Tipsy: How to Correct Password Typos Safely

February, 2021

Philippe Partarrieu
M.Sc. Security and Network Engineering
University of Amsterdam
philippe.partarrieu@os3.nl

Prof. Zeno Geradts
Special Chair Forensic Data Science
University of Amsterdam
geradts@uva.nl

***Abstract*— We implement and test a simple typo tolerant password authentication scheme as well as its personalised counterpart. Our experiments measure the security loss and give examples of the difference in security one could expect when moving from a "strict" authentication system to a typo tolerant one. In our tests, we use publicly available breached password datasets and mock exact knowledge attacker to mimic real-world scenarios and constraints. During our evaluation, we uncover flaws in previously published security loss calculations and we suggest a refined algorithm. Our results corroborate previous studies: in practice, typo tolerant systems offer a minimal decrease in security.**

***Index Terms*—password authentication, typo tolerance**

I. Introduction

Passwords are an important aspect of today's security landscape. In order to be considered safe, passwords must have a length greater than a given number of characters, contain special characters and have mixed capitalisation. To follow best practices, users must register different passwords per account to mitigate the risk of having an account compromised in a password breach. Although password managers are increasingly popular, adoption remains low, especially on mobile devices [1]. Longer passwords have been shown to be harder to remember and harder to type.

Because of these issues, security researchers have been wanting to deprecate passwords as a form of authentication [2]. In recent years, it has been shown that there exist typo tolerant authentication systems that can correct certain types of typos and that these are comparatively as secure as schemes that reject mistyped passwords [3].

In this research we plan on investigating the intrinsic tension between typo-tolerance and security. We will design, implement and test various typo tolerant authentication systems and compare the trade offs in terms of security.

II. Background & Related Work

Problems with password-based authentication. Previous studies have repeatedly shown that user-selected passwords are short and easily-guessable [4] [5] [6]. Intuitively we know that longer passwords are more likely to contain typos for one of two reasons: longer passwords are harder to memorise and harder to type correctly. In order to mitigate these issues, some have proposed spaced repetition of complex passwords [6], or encourage the use of a password manager. Research shows that in practice adoption remains very low [1], although we believe that password manager use is on the rise since popular browsers such as Chrome and Safari now have them built-in.

Many organisations implement password policies to attempt to improve security but complex policies such as insisting on very long passwords or requiring a frequent password change paradoxically lead to a decrease in security [5].

Typo tolerant password checking in academia. Wanting to improve the usability of passwords, Chatterjee et al. introduce a framework for reasoning about typo tolerant password checking systems for a user-selected password [3]. They prove that there exists typo tolerant authentication systems that are as secure as schemes that always reject mistyped passwords and they elaborate on various strategies for implementing typo-tolerance.

Building on these formal theories Chatterjee et al. introduce a personalised typo tolerant password checking mechanism that learns the typos made by a each user and stores them securely. Once the most frequent typos have been learned, the authentication system checks if the submitted password is either the same as the initially registered password or if it belongs to one of the learned variants [7]. They show that they can maintain strong security guarantees all the while correcting a larger set of typos than before [3].

Typo tolerant password checking in the industry. As explained by the former Facebook engineer Alec Muffet [8], Facebook's authentication system allows users to login with three variations of their registered password (since 2011):

1) The case of the first letter may be capitalised (if the first character is a letter). This correction aims to fix the tendency of mobile keyboards to capitalise the first character entered
2) The case of all letters in the password may be flipped. This correction aims to account for users who accidentally type their password with caps lock enabled
3) In recent years, Facebook has added a third variation to the set of authorised passwords: the password may contain an additional character at the end. This could correspond, for example, to users who accidentally press

a key close to the enter button (instead of the enter button) before submitting their passwords

It is not surprising that we meet these three correctors again in Chatterjee et al.'s 2014 experiment [3]. Indeed they find that these are the three simple correctors with the highest utility. It is important to note that Facebook's authentication service also considers other factors when deciding if a user's authentication attempt is successful, namely previously seen cookies, location of the request and more [8].

Around 2011, Amazon's authentication system also allowed users to authenticate with typos for a limited period of time. Any character after the 8th character was ignored and capitalisation of the password was ignored, although this was likely due to a bug [9].

Because typo tolerant authentication systems are believed to weaken security, users and security professionals have shown reservations [10] [11]. The common misconception is that accepting variations of the password will improve the success rate of online guessing attacks.

III. Problem

In the case of an exact matcher, users can only login if the password they submitted is the same as the password they have registered. We now know that an authentication system can safely try multiple variations of the submitted password with a negligible decrease in security, using one of the checkers described in [3]. Namely, Chatterjee et al. describe 3 checkers:

- *Always*: will check the submitted password as well as the 3 variations of the submitted password described earlier. While having the best acceptance utility, this checker also has the biggest decrease in the security.
- *Blacklist*: will use a blacklist of high-probability password (such as Twitter's list of banned password [12]). For every variation of the submitted password, we will verify if the variation belongs to the blacklist, if and only if it doesn't, we will check it against the registered password.
- *Approximately optimal*: will approximate the distribution of the registered passwords by using a breached password list. The checker will also approximate the distribution of typos users make when typing a password, based on a previous study [3]. Using these, the checker will solve a simple optimisation problem to compute the set of password variations that maximise utility subject to completeness and security.

We will refer to these three checkers as the "relaxed" checkers. A formal definition is given in Section V-E.

In this research we aim to design a typo tolerant authentication system that uses secure typo correction schemes. We will need to consider practical security issues that may arise, related to offline & online attacks. Since the checkers described above may perform multiple checks, we will also consider timing attacks.

IV. Research Question

What are the pitfalls when building a secure typo tolerant authentication scheme?

We also raise the following subquestions:

- What is the optimal query budget q?
- Can we improve upon the security loss evaluation published in previous research?
- What information could hypothetically be retrieved from the authentication system by using a forensic investigative approach?
- How can we mitigate timing attacks against the checkers?

V. Experiment Design & Implementation

A. Architecture

To test the various typo tolerant schemes, we have built a Go framework that implements the three relaxed checkers (always, blacklist, approximately optimal) as well as personalised typo correction (TypTop). The application was built in a modular way such that the desired checkers and schemes can easily be swapped. We will refer to our framework as tipsy. The source code is publicly available at https://github.com/ppartarr/tipsy.

B. Attackers

There exists two kinds of attackers when performing the security evaluation for online attacks:

- *Estimating attackers* are more closely related to real attackers since they do not have knowledge about the password distribution. Instead they use password leaks and custom wordlists to tweak password generation algorithms such as PCFGs, John the Ripper or Hashcat [13].
- *Exact knowledge attackers* know the exact distribution of the registered passwords. Although this is highly unlikely, this is useful to build a worst-case scenario attack. An important detail to note is that the exact knowledge attackers used in our research and previous research [3], know the registered password distribution estimation but do not have access to the blacklist used with the blacklist checker and the password distribution estimation used with the approximately optimal checker.

Our experiments will mimic a vertical attack in which targets a single account and tries multiple passwords. It is trivial to modify our experiments to a simulate a horizontal attack.

Since the relaxed checkers don't require any change to the stored password hashes we will not investigate offline attacks.

In practice, attackers might use Open Source Intelligence (OSINT) to improve their success rate.

C. Rate-limiting

We will implement configurable rate-limiting such that q queries at most can be made before a user account is locked. This is a key aspect of typo tolerance since the Free Corrections theorem from Chatterjee et al. states that "the optimal remote attack up to some query budget q is no more successful than the optimal attack against an exact checker" [3], so finding an appropriate query budget will be primordial. For many popular systems, this limit is set to 10 [14]. In our experiments we will allow an increasing number of guesses (ranging from 10 to 1 000) in order to visualise the security loss as a function of q for each checker we build.

D. Datasets

All our experiments were run on real passwords lists that were previously leaked and made publicly available on github [12]. As a requirement, the lists must contain the password along with the given password frequency. We will use this to build a password probability distribution. We selected the 3 lists that matched our requirements and had the most passwords, namely:

- RockYou. The original dataset contains over 28 million passwords. To speed up our experiment runtime, we decided to trim the dataset and only use the 1 million passwords with the highest frequency and with length 8 or greater
- PhpBB contains 254 thousand passwords with 116 thousand passwords of length 8 or greater
- Muslim Match contain 263 thousand passwords and has 105 thousand passwords with length 8 or greater.

As most authentication systems nowadays require passwords to be of length 8 or greater, we have discarded all passwords that don't match this criteria.

E. Definitions

An *exact checker* is a password verification scheme that takes in two (optionally) salted and hashed strings and returns either true or false. The first string s is the submitted password hash and the second string r is the registered password hash. Theoretically an exact checker will never output true if $s \neq r$ but we cannot guarantee this in practice because hash functions have a small probability of running into a hash collision. For the sake of simplicity we will consider the hash collision probability to be negligeable.

We can build *relaxed checkers* which apply some corrector function(s) to the submitted password s and then checks these variations with the exact checker. We call these variations the *ball* of the submitted password s and we say that it is the set $B(s) \subseteq S$ where S is the set of strings that can be chosen as passwords. See Figure 1 for an illustration.

The *corrector* functions are are listed in Appendix B. Given a string s, a corrector outputs a string $\tilde{s}$. The simplest corrector is the identity corrector function *Same* defined as $Same(s) = s$. For this research we will use the following 3 correctors: *SwitchCaseAll*, *SwitchCaseFirst*, *RemoveLast*. These correctors have the highest utility according to Chatterjee et al. [3] and they are also the correctors used in Facebook's authentication system.

In order to compare the various typo tolerant checkers, we will measure the *security loss* λ_q which is usually referred to as the "work factor" in cryptography literature and is used as a measure of security for secret key cryptosystems subject to guessing attacks [15]. We define λ_q as follows, let $w_1, w_2, ..., wq$ such that the probability of these password satisfies $p(w_1) \geq p(w_2), \geq ... \geq p(w_q)$, where $p(w)$ is the probability that a user picks a string w as their password.

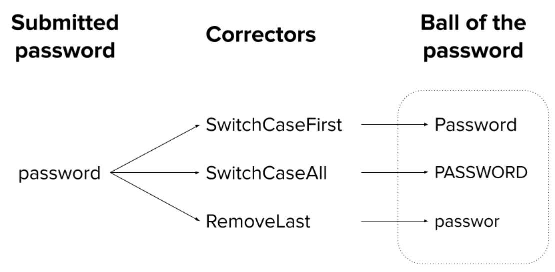


Fig. 1: Illustration of how to generate the ball of a password

For an exact checker, an adversary's success rate λ_A is bounded as follows $\lambda_A \leq \lambda_q$ for any adversary A with q queries where $\lambda_q = \sum_{i=1}^{q} p(w_i)$.

To measure the security loss of a relaxed checker we must assume the adversary can determine what ball is associated with any given string. An optimal attacker will then simply guess the passwords whose ball has the highest aggregate probability.

This is an instance of the weighted max coverage problem which is a NP-hard problem. Chatterjee et al. show the formal reduction in their seminal 2016 paper [3].

To find the set of passwords $\tilde{w}_1, \tilde{w}_2, ..., \tilde{w}_q$ that maximise an attacker's success rate against a relaxed checker, we must first define λ_q^{fuzzy} which is the maximum guessing success probability against a relaxed checker. Using the reduction we can claim that $\lambda_q^{greedy} \geq (1 - 1/e)\lambda_q^{fuzzy}$ where λ_q^{greedy} is the greedy adversary that follows this strategy:

1) Initialise a set of possible passwords P
2) Repeat the following q times:
   a) Guess a string $\tilde{w}$ that maximises $p(B(\tilde{w}) \cap P)$
   b) Remove $B(\tilde{w})$ from P and repeat

Given the set of passwords $w_1, w_2, ..., wq$ and $\tilde{w}_1, \tilde{w}_2, ..., \tilde{w}_q$ and a password distribution, we can now calculate λ_q and λ_q^{greedy}. By measuring the difference between λ_q^{greedy} (the success rate for the best attack a relaxed checker can face in practice) and λ_q (the success rate for the best attack against the exact checker), we obtain the security loss of moving from an exact checker to a relaxed checker, relative to the greedy attacker.

Since we can also calculate λ_q^{fuzzy}, we can measure the worst-case loss relative to a given password probability distribution and a typo probability distribution.

VI. Results & Discussion

A. Security loss as a function of q

It is fairly obvious that the security loss will increase for a larger number of guesses q. This is true for accross all datasets

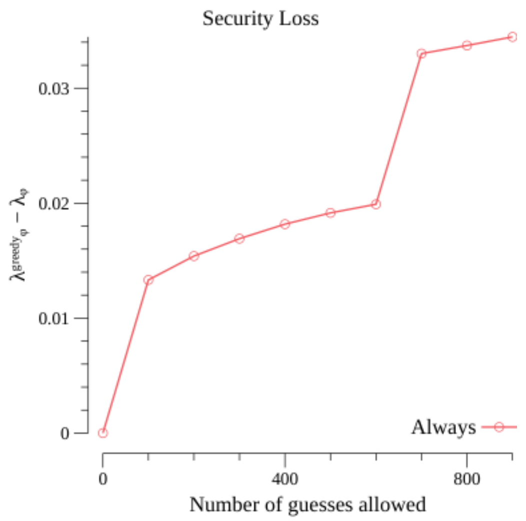


Fig. 2: security loss for the always checker, using the RockYou dataset and 3 correctors (SwitchCaseFirst, SwitchCaseAll, RemoveLast)

and checkers. This is shown in Figure 2. Therefore the best query budget q for an authentication system is the smallest q such that the user has enough guesses such that they aren't regularly locked out of their account, and that an attacker has a limited number of guesses. In practice, 10 is a good query budget [14].

B. Mitigating timing attacks

When the user submits a password, a typo tolerant authentication system must start by checking the submitted password against the registered password hash. If this fails, it gets the set of passwords to verify (depending on the checker) and performs a constant time check of the remainder of the set. Implemented in this way, we can mitigate timing attacks such that an attacker may potentially learn that a user has made a typo because authentication took longer than the exact checker but no information about the password is leaked.

If an attacker runs a timing attack, they may be able to measure the time it takes for the various relaxed checkers to get the set of passwords to check. Indeed, the approximately optimal checker has a significant overhead since it is solving a simple optimisation problem for every login request. Armed with this information, our attacker would still be considerably worse off than the exact knowledge attacker, which knows the registered password distribution as well as the checker being used.

C. Security loss against exact knowledge attackers

An exact knowledge attacker must know the password probability distribution as well as the inner workings of the checker that is being used. In practice, this is a generous assumption but it helps us depict a worst case scenario security loss e.g. a breach occurs and all password hashes are leaked and cracked.

The results in Table I report the success probability λ_q as a percentage in the Exact checker column. λ_q is simple to calculate, we just sum the probabilities of the q most probable passwords in the distribution.

We measured the security loss $\lambda_q^{greedy} - \lambda_q$ for all three checkers (always, blacklist, optimal) using all three datasets (RockYou, phpBB, Muslim Match). For the blacklist checker we used the one thousand most frequent and unique passwords from RockYou as a blacklist. For the optimal checker, we used the RockYou dataset as a password probability distribution estimation.

This explains why the security loss for the approximately optimal checker is 0 for the exact knowledge attacker when looking at the RockYou dataset in Table I or Figure 3a. If we turn our attention to phpBB and Muslim Match, we notice that there is a security loss when using the approximately optimal checker. This is because the authentication system is using phpBB or Muslim Match as the set of registered passwords but RockYou as a password distribution estimation.

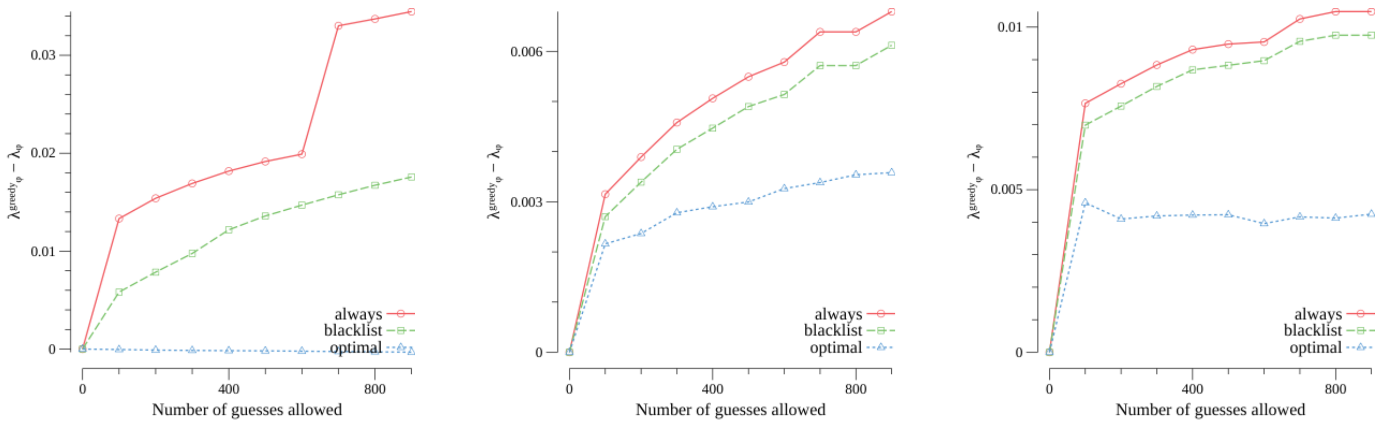
In general, the larger the difference between the password distribution estimation and the actual registered password distribution, the more λ_q^{greedy} will approach the worse-case bound λ_q^{fuzzy}. In this case, the exact knowledge attacker has more information about the registered passwords than the authentication system.

To learn more about the importance of selecting a good password distribution estimation for the optimal checker, please refer to Appendix D. We discuss some counter-intuitive results and improve upon the security loss calculation of an estimating attacker.

In practice, a secure authentication system should not know the distribution of registered passwords since it must only store the password hashes. It must therefore estimate its own registered password distribution. There are many factors that will impact this estimation such as language, keyboard layout, password strength requirements during registration, previous password leaks and more. An authentication system that wants to use the approximately optimal checker must take care when selecting the password probability distribution estimation. The same thing can be said about the typo probability distribution estimation.

In our experiments the largest security loss occurs for the RockYou dataset when using the always checker. The attacker's success increases by 3.4% from 22% to 25.4%.

Most critics of typo tolerance falsely assume that when using 3 correctors, the security loss should be 3 times that of the exact checker. According to this reasoning, the attacker's success rate would increase from 22% to 66%. This intuition isn't completely false since it holds true when the set of registered passwords is uniform [3], in which case $\lambda_q^{fuzzy} = c\lambda_q$ when $q \leq |S|/c$. In reality the set of registered passwords is not uniform and passwords with high probabilities are seldom in the ball of one another (we say they are sparse). This is important because maximising λ_q^{greedy} depends on finding passwords whose ball includes many other passwords with a high probability.



(a) RockYou dataset (b) phpBB dataset (c) Muslim Match dataset

Fig. 3: Security loss accross checkers and datasets using 3 correctors (SwitchCaseFirst, SwitchCaseAll, RemoveLast)

D. *Security loss against estimating attackers*

Since exact knowledge attackers do not exist in the real-world, we could consider an estimating attacker. If an estimating attacker uses a password generation algorithm such as Probabilistic Context Free Grammar (PCFG), they will quickly run out of guesses. PCFGs are suitable for password hash cracking which isn't bound by a number of attempts but they perform poorly in an online attack scenario. We could model estimating attacker that use a breached password distribution to estimate the registered password distribution. This is common practice and performs much better against relaxed checkers according to Chatterjee et al. [3]. Since we are interested in the viability of typo tolerance we found it more compelling to test the worse-case situations and therefore, evaluating the security loss against estimating attackers is left as future work.

E. *Personalised Typo Correction*

1) *Utility limitation of the relaxed checkers*

Thanks to the Free correction theorem, we know it is possible to check a set of strings along with the submitted password with a relatively small decrease in security. The astute reader will notice that the three correctors used by our relaxed checkers are designed to fix very simple typos. These so-called "easily-correctable" typos have been chosen because they account for a significant portion of typos and are simple to fix.

Indeed, the 3 correctors we have been using until now, namely SwitchCaseAll, SwithCaseFirst, and RemoveLast, account for 20% of all typos according to Chatterjee et al. [3]. According to the same study 21.8% of typos are due to keyboard proximity errors (pressing a key that is close to the intended key), and another 53.6% are not classified. This is an important limitation of the relaxed checkers we have been employing, since many typos are simply not corrected. Personalised typo tolerance aims to overcome this by learning the most frequent typos made by any given user over time.

2) *Design overview*

The idea is that the authentication system securely stores a set of passwords under which the user is allowed to login with. When a user submits a password, the system will check the registered password along with this set of passwords with the aim of correcting a larger set of typos then before. We will refer to this personalised typo tolerance as TypTop [7].

Here is a brief overview of how TypTop works. It uses an encrypted typo cache and an encrypted wait list. The typo cache contains the set of passwords that the user can login with and the wait list contains the recent incorrect password submission that are not already in the typo cache. TypTop makes use of two encryption schemes: a Public Key Encryption (PKE) and a Password Based Encryption (PBE). Every user has a public/private key pair.

The typo cache contains a list of ciphertexts resulting from the symmetric encryption of the secret key with the registered password or the most frequently made typos. Given typos $t_0, t_1, ..., t_n$, the typo cache is the list T containing $E(privateKey, t_0), E(privateKey, t_1), ..., E(privateKey, t_n)$ where we use the cryptographic notation for symmetric encryption $C = E(P, K)$ and $P = D(C, K)$ where C and P are ciphertext and plaintext respectively, K is the key, $E()$ and $D()$ are the encryption and decryption operations.

In turn the wait list contains a list of ciphertexts resulting from the asymmetric encryption of a recently made typo. Given typos $w_0, w_1, ..., w_m$, the wait list for user Alice is the list W containing $\{w_0\}_{Alice}, \{w_1\}_{Alice}, ..., \{w_m\}_{Alice}$. Here we use the cryptographic notation for symmetric notation where m is a plaintext message, $\{m\}_{Alice}$ is the message encrypted with Alice's public key and $[m]_{Alice}$ is the message signed by Alice's private key.

When a user attempts to login by submitting a password s, the TypTop checker attempts to decrypt every ciphertext in the wait list as shown: $P = D(T[n], s)$. If the resulting plaintext P matches the user's private key, the wait list is then decrypted and a cache eviction policy is used to decide if any typos from the wait list have been used frequently enough to be moved

Dataset	**q = 10**				**q = 100**				**q = 1000**			
	Al	Bl	AOp	Ex	All	Bl	AOp	Ex	All	Bl	AOp	Ex
RockYou	0.4	0.09	0	3.4	1.0	0.4	0	9.4	3.4	1.7	0	22
phpBB	0.1	0	-0.1	2.3	1.3	0.2	0.1	4.9	1.8	0.6	0.3	11
Muslim Match	0.5	-0.3	-0.4	6.1	1.3	0.6	0.4	11	1.9	1.0	0.4	18

TABLE I: Security loss as a function of q using 3 correctors (SwitchCaseAll, SwitchCaseFirst, RemoveLast)

into the typo cache.

Pseudo code for the TypTop scheme is given in Appendix C.

3) Security loss

TypTop has been integrated into our Tipsy framework but we have not included the TypTop checker in our security loss experiments since it requires complex changes to the underlying experiment design. In addition, the utility of personalised typo tolerance is tightly coupled to the typos in the typo cache, which get filled progressively as the user makes erroneous login attempts. Furthermore, the security loss is still dependant on the length of the typo cache and wait list similarly to how the size of the ball impacts security when looking at the relaxed checkers [3]. Due to all these reasons, the security loss evaluation of TypTop's checker is left as future work. Instead we will discuss the benefits and drawbacks of TypTop versus our three previous simple checker.

4) Limitations

One of the main benefits of the 3 relaxed checkers is that they are simple to implement. They and can easily be built into existing authentication systems that are using a exact checker because no change to the database is required. This is not the case for TypTop as it requires changes to the user model, as shown in Listing VI-E5. Rather than storing a hashed password for every user, TypTop stores a public/private key pair along with the $TypTop$ state which in turn contains the typo cache & wait list.

Personalised typo correction [7] requires many other changes to the registration and login functionality as shown in Appendix C. This considerable increase in authentication code opens up the attack surface to implementation errors and forensic analysis.

5) Benefits

When a user successfully logs into TypTop by submitting one of the passwords in the typo cache, a cache eviction policy must decide what password moves from the wait list to the typo cache. To guarantee there isn't a reduction in security, the cache update algorithm performs a number of arbitrary checks. In their paper introducing TypTop, Chatterjee et al. define a couple of requirements:

- the typo w from the wait list that is being considered for inclusion in the typo cache must not have a Damereau-Levenshtein distance greater than 1. This allows 41% of typos made by users to be considered for correction [7], which is a considerable increase compared to the 20% of relaxed checkers.
- an arbitrary password strength estimation algorithm such as zxcvbn [16] is used to determine if the typo w is easily guessable and if it's significantly more guessable than the registered password.

The beauty of TypTop lies within the extensibility of these security requirements. An authentication service with very high password strength requirements could, in theory, relax the Damerau-Levenshtein to a distance of 2 as long as the typo w satisfies the other requirements, in order to capture a even larger percentage of typos. The experiments for determining the security loss of such a configuration is left as future work.

Listing 1: Difference between the user & TypTop user model

```
type User struct {
   ID            int
   Email         string
   PasswordHash  string
   LoginAttempts int
}

type TypTopUser struct {
   ID            int
   Email         string
   LoginAttempts int
   PrivateKey    *rsa.PrivateKey
   State         *struct {
      Publickey           *rsa.PublicKey
      CipheredCacheState  []byte
      TypoCache           [][]byte
      WaitList            [][]byte
      Gamma               int
   }
}
```

VII. Conclusion

We implement and test two different kinds of typo tolerance, the first based on "relaxed" checkers and the second based on personalised corrections.

We found that adding relaxed checkers to an existing authentication system is straightforward as minimal changes are needed. These checkers can only correct a small proportion of all typos but as our results show, the security loss is minimal. This makes them interesting for authentication systems which aim to improve user experience by reducing the time users spend logging in and this justifies their use in production authentication systems.

Personalised typo correction aims to correct a larger portion of typos and increase the utility of the typo correction by personalising the typos under which a user is allowed to login with. We find that it is considerably more complex to implement and requires a complete overhaul of the registration and login functionalities, as well as a redesign of how users are stored in the database.

References

[1] Alkaldi, N. and Renaud, K. "Encouraging password manager adoption by meeting adopter self-determination needs". In: *Proceedings of the 52nd Hawaii International Conference on System Sciences*. 2019. URL: http://128.171.57.22/bitstream/10125/59920/0480.pdf.

[2] Bonneau, J. et al. "The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes". In: *2012 IEEE Symposium on Security and Privacy*. 2012, pp. 553–567. DOI: 10.1109/SP.2012.44. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=%5C&arnumber=6234436.

[3] Chatterjee, R. et al. "pASSWORD tYPOS and How to Correct Them Securely". In: (May 2016). URL: https://cs.cornell.edu/~rahul/papers/pwtypos.pdf.

[4] Florencio, D. and Herley, C. "A large-scale study of web password habits". In: *Proceedings of the 16th international conference on World Wide Web*. 2007, pp. 657–666. URL: https://dl.acm.org/doi/pdf/10.1145/1242572.1242661.

[5] Yan, J. et al. "Password memorability and security: Empirical results". In: *IEEE Security & privacy* 2.5 (2004), pp. 25–31. URL: https://prof-jeffyan.github.io/jyan_ieee_pwd.pdf.

[6] Bonneau, J. and Schechter, S. "Towards reliable storage of 56-bit secrets in human memory". In: *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 2014, pp. 607–623. URL: https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-bonneau.pdf.

[7] Chatterjee, R. et al. "The TypTop System: Personalized Typo-Tolerant Password Checking". In: (Oct. 2017). URL: https://eprint.iacr.org/2017/810.pdf.

[8] Muffet, A. "Facebook: Password Hashing & Authentication". In: *Proceedings of the International Conference on PASSWORDS 2014*. 2014. URL: https://www.youtube.com/watch?v=7dPRFoKteIU%5C&feature=youtu.be.

[9] Tweney, D. *Amazon.com Security Flaw Accepts Passwords That Are Close, But Not Exact*. https://www.wired.com/2011/01/amazon-password-problem/. Jan. 2011.

[10] Protalinski, E. *Facebook passwords are not case sensitive*. https://www.zdnet.com/article/facebook-passwords-are-not-case-sensitive-update/. Sept. 2011.

[11] *Is autocorrecting typos in passwords secure?* https://security.stackexchange.com/questions/124862/is-autocorrecting-typos-in-passwords-secure/124966. June 2016.

[12] *Leaked Databases available on github*. https://github.com/danielmiessler/SecLists/tree/master/Passwords/Le Databases. 2020.

[13] Ur Blase an/d Segreti, S. M. et al. "Measuring real-world accuracies and biases in modeling password guessability". In: *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 2015, pp. 463–481. URL: https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-ur.pdf.

[14] *Account Lockout Threshold*. https://docs.microsoft.com/en-us/windows/security/threat-protection/security-policy-settings/account-lockout-threshold#best-practices. Nov. 2018.

[15] Boztas, S. "Entropies, guessing, and cryptography". In: *Department of Mathematics, Royal Melbourne Institute of Technology, Tech. Rep* 6 (1999), pp. 2–3.

[16] Wheeler, D. L. "zxcvbn: Low-budget password strength estimation". In: *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 2016, pp. 157–173.

Appendix

A. Free corrections theorem

This section contains the free corrections theorem as originally published by Chatterjee et al. [3]

1) Definitions

Let **S** be the set of all possible strings that could be chosen as passwords. We associate to s a distribution p that models the probability that some user selects a given string w such that $p(w)$ is the probability that some user selects a given string $x \in s$ as a password.

Let $PW \subseteq S$ be the set of possible passwords. This model assumes that the distribution of passwords is independent from the user selecting them and that passwords are independently drawn from p.

Let τ be the family of distributions of s such that $\tau_w(\tilde{w})$ denotes the probability that a user types the password $\tilde{w}$ and has a registered password w. If $w \neq \tilde{w}$ then $\tilde{w}$ is a typo. Therefore $\tau_w(w)$ is the probability that the user makes no typo.

2) Optimal checker

For any set of corrector functions $C = \{f_0, f_1, ..., f_c\}$, where $f_0(\tilde{w}) = \tilde{w}$ is the identity function, we want to achieve the best-possible acceptance utility and no security loss relative to the best possible attack, assuming the checker has exact knowledge of the distribution pair (p, τ).

For some query budget q, recall that $p(w_q)$ is the probability mass of the q^{th} most probable password. Let our optimal checker $OpCheck$ be defined as follows. Upon input $\tilde{w}$, generate a list of candidate typo corrections:

$$\hat{B}(\tilde{w}) = \{w' | w' \leftarrow f_i(\tilde{w}), f_i \in C,\ and\ p(w') * \tau_{w'}(\tilde{w}) > 0\}$$

$OpCheck$ will then solve the following optimisation problem to compute the set B:

$$\underset{B \subseteq \hat{B}(\tilde{w})}{\text{maximize}} \sum_{w' \in B} p(w') * \tau_{w'}(\tilde{w})$$

which maximises utility subject to completeness and security, respectively:

$$p(\tilde{w}) > 0 \Rightarrow \tilde{w} \in B$$

$$p(B) \leq p(w_q)\ or\ |B| = 1$$

We let $B(\tilde{w})$ denote the solution to the optimisation problem.

3) The Free corrections theorem

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

4) Proof

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

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

$$p(B(\hat{S}_1)) \leq \sum_{i=1}^{|\hat{S_1}|} p(w_i) \quad (1)$$

$$p(B(\hat{S}_{>1})) \leq |\hat{S}_{<1}| * p(w_q) \leq \sum_{i=|\hat{S_1}+1}^{|\hat{S_1}|} p(w_i) \quad (2)$$

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

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

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

B. Correctors

```
// Corrector function constant names
const (
    SwitchCaseAll       = "swc-all"
    RemoveLastChar      = "rm-last"
    SwitchCaseFirst     = "swc-first"
    RemoveFirstChar     = "rm-first"
    SwitchCaseLast      = "sws-last1"
    SwitchCaseLastN     = "sws-lastn"
    UpperNCapital       = "upncap"
    NumberToSymbolLast  = "n2s-last"
    Capital2Upper       = "cap2up"
    AddOneLast          = "add1-last"
)
```

```
# Typo frequency from Chatterjee et al. study on
    password typo rate (total of 96963)
typos:
  same: 90234
  other: 1918
  swc-all: 1698
  kclose: 1385
  keypress-edit: 1000
  rm-last: 382
  swc-first: 209
  rm-first: 55
  sws-last1: 19
  tcerror: 18
  sws-lastn: 14
  upncap: 13
  n2s-last: 9
  cap2up: 5
  add1-last: 5
```

C. TypTop pseudocode

This section contains python inspired pseudo code that aims to provide a high-level overview of how TypTop works. It is adapted from the original definition given in [7] and aims to provide a simpler notation as well as fix omissions (concerning the cache warm up).

Listing 2: Registration algorithm

```
def register(submittedPassword):
  privateKey, publicKey = pke.generateKeyPair()
  typoCache[0] = pbe.encrypt(privateKey,
      submittedPassword)
  // init typo cache with random ciphertext
  for i in range(1, len(typoCache)):

    // fill typo cache with random typos
    typoCache[i] = pbe.encrypt(privateKey,
        randomPlaintext)

  // init wait list
  for j in range(0, len(waitList)):
    waitList[j] = pke.encrypt(publicKey, "")

  initialCacheState, typoIndexPairs =
      cacheInit(submittedPassword)
  ciphertext = pke.encrypt(publicKey,
      initialCacheState)

  for i in range(1, len(typoIndexPairs)):
    typo, index = typoIndexPair[i]
    typoCache[index] = pbe.encrypt(privateKey,
        typo)

  gamma = randomInt(0, len(typoCache))
  return typtopState = (
    publicKey,
    initialCacheState,
    typoCache,
    waitList,
    gamma
  )
```

Listing 3: Cache initialisation algorithm

```
def cacheInit(submittedPassword):
    for i in range(len(typoCache)):
        typoCachePasswordFrequencies[i] = 0
    cacheState = (submittedPassword,
        typoCachePasswordFrequencies)
    if cacheWarmUp:
      // return next most likely tpo according to
          a given typo distribution
      typo = nextMostLikelyTypo()
      typoIndexPairs = (typo, index)
    else:
      // empty set
      typoIndexPairs = set()

     return cacheState, typoIndexPairs
```

Listing 4: Checker algorithm

```
def check(password, typtopState):
  success = false
  for i in range(len(typoCache)):
    decryptedPrivateKey =
        pbe.decrypt(typoCache[i], password)

    if decryptedPrivateKey == privateKey:
      success = true

      // given 3 will return [[0, 1, 2], [0, 2,
          1], [1, 0, 2], [1, 2, 0], [2, 0, 1],
          [2, 1, 0]]
      permutations =
          getPermutations(len(typoCache))
      pi = permutations[randomInt(len(typoCache))]

      cacheState = pke.decrypt(privateKey,
          cipherText)

      for j in range(len(waitList)):
        decryptedWaitList[j] =
            pke.decrypt(privateKey, waitList[j])

      typoIndexPair = (password, i)
      newCacheState, typoIndexPairs =
          cacheUpdate(pi, cacheState,
          typoIndexPair, decryptedWaitList)

      cipheredCacheState = pke.encrypt(publicKey,
          newCacheState)

      // update typo cache
      for typo, index in typoIndexPair:
        typoCache[index] = pbe.encrypt(publicKey,
            typo)

      // for security, randomize the order of the
          ciphertext in the typo cache
      for j in range(len(typoCache)):
        newTypoCache[pi[j]] = typoCache[j]

      // empty the wait list
      for j in range(len(waitList)):
        waitList[j] = pke.encrypt(publicKey, "")

      typtopState = (
        publicKey,
        newCacheState,
        newTypoCache,
        waitList,
        gamma
      )
  if success == false:
    waitList[gamma] = pke.encrypt(publicKey,
        password)

    // % is the modulo operator
    newGamma = (gamma + 1) % len(waitList)

    typtopState = (
    publicKey,
    cacheState,
    typoCache,
    waitList,
    newGamma
  )

return success, typtopState
```

Listing 5: Cache update algorithm

```
def cacheUpdate(pi, cacheState, typoIndexPair,
    decryptedWaitList):
  password, typoCachePasswordFrequencies =
      cacheState
  if typoIndexPair.index > 0:
    typoCachePasswordFrequencies[typoIndexPair.index]++

  for j in range(len(waitList)):
    if valid(password, decryptedWaitList[j]):
      waitListPasswordFrequencies[password]++

  waitListPasswordFrequencies.sortDecreasing()
  for entry in decryptedWaitList:
    if waitListPasswordFrequencies[entry] > 0:
      k = argmin(typoCachePasswordFrequencies)
      nu = waitListPasswordFrequencies[entry] /
           (typoCachePasswordFrequencies[k] +
           waitListPasswordFrequencies[entry]
      if nu >= 0.5:
        typoCachePasswordFrequencies[k] =
            typoCachePasswordFrequencies[k] +
            waitListPasswordFrequencies[entry]

        newTypoIndexPair = (entry, k)
        typoIndexPairs.append(newTypoIndexPair)

  // for security, randomise order of elements in
      typo cache frequencies
  for j in range(len(typoCache)):
    newTypoCachePasswordFrequencies[pi[j]] =
        typoCachePasswordFrequencies[j]

  newCacheState = (
    password,
    newTypoCachePasswordFrequencies
  )

  return newCacheState, typoIndexPairs
```

D. Approximately Optimal Password Distribution Selection

1) Improving the security loss calculation

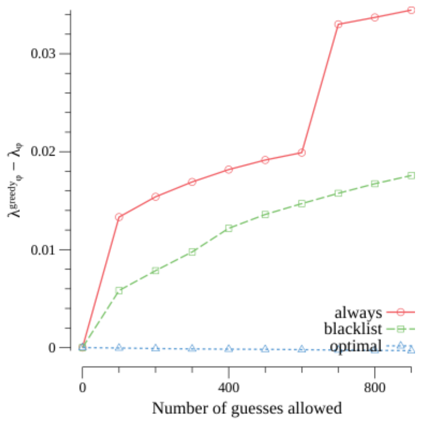
The approximately optimal checker is only as useful as the typo and password distribution estimation it employs. After solving the max coverage problem, the exact knowledge attacker has a list of passwords $\tilde{w}_1, \tilde{w}_2, ..., \tilde{w}_q$. In previous research, λ_q^{greedy} is calculated by summing the union ball of this list (for a password w, the union ball is simply $B(w) \cup w$). This provides an inflated security loss value since the approximately optimal checker and blacklist checker do not checker the attacker's submitted password against the entire ball (as the always checker does). For the blacklist and approximately optimal checkers, a better calculation of λ_q^{greedy} would only check the set returned by the passwords by the respective checkers. When calculated this way, we obtain the values shown in Figure 4.

2) Choosing the good distribution

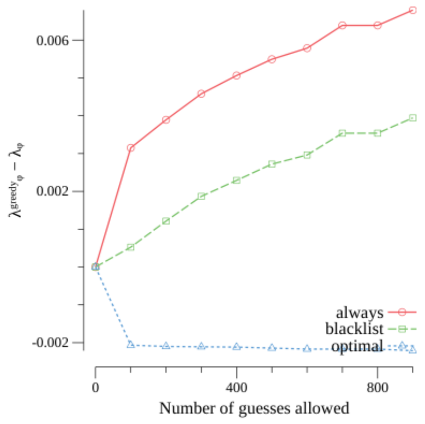
It may seem odd that we have a security increase when switching to the approximately optimal checker, but this makes statistical sense considering we are using the RockYou dataset as the password distribution estimation for the optimal checker. Remember that the exact knowledge attacker will attempt to submit the passwords with the highest aggregate probability by solving the max coverage problem. Since the distribution estimation is considerably larger than the actual set of registered passwords, the probability of the qth password in the distribution estimation will be small, which in turn makes the cutoff point small. This increases the likelyhood that the approximately optimal checker will return the empty set. Since the attacker is an exact knowledge attacker, it may at times submit passwords that do not belong to the registered password set but still have a large aggregate probability because one of the passwords in it's ball has a high probability. Because the approximately optimal checker is not using the registered passwords probability distribution but a preconfigured estimation that the exact knowledge attacker does not know, the attacker may submit guesses that neither belong to the registered password distribution or to the estimated distribution.

Due to this security increase the reader might believe it is beneficial to use a very large password distribution estimation with the optimal checker but this would be a mistake. Indeed it is wiser to use an estimation that roughly matches the size of the registered password estimation so that the approximately optimal checker returns a non-empty ball. In layman terms, this means the approximately optimal checker will correct typos.

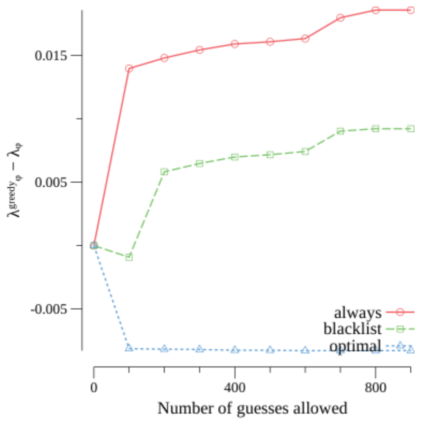
For continuity with previous research we have decided to calculate λ_q^{greedy} as the sum of probability of the passwords in the union ball of the submitted password, instead of using our refined calculation. We must therefore remember that the blacklist and approximately optimal security loss are smaller than what's reported in our results section.



(a) RockYou dataset



(b) phpBB dataset



(c) Muslim Match dataset

Fig. 4: Security loss accross checkers and datasets using 3 correctors (SwitchCaseFirst, SwitchCaseAll, RemoveLast)