, and then the user has to reproduce the same secret exactly during the login process to get access to their account.

Passwords, ideally, should not be stored in plaintext on the server; they are hashed using a slow cryptographic hash function, such as scrpyt [30], bcrypt [32], Argon2 [9]. The communication channel between the server and the user device is normally secured from network adversaries using SSL/HTTPS. Nevertheless,

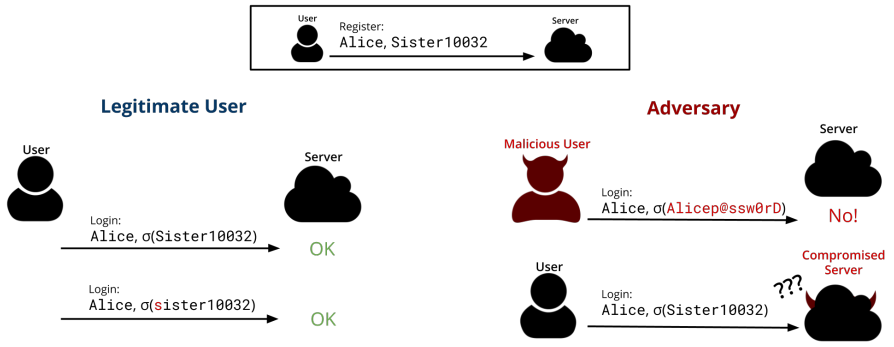


Fig. 1. Overview of how tPAKE operates. σ is a pseudo-random function that obscure the password. See Section 4 for more details.

user passwords are still exposed to the server in plaintext format every time a user tries to log in. Therefore, an attacker with persistent access to the login server can learn users' plaintext passwords as they log in. Such persistent adversarial (or sometime accidental) access to the login server is not a rare incident. For example, due to poor security practices, certain Facebook server-side applications stored plaintext user passwords on disk for several years (between 2012 and 2019), and Facebook employees could view those stored passwords [24]. In 2018, Twitter had to ask its users to change their passwords after an incident that revealed millions of user passwords in the error log [4]. Therefore, in our current setting, an adversary with persistent access to the server can learn the passwords of the users who log in during the period when the adversary has access to the login server.

Password-authenticated key exchange (PAKE) [8] is proposed to protect user passwords from getting exposed to the service during login.[1] PAKE allows a user to prove their knowledge of the password without revealing it to the server during login. Thereby, an adversary that compromises the server would not learn the user passwords even if the user logs in during the time adversary controls the server: The adversary has to crack the stored computationally expensive hash digests of the passwords.

Several PAKE protocols [8,7,6,11] have been proposed over the years. However, none of them allows even small typographical errors in the entered password by the user. A PAKE protocol will result in an error if the user, for example, accidentally switches the case of the first character of their password. Previous studies [12,13] have shown that users often make mistakes while typing their passwords, especially while typing long and complex passwords. A key challenge in encouraging users to use long and complex passwords is their usability. And

[1] It was originally designed for exchanging secret keys between two parties with knowledge of the same password over an untrusted network connection. Nonetheless, the same protocol can be used to protect passwords from being exposed to persistent adversaries who compromised the server as well. The later usage gained more interest over the years, especially as TLS can be used to exchange secrets.

if PAKE is going to be used as a de-facto protocol for future authentication, it must overcome this usability challenge and allow users to complete a PAKE protocol despite making small typographical mistakes.

To enable small error tolerance in PAKE protocols, prior work [17] has proposed a modified PAKE protocol, called *Fuzzy-PAKE*, that draws from some ideas of secure sketches [16]. However, Fuzzy-PAKE tolerates Hamming errors — the protocol succeeds if the password entered by the user is withing a small Hamming distance from the registered password. However, this error model is not realistic as typographical mistakes are better represented as edit-distance errors — the protocol should succeed if the entered password is within some *small edit-distance* from the registered password. There is no straight forward way to extend Fuzzy-PAKE to allow edit distance-based errors. Also, allowing *any* error within certain Hamming or edit distance could degrade security [12].

In this paper, we propose a new protocol, called tPAKE (short for typo-tolerant password-authenticated key exchange), that can tolerate a fixed (configurable) number of typos that are frequently made by users while typing their passwords. The core idea behind the protocol for tPAKE is simple and elegant, allowing us to provide a simple argument for its security. Intuitively, the server stores a fixed number of possible variations of the password submitted during registration. During login, the client and the server engage in a private set intersection (PSI) protocol. (The variations to allow the user to log in can be chosen by the user or the server during registration.)

Although, simple, this basic protocol has some key limitations in terms of security and efficiency. A secure PAKE must protect the user passwords from phishing attacks[2] or typo-squatting attacks[3]. That is to say, even if an adversary tricks the user into running the PAKE protocol with them, the adversary should not learn the user passwords. Similarly, the adversary also should not be able to impersonate the user as long as they don't know the user passwords (or any of the typos of that password). Therefore, the client and the server in our setting can act maliciously in the protocol (during login)[4] A schematic diagram of the functionality and threats of tPAKE is shown in Figure 1. To protect the confidentiality of the user passwords, we have to use malicious-secure PSI protocols, which are yet to be efficient (see for example [31]). For designing tPAKE, we show how to extend a known PAKE protocol to enable typo-tolerance without degrading security.

We provide two protocols for allowing typo-tolerance. In the first case, the protocol picks the set of variations to tolerate during the registration process. In the second protocol, the server adaptively learns about the mistakes that a particular user frequently makes and let the user login with those passwords, without ever learning user password or typos. The two modes of typo-tolerance are motivated by the two variations proposed in [12] and [13], respectively.

[2] `https://en.wikipedia.org/wiki/Phishing`
[3] `https://en.wikipedia.org/wiki/Typosquatting`
[4] We assume the registration process is done in a secure manner.

The protocol tPAKE utilizes oblivious pseudo-random functions (OPRF) to hide the user-entered passwords from the server. Along with the OPRF protocol, tPAKE ensures an implicit authentication so that at the end of the protocol, the server and the client learns the legitimacy of each other. During (trusted) registration, the server obtains the OPRF output of the user's password and its variations. During (untrusted) login procedure, the server only obtains proof that the user has the knowledge of the password, without obtaining the password in plaintext. We describe this protocol in Section 4. Adaptive-tPAKE extends on this protocol to enable secure storage of user's typos and other password submissions (some of which might not qualify as a typo) using a public key encryptions scheme. We describe this protocol and its security in Section 5. We prototype these protocols and measure their efficacy and computational performance in Section 6.

The main contributions of this work are the following:

(1) We design tPAKE, a PAKE protocol that allows a user to log in despite making small typographical mistakes.

(2) We analyze the security of tPAKE. We implement tPAKE, and show that the computational overhead of tPAKE is acceptable.

(3) We provide another variants of tPAKE, which we call adaptive-tPAKE, that can learn a user's typos over time (without ever seeing the typos themselves) and allow the user to log in with frequent but safe typos.

2 Background and Related Works

Despite several usability challenges, passwords are still used as the primary method for user authentication. The key challenges with passwords are: (a) they are easy to guess — low entropy secrets; and (b) hard to remember or type. In this section we discuss some background on password usability, tolerating password typos, and how to protect passwords from being revealed to a malicious server that tries to steal user passwords.

Usability challenge in password-based authentication. Users are more inclined to use simple, easily guessable passwords [10,19,29]. Mazurek et al. [28] hypothesize that one of the reasons for that is due to the increased difficulty in memorizing and typing more complex passwords. These simple passwords make these systems susceptible to various forms of guessing attacks, thus, greatly impacting the security of said systems. The increased effort required to type a more complex password can be one of the many factors that persuade users to resort to using more vulnerable passwords, especially when users mistype their passwords and redo the authentication process again. Moreover, a research [33] has found a correlation between the length of a password and the rate in which typo would occur, suggesting that longer passwords might lead to reduced usability due to mistyped passwords. Studies by Keith et al [22,23] showed that up to 2.2% of entries of user-chosen passwords had a typo (dened by thresholding via Levenshtein distance), and the rate of typos roughly doubles for more complex

passwords (at least length 7, one upper-case, one lower-case, one non-letter). A study found that up to 10% of failed login attempts fail due to a handful of simple, easily correctable typos [12]. The study also shows that out of all the typos made by users, a significant proportion of the typos are simple mistakes that can be fixed through simple operations, and fixing these typos can help reduce the frustration of mistyping their password for a significant fraction of users.

Typo-tolerant password checking. Users often make mistakes while typing their passwords. Previous studies [12,13] have shown that this causes a huge usability burden on users. Allowing legitimate users to login with small mistakes improves user experience with a platform and saves unnecessary time wasted in retyping the password. Chatterjee et al. [12] employed typo correction functions by attempting to correct a typo using a number of preset functions (e.g., changing the capitalization of the first character, removing tailing character) when the first attempt at login fails. It is stated that by just correcting a handful of easily correctable typos, it allows a significant portion of users to achieve successful login. The TypTop system [13] is a personalized password checking protocol that allows the authentication system to learns over time the typos made by a specific user from failed login attempts. The TypTop system is able to provide an authentication system with minimal security loss that benefits 45% of users as a result of personalization. tPAKE works by having the server generate a set of typos from the password during registration. Different from traditional PAKE, the key exchange procedure would be performed multiple times during each round of communication to find the value that matches the user's input. If a match is found, a shared key will be established between the client and the server. Timestamp is also used during the authentication process for verification purposes. The server would be able to customize the set of accepted typos by adjusting the typo generation function used during registration.

Password-authenticated key exchange (PAKE) [6,11] is a key exchange protocol that allows a user to convince a server that the user possesses the password without revealing the password to the server. This prevents unnecessary exposure of the password and prevents server compromise. PAKE is a cryptographic method that allows two or more parties to safely establish a shared key through the knowledge of a shared secret, in this case, a password, such that an unauthorized party would not be able to participate. The protocol only reveals whether or not the shared secret matches and not the secret itself. PAKE is also safe against interception as plain text password is not sent during login. Boyko et al. implement PAKE protocol using Diffie-Hellman, which are secure against both passive and active adversaries. PAKE works by obscuring the shared secret through exponentiation and modulo operation, in which an attacker would not be able to extract any extra information from the communication after the one-way operation.

Fuzzy PAKE [17] is a protocol based on PAKE that share a similar motivation with tPAKE. fPAKE aims to allow 2 parties to agree on a high-entropy cryptographic key without leaking information to man-in-the-middle attack. The leniency built into this protocol allows authentication to be done using a mistyped password. One of the fPAKE protocol is constructed using PAKE and Robust Secret Sharing (RSS) to allow agreement on similar passwords. However, this fPAKE protocol is limited in terms of how similarity is defined. This construction of fPAKE only allows comparison using hamming distances, which severely limits the options in how passwords can be compared as hamming distances requires two strings to be equal in length, making it impossible for fPAKE to account for password typos resulting from accidentally inserting or deleting a character. Hamming distance cannot be calculated between 'asdf1234' and 'asdf123', even though the 2 strings may very well be a typo of one another. Moreover, fPAKE is not modeled after user behavior. Typos with the same hamming distances might not have the same likelihood to happen in real-world usage as any character is more likely to be mistyped as only a certain few characters and not others, which is not a distinction that can be made using hamming distance. For instance, 'asdc1234' and 'asdp1234' would both be 1 hamming distance away from 'asdf1234', however, 'asdc1234' would be a more likely typo of 'asdf1234' assuming the user uses a QWERTY keyboard. Using Hamming distance as the metric does not account for the difference in the likelihood of one typo over another. We believe a more flexible protocol that allows the fine-tuning of typo acceptance could be helpful in modeling typo-tolerant password checking after real-world usage.

2.1 Threat model

In PAKE (and tPAKE), the user stores information about their passwords and typos during registration, which we will refer to as server state. The registration process is secure and free from adversarial interference. However, it might be possible that the user passwords are exposed during registration (or password reset) on a compromised server, but for this paper, we will ignore that threat. This is because registration (and password reset) is a rare event compared to password login events. We will discuss in Section 4 how we can modify the protocol to protect against such attacks as well.

A secure PAKE protocol must protect the user passwords from a compromised server. The goal of the attacker is to learn user passwords. There are two kinds of compromise we will consider. In the first setting, a legitimate server is compromised, and the attacker has persistent access to the server. In this case, the attacker learns all the state stored on the server as well as all the transcripts of the login protocols ran during the malicious access. The attacker can also deviate arbitrarily from the protocol. Note the attacker can do an offline brute-force attack to uncover user passwords by simulating the login protocol with the stored state. Ideally, we would like to have this as the best attack for an adversary.

The second type of compromise is where the user accidentally engages in a PAKE protocol with the adversary — due to being a victim of phishing attack or typo-squatting attack. In this case, the attacker does not have access to the server state. Therefore, the attacker should not learn anything about the user password despite actively manipulating the protocol.

Given this threat model, the passwords of the registered users should not be stored in plaintext. Currently, without a PAKE protocol, the user's password is sent to the server in plaintext. (TLS protects the password in transit, but the server has plaintext access to the password.) That means an attacker would learn the password of the user when they try to log in to the compromised server.

Another threat model we have to consider is that if a malicious user tries to learn the password of another user. A malicious user can try to use an offline dictionary-based attack on the values shared by the server across multiple sessions to learn the user password. In a secure PAKE, the attacker should not be able to do any better than online password guessing (against the server).

3 The Problem Setting and Naive Solutions

Preliminaries. Let $\mathcal{S}$ be set of all strings from alphabet Σ (e.g., printable ASCII characters) of size no more than l (e.g., 50). Though Σ is typically printable ASCII characters, in modern authentication systems, users can pick any UTF-8 characters in their passwords. Let $\mathcal{W} \subseteq \mathcal{S}$ be the set of strings that are chosen as passwords by users; we associate a probability distribution p to $\mathcal{W}$, such that $p(w)$ denotes the probability that a random password w is chosen by a user.

Users often mistype while typing their passwords. We assume $\tau_w(\tilde{w})$ denotes the probability that a user whose real password is $w \in \mathcal{W}$ types $\tilde{w} \in \mathcal{S}$. Of course, $\tau_w(\cdot)$ is a probability distribution over $\mathcal{S}$, and $\sum_{\tilde{w}\in\mathcal{S}} \tau_w(\tilde{w}) = 1$. Note, following the prior work [12], we assume that mistyping a password solely depends on the password, and not on the user who is typing. Our solution, however, can be extended to other scenarios where the typo-distributions τ also depends on the user who types the password. We only require a way to enumerate the points in the distribution in decreasing probabilities in an efficient manner. Let $\mathcal{T}_l(w)$ be an enumerator of τ_w that outputs $l+1$ most probable typos of w, including w.

For our construction, we will use a prime-order group $\mathbb{G}$ of size q where the discrete log problem is hard. We will also assume that there exists a cryptographic hash function $\mathsf{H}_{\mathbb{G}} : \{0,1\}^* \mapsto \mathbb{G}$ that can hash an arbitrary string onto the group. See [18,21] for details on how to do so.

Let H_{sl} and H_{fa} are two cryptographic hash functions that are oneway and collision-resistant. Moreover, H_{sl} is a slow and computationally expensive hash function (such as Bcrypt [32] or Scrypt [30]) ideal to be used for hashing password. The parameter for the computational overhead of this hash function can be tuned as necessary by the system engineers who deploy tPAKE. The hash function H_{fa} is a fast hash function such as SHA-256 or SHA-512. Also, we have a semantically secure and robust symmetric encryption scheme $\mathbf{SE} = (\mathsf{E}, \mathsf{D})$. Let

the security parameter be ℓ. For simplicity, we will assume the ranges of both H_{sl} and H_{fa} are $\{0,1\}^\ell$, and the domain of keys required for **SE** is also $\{0,1\}^\ell$.

3.1 Naive Proposals for making PAKE typo-tolerant

Typos are typically modeled using edit distance, also known as Levenshtein distance [26]. Therefore, to tolerate typos, one could envision accepting any variations of passwords within a certain edit distance. There are multiple issues with this approach. Firstly, any mechanism for computing edit distance coupled with PAKE will not work as the server learns nothing about the user password except it being equal to the one used during registration or not; the client obscures the original value of the password before sending to the server, making it impossible for calculating correct edit distance between the submitted password and the stored password. To overcome this challenge, we could employ secure multiparty computation technique to allow for computation of edit distance during key exchange protocol without each party revealing their secrets. However, implementation of this scheme would compromise security as it requires both parties to have the password in its plaintext form in order for the computation to be possible, which violates our security requirements. Moreover, SMC protocols secure against malicious adversaries — as required in our threat model — are slow and computationally expensive.

Besides the technical challenges, as shown in [12], allowing any typo within certain edit distance (such as ≥ 2) can degrade the security of the scheme significantly; a malicious client can try to impersonate a legitimate user by guessing their password or a variant within the given edit distance threshold. Chatterjee et al. proposed using a fixed set of variations instead of any typos within an edit-distance. The variations can be chosen based on population-wide or personal statistics and allow those that degrade security minimally. The method of finding such a list is empirical, and we refer the reader to [12] for more details.

Given a fixed number of possible variations of the user password, we could enable typo-tolerance by running multiple copies of an existing PAKE protocol — one for each variations (or corrections, as called in [12]) — albeit in parallel. However, this is not secure if the client picks the variations during login. Because in PAKE there is no way for the server to learn anything about the submitted password, a malicious client can send multiple password guesses, instead of variations of the same password. This will effectively give an attacker lx more online guesses if the protocol allows l variations of the user password. Therefore, the password variations that will be accepted must be picked by the server, and it has to be during the registration.

Finally, private set intersection (PSI) protocols can be used for building typo-tolerant PAKE (or PAKE in general): The server has a list of password variations for a user, and the user has a singleton set of the input password. The protocol attempts to determine if the intersection between the two sets is non-empty without revealing anything beyond that.[5] The biggest challenge in using off-the-

[5] This problem is more formally known as cardinality private set intersection (PSI-CA) [15].

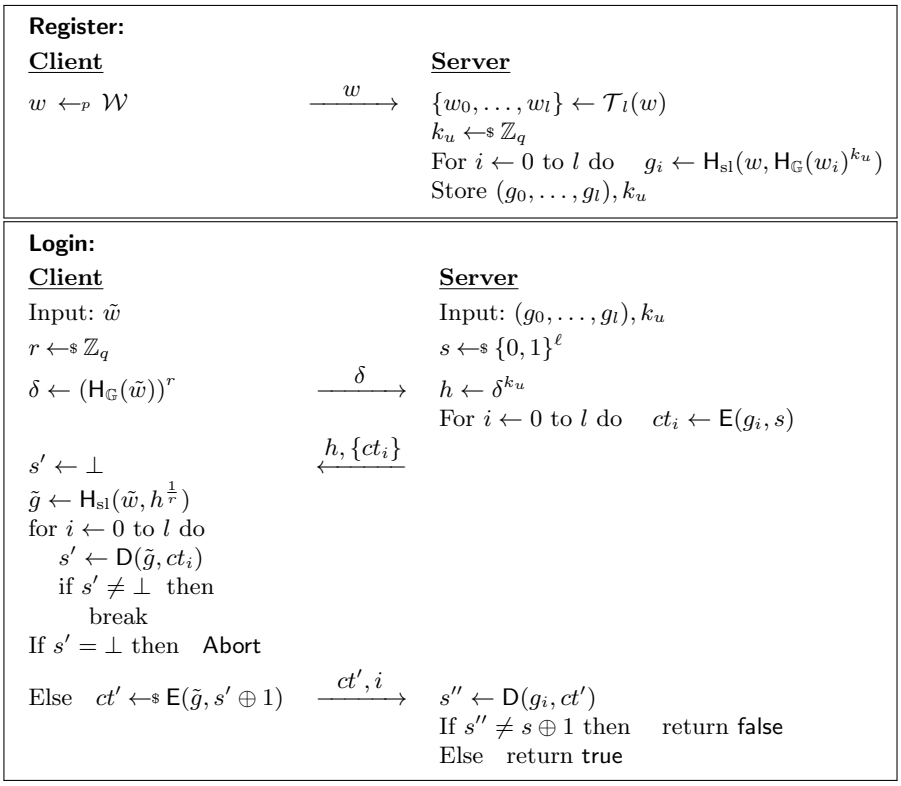


Fig. 2. Protocol for tPAKE. The client and the server are aware of the symmetric encryption scheme $\mathbf{SE} = (\mathsf{E}, \mathsf{D})$, the slow hash function H_{sl} and the (fast) hash function $\mathsf{H}_{\mathbb{G}}$, and the group (G, q). The hash function $\mathsf{H}_{\mathbb{G}}$ is used to hash any string onto a group element in G.

shelf PSI/PSI-CA protocol is that in most of those protocols one of the parties will learn the outcome, and it is hard to ensure the other party does not lie. For example, if the client gets to learn the final result that the intersection is non-zero (or zero), the client can lie to the server. A PSI protocol is not an effective solution as we need security against malicious parties, and the goal of the client and the server is slightly different in a simple PSI setting. But we take some ideas from PSI and build our tPAKE protocol that we present in details in the next section.

4 tPAKE protocol

To enable typo-tolerance, tPAKE uses oblivious pseudo-random functions (OPRF) [20] and implicit authentication to provide a secure PAKE protocol that can tolerate a set of typos in the user-entered password. Our protocol is different from prior PAKE protocols. Recall that we use a group $\mathbb{G}$ of size q, where discrete

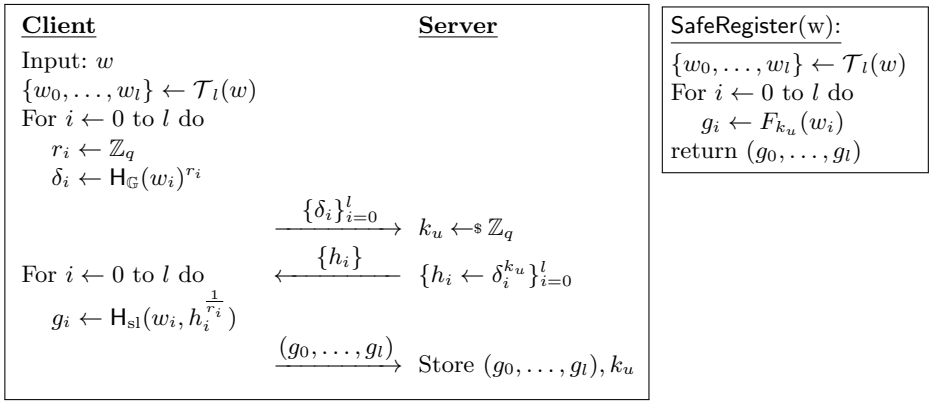


Fig. 3. Safe registration protocol for tPAKE. The server never sees the plaintext password from the user. On the **right** we show a short-hand notation that utilizes OPRF queries $F_{k_u}(\cdot)$ to the server holding the OPRF key k_u.

log problem is hard. At its core, tPAKE uses an OPRF, F_k, which we describe below.

OPRF protocol (F_k). OPRF protocols allows a client to obtain the pseudorandom output of a string x from a server holding the key k without revealing the value x. This is done by the client first sending a "blinded" value of the x, by computing $y \leftarrow \mathsf{H}_{\mathbb{G}}(x)^r$, where is a freshly chosen random number $r \leftarrow\!\!\$\ \mathbb{Z}_q$ and $\mathsf{H}_{\mathbb{G}}$ is a hash function that maps any binary string onto an element in $\mathbb{G}$. The server responds with y^k. The client then "deblind", by raising the server's response with $1/r$; that is $(y^k)^{1/r} = \mathsf{H}_{\mathbb{G}}(x)^k$. We also apply a slow hash H_{sl} to the output of the final exponentiation. Together we have, OPRF $F_k : \{0,1\}^* \mapsto \{0,1\}^\ell$, where $F_k(x) = \mathsf{H}_{\mathrm{sl}}\left(x, \mathsf{H}_{\mathbb{G}}(x)^k\right)$.

In tPAKE, during registration, the server picks a random key k_u for user u, and use that to evaluate the PRF F_{k_u} (locally) on each of the variations of the input password w (sent by the user). The server stores the PRF outputs along with the key and the username. The registration process for user u is shown at the top protocol in Figure 2. Note, we assume the registration process is done in a safe environment, which is to say that the user and the server act honestly during registration. This is a standard assumption given that registration is done only once, while login protocol is executed multiple times. For added security to prevent giving away user password to the server, we can tweak the protocol so that the server only learns hash of the password variations chosen by the client. We show this safe registration protocol in Figure 3, which will ensure that the server doesn't see the actual password in plaintext, ever.

During Login, the client masks the password $\tilde{w}$ given by the user by raising the hash $\mathsf{H}_{\mathbb{G}}(\tilde{w})$ of it to a freshly generated random value r. Then the client forwards that value $\delta = \mathsf{H}_{\mathbb{G}}(\tilde{w})^r$ to the server, which raises δ to the power k_u, the secret value for the user u, selected during registration. The server also

encrypts a freshly chosen random bitstring s using all the stored PRF outputs g_i for user u. The server sends back δ^{k_u} and the ciphertexts $\{ct_i\}$ to the client.

The client unmasks δ^{k_u} by raising it to the power $\frac{1}{r}$ and then computes $\tilde{g} = F_{k_u}(\tilde{w})$. With $\tilde{g}$, the client tries to decrypt each of the ciphertexts ct_i received from the server, and if any of the decryption succeeds, it learns that the entered password $\tilde{w}$ is either the correct password or one of its variations. One important design decision we made is to let the client know early whether the login is going succeed. Alternative options that reveal this decision to the server first could reveal the user password to a malicious or compromised server.

Finally, to prove to the server that the user successfully decrypted one of the ciphertexts, the user encrypt the modified value $s \oplus 1$ using its PRF output $\tilde{g}$. The server tries to decrypt that ciphertext using its knowledge of the stored PRF values, and if the decryption succeeds and $s'' = s \oplus 1$, the login succeeds, else the login fails.

The benefit of using encryption (instead of just hash functions, as used in prior works, such as [11]) is that we obtain implicit client and server authentication. We will expand on this next.

4.1 Security of tPAKE protocol.

The OPRF protocol we use is based on Chaum's blind signature [14]. This ensures that the server learns nothing about the user password through this OPRF call. We also assume that the hash functions are one-way and collision-resistant. The secret-key encryption scheme is semantically secure and robust, that is to say without the key, an attacker cannot learn anything about the plaintext.

As noted in our threat model (see Section 2.1), there are three threats we need to defend against. The first threat is about malicious client attacker who wants to learn about the password (or any non-trivial information about the password) of a legitimate user. The only values that are revealed to the client in the protocol are $h := \mathsf{H}_{\mathbb{G}}(\tilde{w})^{rk_u}$ and $\{ct_i := \mathsf{E}(g_i, s)\}$. Here, h does not contain any information about the passwords stored on the server; and ct_i is equivalent of random string unless the attacker can learn one of the $\{g_i\}$'s, for which the attacker has to be able to make OPRF queries on behalf of the user. This is equivalent of online guessing attack, which is the base case of any attack against authentication services. Thus a malicious client obtains no advantage over an attacker attempting to log in to a server that just allows typo-tolerant passwords.

Next threat is about a compromised server learning the user password. Ideally, a compromised server should not learn anything about the user password beyond what it already knows. There are two types of malicious server attacker. In the first case, where the attacker compromised the server, it already knows the PRF outputs $\{g_i\}$ of the user passwords. Therefore, from the interaction with the client, the attacker should not learn any information about the user password that can be used to recover the password faster than mounting dictionary attack against the g_i values. For the second case, where the attacker pretends to be a server via, say a phishing campaign, the server does not have g_i values. In

this case the server should not learn *anything* about the password. Our protocol achieves both the security goals.

First, if a server does not have correct $\{g_i\}$ values and k_u, then the client will always end up aborting the protocol. This is because, given the server does not know the input password $\tilde{w}$, it is impossible for the server to guess $\tilde{g}$ without guessing the password $\tilde{w}$ — a guarantee provided by the OPRF protocol we use. Without the knowledge of $\tilde{g}$ the server cannot construct ciphertexts ct_i, such that the decryption by the client $\mathsf{D}(\tilde{g}, ct_i)$ succeeds — a guarantee we get from the robustness of the symmetric-key encryption scheme we use. Therefore, the attempt by the client to decrypt ct_i will always fail, the client will abort, and it will not send any further communication. The only value that the serve receives from the client is $\mathsf{H}_{\mathbb{G}}(\tilde{w})^r$, which is indistinguishable from a random value in $\mathbb{G}$ — from the obliviousness guarantee provided by the OPRF. It is important to note that the abort is necessary, otherwise, the malicious server will learn the encryption of $s \oplus 1$ under $\tilde{g}$, which the server can now use to mount offline dictionary attack.

In the second case, where the server that knows g_i, the only inputs are δ, ct', i. Here, δ is indistinguishable from as a random value in $\mathbb{G}$, without the knowledge of the random exponent r; i has no information about the input password besides which typo is entered.[6] Finally, given g_i's, ct' contains no new information beyond if the login is successful or not. Therefore, the attackers best strategy is to mount guessing attack against g_i's. Therefore, the attacker who compromises the server learns no new information from the Login protocol, that can be used to learn user passwords faster. This concludes our security argument.

5 Adaptive-tPAKE Protocol

In Section 4, we give a protocol for allowing a fixed, predefined set of typos. By allowing population-wide popular typos, the previous solution limit its applicability to only those users who make "popular" typos, leaving out other users who make frequent but rather "unpopular" typos. Therefore, in this section we show how to build a typo-tolerant PAKE that can allow dynamically learned, personalized set of typos. We propose adaptive-tPAKE for the same. Adaptive-tPAKE employs a system similar to TypTop [13] that works by caching failed login attempts to let the system learn user's typos over time. PAKE's design prevents revealing the typos to the server, and, therefore, we do the testing for typo on the client-side after every successful login.

The pseudocode is shown in Figure 5. The protocol uses a secure public-key encryption scheme $\mathbf{PKE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$, where $\mathcal{K}$ is a key generation algorithm, $\mathcal{E}$ is a (randomized) encryption algorithm that uses the public key, and $\mathcal{D}$ is the decryption function that uses the secret key. So, for any message m,

[6] The server might be able to use this information to find out the most frequently entered password among $(g_0, \ldots, g_l)$. We can protect against such leakage by not sending the i, but that will require the server to try to decrypt ct' using every g_i, which is inefficient.

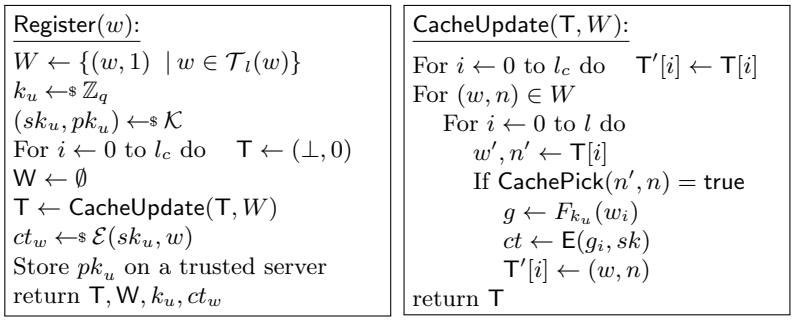


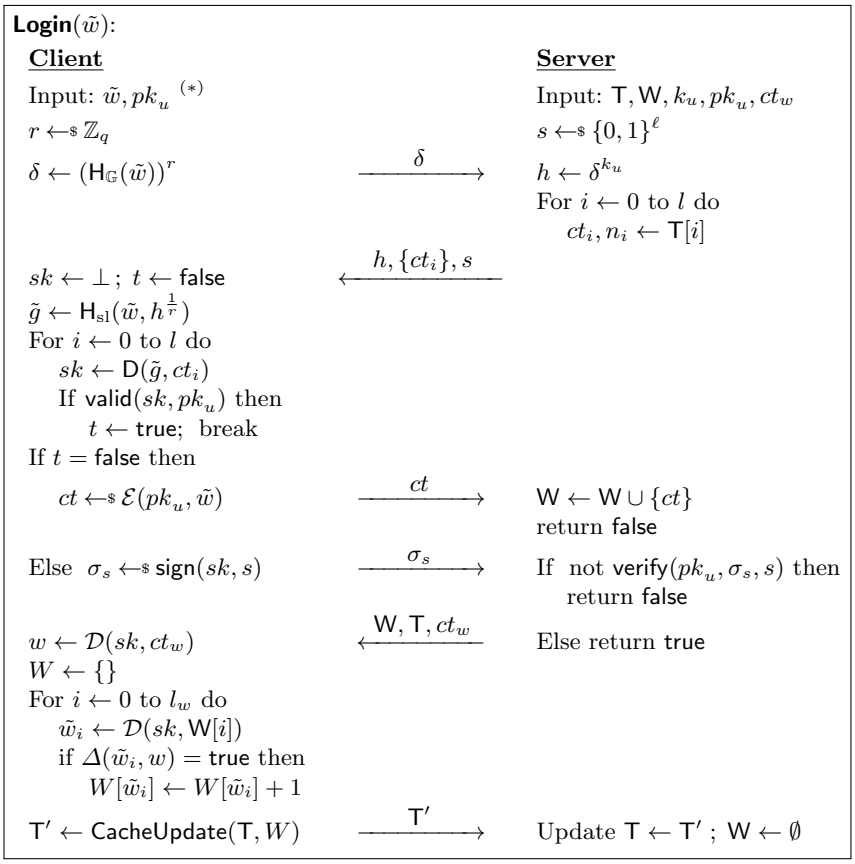
Fig. 4. (Left) ServerRegister protocol for adaptive-tPAKE. Here T is the typo-cache that keeps track of the typos the user is allowed to log in with; W is the wait list which stores the submitted passwords that are not yet checked for typo by the user. We also use a public key encryption scheme where, $\mathcal{K}$ is the key generation method. It is security critical that the public key pk_u is stored on a trusted server that the user can verify during login. **(Right)** CacheUpdate Protocol; CachePick is a function that would pick the cache line to be replaced following the caching policy(e.g. LFU, LRU, etc.). F_{k_u} is a OPRF call to the server holding the key k_u. Multiple queries to the server can be combined for network efficiency.

$\mathcal{D}(sk, \mathcal{E}(pk, m)) = m$ if $(sk, pk) \leftarrow\!\!\$\ \mathcal{K}$. We also assume the this public-key encryption scheme can be also used for signing messages. We assume there is a function $\mathsf{valid}(sk, pk)$ that verifies if the sk and pk pairs are generated together from $\mathcal{K}$. So, let sign and verify are the functions for signing and verifying functions such that for any message m, $\mathsf{verify}(pk, \mathsf{sign}(sk, m)) = \mathsf{true}$, if $\mathsf{valid}(sk, pk) = \mathsf{true}$.

The main idea behind adaptive-tPAKE is similar to that of tPAKE: use OPRF and implicit authentication. But adaptive-tPAKE has to store some more data for each user. The additional data includes a typo-cache T, and a "wait-list" W. The typo-cache holds the passwords that the user is allowed to log in with, and the wait-list holds the set of typos that are not yet verified by the legitimate user. Of course, none of these data is in plaintext. The typo-cache stores encryption of a secret key sk_u under different allowed typos. Only the legitimate user with the access to the real password or a typo of it can obtain the secret key sk_u. The secret and public keys are generated during registration (see left figure of Figure 4). The corresponding public key pk_u is uploaded to a public key repositories, such as MIT PGP public key server[7].This is important to ensure the user can reliably obtain the public key from a trusted server for security. If the public key is compromised (changed in a way that the user cannot verify), the security of this protocol is violated. We are not sure if adaptive-tPAKE can be created without the requirement of a trusted server to hold the public key.

Note that the registration makes OPRF call to the server which holds the OPRF key k_u, similar to what we did in SafeRegister. Thereby, the registration ensures that the server does not learn the plaintext passwords. The registration fills the typo-cache T with potential typos of the password (based on population-

[7] `https://pgp.mit.edu/`



(*) pk_u is obtained from the trusted server.

Fig. 5. Pseudocode for the adaptive-tPAKE protocol. Here we use a public-key encryption scheme $\mathbf{PKE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$. The typo-cache T holds the set of passwords that the user is allowed to log in with; and the waitlist W holds not-yet-verified typos. The function Δ is a function comparing two strings that return true when the two are deemed similar enough to be typo of one another.

wide statistics) and the waitlist W. In addition to these, the server also stores the encryption of the real password w under the secret key sk. This password is used later to help a user identify valid typos present in the waitlist.

The first part of the protocol is similar to tPAKE (see Figure 2), with the following difference. In adaptive-tPAKE, the server does not encrypt random value s, instead it forwards the encryption of the secret key sk to the user. Note the user can only decrypt after obtaining the OPRF output from the server. This ensures a malicious client cannot mount offline guessing attack against the user password (or typos). The server also sends challenge s in plaintext to the client, so that if the client is able to decrypt the secret key, it must sign the random value s with the sk, which the server verifies with the stored public key pk_u. If the client successfully obtains the secret key and convinces the server it is the

legitimate user, then the server hands the client the waitlist W, typo-cache T, and the ciphertext of the plain text ct_w. At this point, the client has proven its legitimacy and it is warranted to obtain these information.

The client decrypts the waitlist and checks for the typos that are valid (within small edit distance, and matches other security requirements, e.g. difficult to guess). The valid typos are then inserted into the cache. The cache update procedure is shown in the right figure of Figure 4. Interestingly, we use the same $\mathsf{CacheUpdate}$ function during registration to update the typo-cache T.

In case the user fails to obtain the secret key, it will encrypt the input password $\tilde{w}$, and submits that to the server to store it in the waitlist W.

Security of adaptive-tPAKE. The security of adaptive-tPAKE relies on the legitimacy of the public key pk_u generated during registration. If the client relies on the server to obtain pk_u, then a malicious server can hand over a pk_u for which the server knows the secret key. In that case, when the client sends the encryption of the input password due to failing to obtain the secret key, the server will learn the input password. Therefore, we use a remote trusted server for this, and leave it as a open question whether we can have a secure PAKE protocol that adaptively learns users typos over time.

Assuming the pk_u is not tampered with, we can argue adaptive-tPAKE maintains the required security. When the client fails to obtain the secret key, the only values the client learns are $h := \mathsf{H}_{\mathbb{G}}(\tilde{w})^{rk_u}$, $\{ct_i := \mathsf{E}(g_i, sk_u)\}$, and s. The client is not given the frequencies of each password, as that information can be misused for guessing attack. Following the same argument we used for tPAKE, the client learns nothing about the password in adaptive-tPAKE as well. The best the client can do is mounting online guessing attack by repeatedly calling the server to obtain the $F_{k_u}(\cdot)$. Only after the client successfully learns the secret key can the client learn the typo-cache T, the waitlist W, and ct_w.

Finally, a malicious server can try to learn about the user password. In adaptive-tPAKE, if the server does not have valid T, the client will not be able to decrypt and obtain the sk that is a valid pair of pk_u. Therefore, the client will abort after sending the encryption of the input password to the server. The input password $\tilde{w}$ is encrypted using the public key pk_u. If pk_u is legitimate, then the server cannot learn anything about $\tilde{w}$ from this.

If the server has a legitimate T, then the server will handover the typo-cache T and the waitlsit W to the client, and the client will respond with the updated T and W. During $\mathsf{CacheUpdate}$, it is ensured the client never sends plaintext password to the server — it always use the OPRF protocol to obtain the $F_{k_u}(\cdot)$ of the waitlisted passwords that are to be inserted in the cache. Thus, the server does not obtain any information that it can use to mount guessing attack faster than what it already learns from the registration process.

6 Performance Evaluation tPAKE and Adaptive-tPAKE

Implementation and test setup. We prototype tPAKE and adaptive-tPAKE to measure their computational performance and efficacy of typo-tolerance. We

	Register (ms)	Login (ms)
PAKE	37.85	65.16
tPAKE (5 passwords)	50.86	66.97
tPAKE (10 passwords)	64.94	67.56
Adaptive-tPAKE	109.50	174.54

Fig. 6. Total execution time (excluding network latency) for tPAKE registration and login.

measure the time to compute registration and login processes on the perspective of the client as well as compute some micro-benchmark on the server side. For instantiating the cryptographic primitives we use, SHA-256 as the hash function H_{fa} and H_{sl}, AES for symmetric-key encryption scheme, and brainpoolP256r1 [27] elliptic curve for the group G where discrete is known to be hard. The security parameter ℓ is chosen to be 128.

Both the server and the client are console-based Linux applications written in Python 3.6. We use Cryptography.io [5] for the symmetric-key and public-key encryption operations. For elliptic curve operations, we used `fastecdsa` library [25]. The server uses `flask` [1] for serving HTTP requests and `sqlite3` [3] for storage. On the other hand the client uses Python's `requests` [2] library for making requests. All experiments are run on Ubuntu 20.04 on an Intel Core i5 machine with 16 GB of RAM.

Performance testings are done using two separate machines in the same local network with minimal latency to avoid potential hardware and network bottleneck. The tests are done by setting up a server on one machine with the other machine acting as the client. The client would make HTTP POST requests to the server in order to log in. Server-side execution time is measured starting from when the client makes the request until the response is received. Client-side execution is measured by taking the time it takes to complete the whole protocol. Additionally, each user would not make consecutive login requests to avoid the impact of caching on the execution time.

For tPAKE, we first analyze the effect of the number of typos l on computational time and bandwidth. As shown in Figure 6, tPAKE's performance overhead depends largely on the number of the passwords that the protocol would handle. Bandwidth required is linearly proportional to the number of acceptable passwords. On the other hand, there is no significant difference in execution time between PAKE and tPAKE in our testing. Adaptive-tPAKE, however, is considerably more expensive both in term of execution time and bandwidth. Noted that in our experiment, we assume that pk_u, the public key used to encrypt fail login attempts, is available locally on the client machine, thus, the performance overhead required to acquire pk_u isn't factored into our evaluation.

The set of typos we considered for this is taken from prior work. More details on the set of typo-correctors and their efficacy are noted in Appendix A.

	Time (s)	Success (%)	attempts /success
PAKE	5.497	94.97	1.0529
tPAKE(5 passwords)	5.525	95.85	1.0432
tPAKE(10 passwords)	5.544	96.20	1.0394
Adaptive-tPAKE	5.815	96.40	1.0373

Fig. 7. Complete login test for PAKE, tPAKE, and Adaptive-tPAKE. The time column includes the time it takes for users to input the password (or passwords in case of reentry), the computation overhead on the client and the server side, and the network latency.

Execution time. Execution times for registration shown in the figure is measured on the safe registration protocol we introduced in Section 5 that ensures the security of the password during registration.

To understand the total time spent in login and the benefit of allowing typo-tolerance, we simulate real world use cases by making use of actual user password input data collected in prior work [12] via an Amazon Mechanical Turk experiment. The dataset contains multiple login attempts of each user to their (hypothetical) accounts and the time taken to enter each password We simulate user input in the order it was collected to measure the time and the number of attempts it will take for successfully login should the password checking system employ tPAKE or adaptive-tPAKE. We also measure the average login success rate and the average number of login attempts required to successfully authenticate. The test shows that tPAKE and adaptive-tPAKE improves the login success rate and reduces the number of times the user has to attempt to log in. However, the rate of typos by the users of this experiment is low ($< 4\%$), therefore the benefit of typo-tolerance is reflected in the time saved in logging in. We expect the benefit to be more visible if users are more error prone, like while entering password through mobile touch pads [12].

Bandwidth. Next we measure the bandwidth overhead for our protocol. Bandwidth is measured as the total amount of data communicated between the server and the client. In our setting, the bandwidth is affected by the number of typos handled by the protocol. For our implementation, two rounds of communication are required during login, but only bandwidths of the first response from the server are affected by the number of typos. The size of the login request is 435 bytes. The response size is 345, 837, and 1,452 bytes for PAKE, tPAKE ($l = 5$), and tPAKE ($l = 10$), respectively.

Adaptive-tPAKE requires similar packet sizes to its tPAKE counterpart depending on the size of the typo-cache T. Additionally, adaptive-tPAKE, also requires an additional round of communication, essentially doubling the bandwidth. A verification round is needed for making sure that both agree on a session key and requires an additional 374 bytes. Packets measured in our tests only includes the minimal amount of data that are required to complete the protocols, thus, additional metadata could result in the increase of packet size.

Adaptive-tPAKE. Adaptive-tPAKE enables the personalization of typo-tolerant password-checking by incorporating a typo cache, which happens to adds several layers of complexity to the implementation of the protocol. An extra round of communication is required for every login, increasing the total execution time. While adaptive-tPAKE's execution cost might be considerably more expensive than PAKE or tPAKE, adaptive-tPAKE being able to reduce the number of attempts users are required to input their password not only make up for the lost time while also improving the usability of the login process. According to our test done using data collected via Amazon MTurk, adaptive-tPAKE shows a 1% increase in login success rate compared to conventional PAKE. It only requires an average of 1.0392 login attempts to successfully login with Adaptive-tPAKE compared to 1.0502 attempts using PAKE with no typo-tolerance. The simulation also shows that the time saved from reducing the average number of attempts make up for the more expensive execution time of adaptive-tPAKE. Furthermore, the performance of adaptive-tPAKE could potentially be even better in real-world use cases where the protocol would be better personalized to a user's input habit after an extended amount of login being made. Overall, the result shows that the adaptive-tPAKE improves the login success rate compared to tPAKE without having to increase the number of acceptable passwords.

7 Conclusion

We present typo-tolerant password-authenticated key exchange, or tPAKE, a communication protocol based on password-authenticated key exchange(PAKE) that allows PAKE to be tolerant to small typographical mistakes made by users securely. We provide a formal proof to demonstrate the security of tPAKE against man-in-the-middle and compromised server. we proposed the safe registration protocol that eliminates the need to assume that communication during registration is secure and not eavesdropped by making use of PAKE's security properties. Our safe registration protocol is not only computationally secure against eavesdropper but also compromised server. Furthermore, we also present a PAKE version of the TypTop system that allows the personalization of typo-tolerant password checking with PAKE's security properties. we measure execution time and bandwidth to gauge the performance overhead of our protocols. We also conduct simulations using data gathered via a study on Amazon MTurk to quantify the potential usability gain adaptive-tPAKE has over old-school PAKE.

References

1. Flask documentation, `https://flask.palletsprojects.com/en/1.1.x/`
2. Requests, `https://requests.readthedocs.io/`
3. Sqlite, `https://www.sqlite.org/`
4. Twitter advising all 330 million users to change passwords after bug exposed them in plain text. `https://www.theverge.com/2018/5/3/17316684/twitter-password-bug-security-flaw-exposed-change-now` (2018)

5. Cryptography.io documentation (2019), https://cryptography.io/
6. Abdalla, M., Pointcheval, D.: Simple password-based encrypted key exchange protocols. In: Cryptographers track at the RSA conference. pp. 191–208. Springer (2005)
7. Bellare, M., Pointcheval, D., Rogaway, P.: Authenticated key exchange secure against dictionary attacks. In: International conference on the theory and applications of cryptographic techniques. pp. 139–155. Springer (2000)
8. Bellovin, S.M., Merritt, M.: Encrypted key exchange: Password-based protocols secure against dictionary attacks (1992)
9. Biryukov, A., Dinu, D., Khovratovich, D.: Argon and Argon2: password hashing scheme. Tech. rep., Technical report (2015)
10. Bonneau, J., Schechter, S.: Towards reliable storage of 56-bit secrets in human memory. In: 23rd USENIX Security Symposium (USENIX Security 14). USENIX (2014)
11. Boyko, V., MacKenzie, P., Patel, S.: Provably secure password-authenticated key exchange using diffie-hellman. In: Preneel, B. (ed.) Advances in Cryptology — EUROCRYPT 2000. pp. 156–171. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
12. Chatterjee, R., Athalye, A., Akhawe, D., Juels, A., Ristenpart, T.: password typos and how to correct them securely. IEEE Symposium on Security and Privacy (2016)
13. Chatterjee, R., Woodage, J., Pnueli, Y., Chowdhury, A., Ristenpart, T.: The typtop system: Personalized typo-tolerant password checking. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 329–346. ACM (2017)
14. Chaum, D.: Blind signature system. In: Advances in cryptology. pp. 153–153. Springer (1984)
15. De Cristofaro, E., Gasti, P., Tsudik, G.: Fast and private computation of cardinality of set intersection and union. In: International Conference on Cryptology and Network Security. pp. 218–231. Springer (2012)
16. Dodis, Y., Reyzin, L., Smith, A.: Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. In: Cachin, C., Camenisch, J. (eds.) Eurocrypt 2004. pp. 523–540. Springer-Verlag (2004), lNCS no. 3027
17. Dupont, P.A., Hesse, J., Pointcheval, D., Reyzin, L., Yakoubov, S.: Fuzzy password-authenticated key exchange. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 393–424. Springer (2018)
18. Farashahi, R.R., Shparlinski, I.E., Voloch, J.F.: On hashing into elliptic curves. Journal of Mathematical Cryptology **3**(4), 353–360 (2009)
19. Florencio, D., Herley, C.: A large-scale study of web password habits. In: Proceedings of the 16th International Conference on World Wide Web. pp. 657–666. WWW '07, ACM, New York, NY, USA (2007). https://doi.org/10.1145/1242572.1242661, http://doi.acm.org/10.1145/1242572.1242661
20. Freedman, M.J., Ishai, Y., Pinkas, B., Reingold, O.: Keyword search and oblivious pseudorandom functions (2005)
21. Icart, T.: How to hash into elliptic curves. In: Annual International Cryptology Conference. pp. 303–316. Springer (2009)
22. Keith, M., Shao, B., Steinbart, P.: A behavioral analysis of passphrase design and effectiveness. Journal of the Association for Information Systems **10**(2), 2 (2009)
23. Keith, M., Shao, B., Steinbart, P.J.: The usability of passphrases for authentication: An empirical field study. International journal of human-computer studies **65**(1), 17–28 (2007)

24. Krebs, B.: Facebook stored hundreds of millions of user passwords in plain text for years (2020)
25. Kueltz, A.: fastecdsa (2020), https://github.com/AntonKueltz/fastecdsa
26. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. In: Soviet physics doklady. vol. 10, pp. 707–710 (1966)
27. Lochter, M., Merkle, J.: Elliptic curve cryptography (ecc) brainpool standard curves and curve generation (Mar 2010), https://tools.ietf.org/html/rfc5639
28. Mazurek, M.L., Komanduri, S., Vidas, T., Bauer, L., Christin, N., Cranor, L.F., Kelley, P.G., Shay, R., Ur, B.: Measuring password guessability for an entire university. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 173–186. ACM (2013)
29. Morris, R., Thompson, K.: Password security: a case history. Commun. ACM **22**(11), 594–597 (1979). https://doi.org/10.1145/359168.359172, http://doi.acm.org/10.1145/359168.359172
30. Percival, C., Josefsson, S.: The scrypt password-based key derivation function (2015)
31. Pinkas, B., Rosulek, M., Trieu, N., Yanai, A.: Psi from paxos: Fast, malicious private set intersection. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 739–767. Springer (2020)
32. Provos, N., Mazieres, D.: Bcrypt algorithm. USENIX (1999)
33. Shay, R., Kelley, P.G., Komanduri, S., Mazurek, M.L., Ur, B., Vidas, T., Bauer, L., Christin, N., Cranor, L.F.: Correct horse battery staple: Exploring the usability of system-assigned passphrases. In: Proceedings of the Eighth Symposium on Usable Privacy and Security. p. 7. ACM (2012)

A Typo Analysis & Generation

Typo is handled in tPAKE by generating a preset number of typos from the password during registration, which will be used as the list of typos accepted by tPAKE during login. Therefore, it is crucial that the typos generated would coincide with typos that users will make for tPAKE to be useful, so we analyzed typos data that was collected [12,13] and compiled a list of typo generation functions that can be implemented with tPAKE.

The type of typo generation functions implemented can greatly affect the effectiveness of tPAKE, thus, typo analysis is done on collected user data to determine suitable typo generation function (typo-gen). We found that 41.00% of all typos are within 1 edit distance. We analyzed different types of typos that users tend to make by categorizing typos into 4 types, insertion, deletion, substitution, and transposition. Insertion refers to adding a character at a position in the string. Deletion means removing a character from the string. Substitution refers to replacing a character in the string with another character. Transposition is done by swapping the location of 2 existing characters in the string. Out of all typos, insertion makes up of around 30 percent, whereas deletion and substitution make up of 17 and 28 percent respectively of all the typos within 1 edit. Contrary to our expectation, however, transposition makes up only a small fraction of the typos. Only around 4 percent of all typos fixed is from transposition operation.

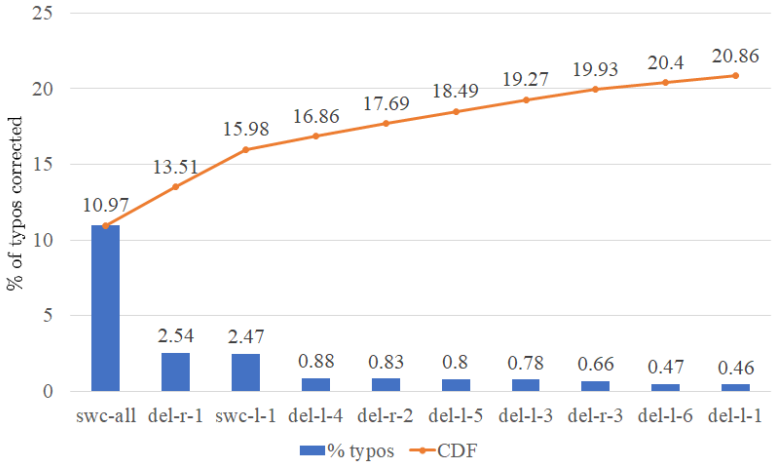


Fig. 8. Performance of different typo functions. First part (**swc**-l-1) of a function name refers to type of the operation. The second part of the name (swc-**l-1**) refers to the position that operations is applied on. E.g. swc-l-1 means substituting at the first character from the left with its shift-modified counterpart.

The substitution of characters with its shift-modified counterpart is the most common type of substitution typos, especially at the first character where the character tends to be capitalized. We use `swc-l-1` typo-generator to handle this type of typos. We found `swc-l-1` can tolerate 2.47% of all typos. While other substitution typos (non-shift substitution) and insertion typos (typos that can be generated from substitution) are common, it is difficult to identify a consistent pattern to formulate a typo-gen. Transposition typos on the other hand are few and far between, which makes it ineffective to have a typo-gen for this type of typos. `swc-all` is typo-gen that switches all the characters in a string to its shift-modified counterpart. `swc-all` proves to be effective in typo generation and is able to account for 10.97% of all the typos. Other common typo-gen are function handling different variations of deletion typos that are both common and easy to program for which make them great candidates for typo generation functions. The 10 typo generation functions included in Figure 8 account for 20.86% of all typos being made, in other words, 48.21% of all typos within one edit distance.

Similar to tPAKE, our adaptive-tPAKE protocol will only accept and cache typos that are within 1 edit away from the correct password. One advantage that Adaptive-tPAKE has over tPAKE is that it doesn't need to preemptively predict during registration what type of typos the user would make in the future, which means that it could account for typos that tPAKE could not, for instance, insertion typos that make up a significant portion of all typos. Furthermore, adaptive-tPAKE would adapt to password input habit that is unique to each user that our typo analysis could not capture.