VaultIME: Regaining User Control for Password Managers through Auto-correction

Le Guan[1]*, Sadegh Farhang[1], Yu Pu[1], Pinyao Guo[1],
Jens Grossklags[2], and Peng Liu[1]

[1] Pennsylvania State University, PA, USA
{lug14,farhang,yxp134,pug132,pliu}@ist.psu.edu
[2] Technical University of Munich, Germany
jens.grossklags@in.tum.de

Abstract. Users are often educated to follow different forms of advice from security experts. For example, using a password manager is considered an effective way to maintain a unique and strong password for every important website. However, user surveys reveal that most users are not willing to adopt this tool. They feel uncomfortable or even threatened, when they grant password managers the privilege to automate access to their digital accounts. Likewise, they are worried that individuals close to them may be able to access important websites holding personal data by using the stored passwords in the password manager.
We propose VaultIME to nudge more users towards the adoption of password managers by offering them a tangible benefit, while only slightly interfering with their current usage practices. Instead of "auto-filling" password fields, we propose and develop a new mechanism to "auto-correct" passwords in the presence of minor typo errors. VaultIME innovates by integrating the functionality of a password manager into the input method editor. Specifically, running as an app on mobile phones, VaultIME remembers user passwords on a per-app basis, and corrects mistyped passwords within a typo-tolerant set. We show that VaultIME achieves high levels of usability *and* security. With respect to usability, VaultIME is able to correct as many as 47.8% of password typos in a real-world password typing dataset. In terms of security, simulated attacks reveal that the security loss brought by VaultIME against a brute-force attacker is at most 0.43% in the worst case.

Key words: Password manager, Auto-correction, IME, Usable security

1 Introduction

To keep their digital accounts safe, Internet users are advised to adopt strong passwords that are hard to crack and guess [24]. However, long and random passwords are also difficult for users to remember [19]. Further, the sizable number of online accounts users need to manage has introduced an additional burden [8].

* This is the authors' full manuscript of the paper appeared in SecureComm 2017.

Using a *password manager* (e.g., 1password, lastpass and keepassdroid), which saves user credentials into a database, is a highly recommended approach by security experts. Contents in the credential database are encrypted for data protection, where the encryption/decryption key is generated from a master password only known to the user [8].

Unfortunately, adoption of password managers is behind expectations despite the benefits apparent to security experts: 1) enhancing convenience by "auto-filling" password fields on behalf of the user [24], and 2) improving security by allowing for long and complex passwords. In addition, a password manager would reduce the perceived need for insecure practices such as storing passwords in clear-text as a memory help etc. Nevertheless, surveys indicate adoption figures as low as 6% [5] and at most as high as 21% [7], which leave a lot to be desired. Further, since password manager adopters are generally more security-savvy [7], this leaves behind those users who would most benefit from the technology.

Prior survey research has shown a split between the perceptions of adopters of password managers and those that hesitate [7]. While adopters echo the security benefits lauded by experts, 78% of non-adopters perceive "some" or "a lot" of individual risk from using a password manager [7]. Some factors for hesitation are quite reasonable, and hard to address. For example, some people simply do not trust providers of password managers [35], and software vulnerabilities may lead to exposure of all user passwords to hackers [11].

Other impeding factors are more amendable to solution approaches. Specifically, the threat of a lost phone or merely unmonitored access to the phone may be perceived quite disconcerting if high value data and important services such as social networking and online banking are left more vulnerable due to the stored credentials in a password manager. In fact, otherwise trusted individuals such as family members are often the cause of such invasions [34]. According to a Javelin Research study, in 2014, there were 550,000 reports of identity theft caused by someone the victim knew [13]. Taking advantage of the bond of trust, individuals are able to more easily access family members' digital accounts and use the stolen identities to gain financial benefits [34, 20, 13]. Further, trust is especially impeded when the provider stores the password file on the cloud [35], rather than on the user's machine. In addition, empirical work shows that people prefer a high *degree of control* when completing form-fields with personal information over having the same done by auto-fill [18]; we anticipate that a similar finding could be made in the highly related context of passwords.

With our work, we want to provide a stepping stone to nudge people towards adopting a password manager by providing an easy-to-understand benefit, while limiting interference with their habituated usage practices. Further, we target adoption hesitation due to the aforementioned reasons by allowing for a higher degree of control by the user.

Concretely, we propose a mechanism to *auto-correct* passwords in the presence of minor typo errors by utilizing a client-side password vault. While the user is still required to input a "near correct" password to activate the auto-correction feature, the approach allows users to apply longer and less trivial

passwords. At the same time, user frustration can be substantially reduced by a tangible reduction of failed attempts. In this sense, our solution provides a potentially sensible middle-ground for the adoption of password managers by leaving full control over authentication in the hands of the user, and reducing the threat of stolen data when a mobile device is lost or individuals with access to the device betray the trust of the user.

While the first systematic work of password auto-correction appears in [6], it is implemented on the server-side with the purpose of increasing the password acceptance rate. The authors found that almost 10% of failed login attempts are caused by simple, easily correctable typos that should otherwise be accepted. Following this observation, the authors proposed an auto-correction framework that can be integrated into existing password-based authentication systems on the server-side. In particular, a set of correctors[2] are first defined, and a received password is adjusted by each of the correctors to generate a set of candidate passwords. The login attempt is granted provided that at least one of the candidate passwords results in a password hash value that matches the one stored on the authentication server. When it comes to the security of the typo-tolerant authentication scheme, the authors show that it does not downgrade the security of user passwords by offering a formal proof of a free correction theorem.

Different from previous server-side auto-correction, we aim to provide added convenience of password typing on the client-side to further enhance user control. We propose VaultIME, a mobile-centric password manager granting users control of password input. VaultIME integrates the functionality of a password manager into an *Input Method Editor (IME)*, which is an app that displays a software keyboard and enables users to enter text. In particular, VaultIME remembers a user password on a per-app basis. If a password input interface is detected, the auto-correction feature is activated, which replaces a mistyped password (within an acceptable set) with the correct one.

The design goals of the new password manager are as follows. First, to meaningfully reduce user frustration, the auto-correction mechanism should cover a wide range of mistypes. Second, our mechanism should not downgrade password security even if an attacker has access to the phone and could perform a brute-force attack to stored passwords. To achieve the first goal, we conducted a mobile-centric password typing analysis. Based on it, we developed a new set of password correctors, which differ from the previous work [6] and cover 26.3% more typos. To achieve the second goal, we designed VaultIME to be compatible with the free auto-correction theory of [6], which states that with a certain filter policy, auto-correction introduces *zero* security loss. To measure the security loss, we ran simulation attacks to our auto-correction scheme. In the worst case, we show that the security loss is 0.43%, assuming that a brute-force attacker has 10 tries. When configured with the filter complying with the free auto-correction theory, VaultIME introduces *zero* security loss as expected. We have developed a proof-of-concept prototype of VaultIME. With reasonable optimization, the prototype results in no user-perceivable delay when auto-correcting passwords.

[2] For example, switching caps status, removing the last character, etc.

However, interface features could be added to increase awareness of the benefits of auto-correction.

Contributions. Our work provides the following key contributions:

1. We propose a design for password managers addressing user concerns substantiated in related work. Without losing control to the login process, our design ameliorates users' concerns for using password manager in a "too open" way and maintains users' habituated login process.
2. To cover a maximum range of typos, while maintaining tight control over security, we analyze the nature of typos on a mobile platform in a systematic way. Based on the analysis results, we develop a new set of correctors, and run simulation attacks to measure the security loss introduced by VaultIME.
3. We implement a prototype of VaultIME as a normal Android IME app. Therefore, VaultIME can be instantly deployed on existing mobile platforms.

2 Background

This section explains the concept and design of the input method framework in the Android mobile OS as well as password managers. We also present difficulties users face when entering passwords on a smartphone, and summarize related efforts the community has made to reduce typos.

Input Method Editor. Since API level 3.0, Android, the most popular mobile operating system, provides an extensible input-method framework. By extending the `InputMethodService` class, developers are able to implement a customized soft keyboard for better experience and capabilities. Besides, extending the `KeyboardView` class allows for the rendering of a personalized keyboard layout. These classes are packaged together to compose an *Input Method Editor (IME)* which provides user control to enable users to enter text.

When a user inputs text for an app, the default IME pops up. The framework allows an IME to completely control user input, including reading current input, and making arbitrary modifications. These functions are supported by operating on an `InputConnection` class. In particular, method `getTextBeforeCursor` and `getTextAfterCursor` can be invoked to read input before and after the current cursor, while an app ultimately receives an input string determined by the `commitText` methods.

Password Managers. Memorizing passwords has become a significant challenge for users. Although difficult to crack by attackers, strong passwords that are sufficiently long and random are also hard for users to remember [19].

Using a password manager is one of the most recommended approaches that can free users from the duty of memorizing lots of complex passwords. Mainly developed as a plug-in for web browsers, or as stand-alone web/smartphone applications, password managers save user credentials into a database, and later automatically auto-complete requests for the credentials on behalf of users [24].

In order to ensure security of the credential database, a user controls access to the password manager database via a master password. Specifically, contents in a credential database are typically encrypted for data protection, where the encryption/decryption key is generated from a master password [8]. This mechanism makes it secure for password managers to store the password database on users' mobile devices [2, 30], in the cloud [15], or even on USB sticks [25]. Due to its advantages in both security and usability, password managers, such as 1Password [2], LastPass [21], and plug-ins in Google Chrome [10] and Mozilla Firefox [27], have seen their popularity increase during the last decade.

Mobile Typos. A number of prior studies have demonstrated the difficulty of correctly entering user-chosen passwords on laptop or desktop computers [16, 17]. However, this error-prone process is exacerbated on mobile devices.

As opposed to a physical keyboard which is standard for traditional computers, most mobile devices have moved towards a virtual keyboard, implemented by the aforementioned IME. Due to the lack of tactile feedback and the small size of soft keys, virtual keyboards have been shown to negatively influence typing abilities [29, 23]. For example, Lee and Zhai [23] report that the error rate is 8% higher for typing on virtual keyboards than on physical ones.

In addition, typing numbers or special characters on most mobile devices requires navigation to a second or third keyboard page [31], which may raise typo possibilities; and frustration if the typing of a password has to be repeated due to a typo.

Furthermore, typing while walking is a common use case in a mobile environment. However, previous research indicates that users have more difficulties typing passwords on their mobile devices while walking, and hence experience a higher error rate [26].

Reducing Typos. There has been multiple lines of research towards designing a better experience for virtual keyboards. To name a few, in [28], the authors present an approach to better process the obtained shadow information from the finger-tip, and thus to add robustness to the virtual key input. In [37], researchers show that optimizing virtual keyboard layouts can significantly improve typing efficiency and reduce errors, while the standard QWERTY keyboard could also be specifically improved for large touchscreens [14]. By dynamically adjusting the key size based on letters' frequencies, the authors designed a promising "real-time" keyboard to improve typing accuracy and speed for future virtual keyboard designs [9]. In addition, providing feedback for each typing action (e.g., a vibration) is commonly implemented in commercial Android IMEs. While these efforts are indispensable for improving typing accuracy, our work focuses on increasing the password acceptance rate even when typos occur.

3 Server Side Typo-tolerant Checking Scheme

To allow for a direct comparison, our work follows the formalization of a password authentication system proposed in [6], and also applies the same model for

evaluating security loss in the presence of a brute-force attacker. To begin with, we review some of the important concepts and notations.

3.1 System Model

Checking Passwords. Two phases are involved in a password authentication process. In the registration phase, a user registers his password, e.g., w, with the server, and the server stores another string, s, derived from a hash function mixing a random salt value and w. In the checking phase, a user submits a password, $\tilde{w}$, to the authentication server, and the server verifies the request by calculating on $\tilde{w}$ and the stored value s. The request is granted only if it returns true. In an *exact checker* (`ExChk`), the checker returns true only if the typed password $\tilde{w}$ is exactly equal to w, i.e., $\tilde{w} = w$.

Typo-tolerant Scheme. Contrary to an exact checker `ExChk`, a typo-tolerant scheme runs a *relaxed checker*, which may return a true value for multiple strings other than w. When a user submits $\tilde{w}$, the authentication algorithm, instead of only examining $\tilde{w}$, examines a set of strings neighboring $\tilde{w}$. This set is represented by a ball of $\tilde{w}$ denoted by $B(\tilde{w})$. If any element in the ball passes the exact checker `ExChk`, $\tilde{w}$ is accepted. Formally, the ball is derived by applying a set of correctors (or transformation functions) $\mathbf{C} = \{f_0, f_1, .., f_c\}$ to $\tilde{w}$.

Brute-force Attacker and Security Loss. Before formalizing a brute-force attacker, we first model the password distribution and typo distribution. The theoretical analysis of security loss introduced by a brute-force attacker against a relaxed checker assumes an attacker with exact knowledge of these distributions.

We associate a distribution p to a set of all possible passwords. Therefore, $p(w)$ is the probability that a user selects a string w as a password. A user with password w may type a password $\tilde{w}$ upon authentication. The probability of this event is represented by $\tau_w(\tilde{w})$. If $w \neq \tilde{w}$, a typo occurred. Furthermore, we say $\tilde{w}$ is a neighbor of w if $\tau_w(\tilde{w}) > 0$.

Let $\{w_1, w_2, w_3, ...\}$ be a non-increasing sequence of passwords ordered by their probabilities. $\lambda_q = \sum_{i=1}^{q} p(w_i)$ is called the q-success rate. The success rate of an attacker $\mathbf{A}$ trying to guess a user's password is denoted by `Att(checker,`$\mathbf{A}$`,`q`)`, in which checker is the checking algorithm, and q represents the maximum number of tries attacker $\mathbf{A}$ can make. For an exact checker, it is obvious that `Att(ExChk,`$\mathbf{A}$`,`q`)` $\leq \lambda_q$. To achieve λ_q, a brute-force attacker must choose the password with the highest probability in each round.

Regarding a relaxed checker, we define an *optimal attacker* to be able to achieve the maximum password guessing probability. Formally, the probability that an optimal attacker successfully guesses a password in q time is denoted by $\lambda_q^{fuzzy} = \max_{\mathbf{A}}$ `Att(Chk,`$\mathbf{A}$`,`q`)`. Similar to the case of an exact checker, where the attacker chooses the passwords with the highest probabilities, an optimal attacker against a relaxed checker tries to guess a password $\tilde{w}$, so that the corresponding ball $B(\tilde{w})$ has the highest aggregate probability in each round. The construction of such an optimal attacker is NP-hard. However, in [6], the authors

proposed a greedy algorithm to realize this attacker in practice. As a result, the security loss caused by such a greedy attacker against a relaxed checker can be calculated by $\Delta_q^{greedy} = \lambda_q^{greedy} - \lambda_q$.

3.2 Secure Typo-tolerant Checker

The naïve relaxed checker downgrades the security of an authentication system in the presence of an optimal attacker, i.e., $\Delta_q > 0$. However, there exists an optimal relaxed checker, `OpChk`, that avoids causing security degradation (*free corrections*), i.e., $\Delta_q = 0$ [6]. When a user submits a string $\tilde{w}$ as password, the relaxed checker creates a set of candidate passwords based on a set of correctors $\mathbf{C}$, and thereby a candidate set $\hat{B}(\tilde{w}) = \{w' | w' = f_i(\tilde{w}), p(w')\tau_{w'}(\tilde{w}) > 0, f_i \in \mathbf{C}\}$. To guarantee security, the optimal checker `OpChk` further rules out some of the candidate passwords by solving an optimization problem with a brute-force algorithm. `OpChk` maximizes the password acceptance rate without losing security. For the detailed explanation of the algorithm see [6]; Section V.D.

3.3 Limitations of Server-side Password Auto-correction

Previous work is invaluable as it provides a theoretical basis for a secure typo-tolerant authentication scheme, in contradiction to the common belief that accepting more than the one correct password would significantly degrade security. However, as shown in the paper, the proposed scheme cannot handle proximity typos, which, however, are the most prevalent form of all typos (21.8%). Their occurrence is even more pronounced for mobile clients (29.6%). Proximity typos occur when a user accidentally hits a key adjacent to the intended one (e.g., hitting an '`a`' instead of an '`s`'). The reason for this limitation is that correcting a proximity typo necessitates the coverage of a larger space of possible passwords, and running the hash-based authentication algorithm for each possible password requires considerable computational resources. For enterprises, this requires more infrastructure investments to enhance computing capability. For customers, the introduced latency can be unacceptable.

Drawing on the specific situational context of the mobile environment and ecosystem, we design VaultIME to overcome innate limitations of the previous work, and enable VaultIME to cover more typos. Specifically, implemented as a password manager on smartphones, VaultIME is aware of the correct password. Therefore, checking a candidate password is as simple as performing a string matching, as opposed to the complex hash calculations needed by previous work. Since computationally intensive hash computation is avoided, covering proximity typos becomes possible.

4 Empirical Study of Typos on Mobile Devices

Prior studies have shown that strong passwords are difficult to type [16, 17, 32]. For example, users could easily mistype a character by slipping to an adjacent

position on the keyboard, or they may forget to switch off the caps lock status. These human problems are further exacerbated on mobile devices. In particular, the cramped, and less tactile virtual keypad, which is widely used on today's mobile phones, has a negative influence on error-free typing [23, 29]. As a result, it has been reported that the error rate is 8% higher for text typed on virtual keypads than for physical keyboards [23].

To understand the most frequent types of typos on mobile devices, we need to analyze a sizable number of real-world password-typing observations. For this purpose, we work on publicly available password-typing datasets from the previous work [6], and particularly focus on the data collected on touchscreen mobile devices.[3] In this section, we first briefly introduce these datasets. Then, we present our analysis results. Our results uncover several new findings, which guide us in designing new mobile-centric auto-correction schemes.

4.1 Password-typing Dataset on Touchscreens

In [6], the authors carried out two experiments on the Amazon Mechanical Turk (MTurk) platform to collect typo records during the entering of passwords. One experiment collected data from either PC or mobile platforms, while the other only collected data from mobile devices with touchscreens. In collecting the latter dataset, human-intelligence tasks (HITs) were assigned to participants over the web, where each participant was required to type 10∼14 passwords in an HTML password input box within 300 seconds. The participants could only use touchscreen mobile devices. The results were later verified by the `user-agent` field in the HTTP header of the workers' browsers. The passwords were sourced from the RockYou password leak [33], one of the largest leaked password databases. In total, 24,000 password-typing records were collected by 1,987 HITs.

4.2 Understanding Typos on Mobile Devices

In this section, we explain our findings by analyzing the dataset mentioned above. We first list top typos and their corresponding correctors in Table 1. A corrector is the reverse operation of the corresponding typo. It returns a set of passwords that could potentially contain the intended one. For example, corrector `rm-last` removes the last character in the received password, which effectively corrects typo `ins-last`. While the definitions of many correctors can be found in work [6], the newly introduced ones are quite self-explanatory. For example, `rep-prox-rs` means for each character, replace it with each of the adjacent ones in the correct keyboard status.

In Table 2, we show top typos that occur in both the mobile and general datasets. Let us first have a look at the "any" row drawn directly from previous work [6]. Their solution can handle all typos except for `prox` and `others`, resulting in a coverage rate of 21.5%. However, `prox` alone contributes 21.8% of

[3] The dataset collected on touchscreen devices can be downloaded from `https://www.cs.cornell.edu/~rahul/data/mturk15-touchonly.json.bz2`.

Table 1. Top typos and their corresponding correctors.

Typo explanation	**Typo**	**Corrector**
Proximity errors, i.e., hitting an adjacent key regardless of the intended keyboard status†, e.g., typing an `'a'` as an `'S'`.	prox‡	`n/a`
Proximity errors with correct status, i.e., hitting an adjacent key in the same keyboard status with the intended one, e.g., typing an `'a'` as an `'s'`.	prox-rs	rep-prox-rs
All letters are flipped.	swc-all‡	swc-all
First letter is flipped.	swc-first‡	swc-first
An extra character is added to end.	ins-last‡	rm-last
An extra character is added to front.	ins-first‡	rm-first
Forget pressing shift for symbol at the end.	n2s-last‡	n2s-last
Miss a character at an arbitrary location.	rm-any	ins-any
Insert an extra character at an arbitrary location.	ins-any	rm-any
An arbitrary letter is flipped.	swc-any	swc-any

†: The keyboard statuses are "normal", "capitalized", and "symbolized" in the AOSP keyboard.
‡: The definition of the typo is also used in [6].

Table 2. Top typos that occur in the mobile dataset and general dataset.

Environment	Typo Percentages						
Any	prox	swc-all	ins-last	swc-first	ins-first	n2s-last	others
	21.8	10.9	4.6	4.5	1.3	0.2	56.6
Mobile	prox-rs	rm-any	ins-any	swc-all	swc-any	ins-last	others
	21.4	20.4	10.8	8.0	7.6	1.2	32.6

1. The "Any" row covers the results drawn directly from [6]. The dataset is collected from participants with PC or mobile devices.
2. The "Mobile" row covers the results obtained from mobile devices only.
3. The sum of all items in the mobile environment is greater than 1. This is because our definitions of typos are not exclusive. For example, `ins-last` is a special case of `ins-any`.

all typos, which the previous solution does not address. We have discussed the reason why previous work cannot handle proximity errors in Section 3.3.

We independently conducted a typo distribution analysis on the mobile dataset, the results of which are shown in the "Mobile" row in Table 2. Our study differs from the previous work as we are more concerned with specifics in the mobile environment. We differentiate between a virtual keyboard and a physical one, and pay more attention to the respective influences on typing.

We explain our new findings in the following. First, we find that PC users frequently make proximity typos with incorrect keyboard status, such as typing 'a' as 'S'. This can be explained by the combined effect of finger slipping and unnoticed caps status. However, mobile users seldom make such mistakes. The reason is that a virtual keyboard typically reflects the keyboard status directly on the display of each key, which a user is likely to notice. Therefore, we define a new

mobile-centric proximity error, i.e., `prox-rs`. The difference to the general `prox` is that `prox-rs` only considers proximity errors with correct caps and symbol status.[4] Therefore, typing '`a`' as '`S`' or '`@`' is not considered as a proximity error in our analysis.[5]

Apart from proximity errors, we found that mobile users frequently miss (20.4%) or insert (10.8%) a character at arbitrary locations. In addition, they may also ignore capitalization, either completely (8.0%) or only for a single letter (7.6%). Compared with the "any" environment, where the users frequently *add* an additional character, mobile users are more likely to *miss* a character. Indeed, unintentional extra key-strokes can happen due to inertia in high-speed input on physical keyboards. Among these typos, we found that correcting a missing character is challenging, i.e., a huge number of password candidates would need to be examined. This number is roughly estimated as the number of all possible characters (over 100) multiplied by the length of a password. Therefore, we do not consider this kind of typo in this work. It is also interesting to mention that both of `swc-all` and `swc-any` contribute substantially to mobile typos. While the previous work only handles `swc-all`, we argue that people are equally likely to flip only one letter, which has already been validated by our experiments. In the next section, we show how we auto-correct these typos. In total, our correctors can handle as many as 47.8% of the typos, which is the union of typos of type `prox-rs`, `ins-any`, `swc-all`, and `swc-any`.

5 Password Auto-correction for Mobile

VaultIME implements a password auto-correction scheme on the mobile client side. Instead of letting the authentication algorithm on the server judge whether a password should be accepted or not, VaultIME directly auto-corrects the passwords on the mobile client's side if only minor typos occur. To achieve this, VaultIME, as a special IME, stores the correct password for users on a per-app basis, and runs a password checker as defined in Section 3. Before a typed password is fed to the corresponding app, the checker checks the received input. If the checker returns true, the stored correct password is forwarded to the app, otherwise, the received input is forwarded as is.

More specifically, after the user is done with password input, the checker in IME first checks the received password $\tilde{w}$. If it matches with the correct password, w, recorded in the password vault, the IME leaves the password as is and returns. Otherwise, a ball $B(\tilde{w})$ of candidate passwords is derived from a predefined transformation function set $\mathbf{C} = \{f_1, ..., f_c\}$, where f_i is a corrector defined in Section 4. Then, w is compared with each element in the ball. If a match is found, $\tilde{w}$ is replaced by w; otherwise, $\tilde{w}$ is left as is.

[4] In the default AOSP keyboard layout, there are three statuses ("normal", "capitalized", and "symbolized"), which map the letter '`a`' to '`a`', '`A`', and '`@`' respectively.

[5] As a result, the results of the previous work exhibit a higher proportion of proximity error (29.6%) than measured with `prox-rs` (21.4%) on the same raw data.

This section first defines the used transformation function sets. Since we enlarge the ball size, more passwords are tested in each query. Therefore, we introduce how to apply checking policies to restrict the size of ball. It has been proven theoretically that after applying the optimal one, the security of the password is not degraded for a brute-force attacker. Then, we present how these functions influence the ball size under different checking policies. A checking policy is a filter applied to the candidate ball obtained by the naïve relaxed checker. A stricter filter leads to a reduced ball size, but retains more security of the password. Our results show that the optimal checker, `OpChk`, does not reduce the ball size significantly. Since `OpChk` has been proven to lose zero security of a password, our system can achieve both high security and high usability. Finally, we also run simulation experiments to demonstrate that our scheme is secure against a greedy attacker.

5.1 Transformation Function Sets

A transformation function is also called a corrector, which is the reverse operation of a typo, and can be used to recover the correct password. We have listed top-rated mobile correctors in Table 2. Based on their capabilities (i.e., coverage of typos) to correct typos, we define four transformation function sets. They are $\mathcal{C}_{top1} = \{rep\text{-}prox\text{-}rs\}$, $\mathcal{C}_{top2} = \mathcal{C}_{top1} \cup \{rm\text{-}any\}$, $\mathcal{C}_{top3} = \mathcal{C}_{top2} \cup \{swc\text{-}all\}$, and $\mathcal{C}_{top4} = \mathcal{C}_{top3} \cup \{swc\text{-}any\}$, respectively.

5.2 Ball Size Estimation

In [6], three checking policies are discussed. In `Chk-All`, the algorithm tries all the derived passwords in the ball $B(\tilde{w})$. In `Chk-wBL`, the ball is filtered by a predefined blacklist that is comprised of a set of frequently used passwords. In `Chk-AOp`, based on empirical distributions of passwords and typos (p, τ), a brute-force algorithm is executed to filter the ball. The algorithm maximizes the password acceptance rate without losing security against a greedy attacker who knows both the distribution (p, τ) and the algorithm of the checker.

Table 3. Average ball size for all RockYou passwords over different checker policies and transformation function sets.

	$\mathcal{C}_{top1}$	$\mathcal{C}_{top2}$	$\mathcal{C}_{top3}$	$\mathcal{C}_{top4}$
`Chk-All`	59.25	69.61	70.54	79.16
`Chk-wBL`	59.24	69.60	70.53	79.14
`Chk-AOp`	53.80	58.77	57.87	64.06

To understand the effect of policies applied to the ball, we run a simulation to calculate the averaged ball size after filtering. As shown in Table 3, the ball size decreases when policies are applied (`Chk-All` can be viewed as an all-pass policy), and increases as more transformation functions are added to the set $\mathbf{C}$. Each increase is a reflection of the added corrector. From $\mathcal{C}_{top1}$ to $\mathcal{C}_{top2}$, we observe an increment of around 10, indicating that `rm-any` produces 10 password

candidates, which conforms to the length of a password. From $\mathcal{C}_{top2}$ to $\mathcal{C}_{top3}$, only one new password is produced. This is expected because `swc-all` is a one-to-one mapping. Lastly, `swc-any` produces less than 9 new passwords as there are around 9 letters in a password on average.

Statistically, all the checkers in Table 3 significantly increase the number of candidate passwords to be checked. On the one hand, this indicates that our checkers could achieve a high auto-correction rate, because more passwords are examined in each query. On the other hand, security could be degraded because an attacker gains more information about the real password in each query. Interestingly, from `Chk-All` to `Chk-AOp`, we do not observe an abrupt shrinkage of the ball size. Since `Chk-AOp` leaks no more information about the real password than an unmodified exact checker leaks to an optimal brute-force attacker, this proves that our checker can achieve both a high auto-correction rate and a low security loss. In the next section, we show results from our simulation experiments. We emulate a greedy attacker who has complete knowledge about the implementation details of the used typo-tolerant checker.

5.3 Security Evaluation

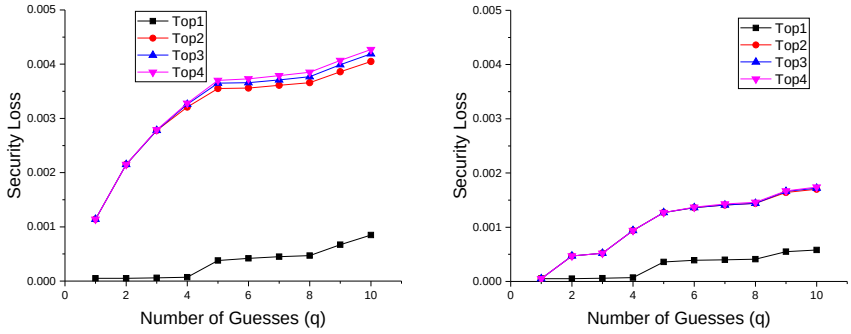
We begin by clarifying the threats we consider in this work. Then, we show the measured security losses under a set of simulated attacks.

Threats in Scope. We consider an attacker who has physical access to an unlocked victim phone. This is particularly likely to happen considering an in-house betrayer. However, we do not consider a fully compromised mobile OS. In a compromised mobile OS, the attacker may retrieve user's credential data (including all keystrokes) remotely.

We consider a brute-force attacker who is given q chances to query the authentication system. Such an attacker has been formalized in Section 3.1. Specifically, the attacker follows the greedy algorithm mentioned in Section 3.1, and the security loss can be represented by $\Delta_q^{greedy} = \lambda_q^{greedy} - \lambda_q$.

Results. In Figure 1, we show the security loss of each checker for different query numbers. We set the upper bound of q to 10, because it is a reasonable upper bound for queries given observations in practice before a device is locked. Mobile devices often enforce a long waiting time if consecutive failed login attempts are detected. For example, in the default setting of Android, q is equal to 5. After 5 unsuccessful tries, a user must wait for 30 seconds before the next screen-locking request can be processed [1]. The default policy of iOS devices is even stricter. The device will be disabled if six wrong passwords are entered in a row [4].

It is obvious that the security loss increases with q. However, `Chk-AOp` remains zero throughout our experiments, because it is an optimal checker that suffers no security loss in theory. For `Chk-All` and `Chk-wBL` shown in the figure, there is a clear gap between the transformation function set $\mathcal{C}_{top1}$ and others. This indicates that the security loss caused by applying `rep-prox-rs` alone can be quite limited – as low as 0.085% ($\lambda_q^{greedy} = 0.02937$ and $\lambda_q 0.02852$) in the worst case when



(a) `Chk-All`. (b) `Chk-wBL`.

Fig. 1. Security loss measured for different checkers and query numbers. Note that the security loss for `Chk-AOp` is zero, so we omit it for the sake of fine typography.

Table 4. Security loss measured against attackers with estimated password distributions. The challenge distribution is fixed to be the RockYou database. To show the worse case security, we used $\mathcal{C}_{top4}$ as the transformation function set.

q	1			5			8			10		
λ_q under `ExChk`	0.01368			0.02400			0.02697			0.02852		
checker	`Chk-All`	`Chk-wBL`	`Chk-AOp`	`Chk-All`	`Chk-wBL`	`Chk-AOp`	`Chk-All`	`Chk-wBL`	`Chk-AOp`	`Chk-All`	`Chk-wBL`	`Chk-AOp`
RockYou	+0.00114	+0.00005	+0.00000	+0.00370	+0.00127	+0.00000	+0.00385	+0.00146	+0.00000	+0.00427	+0.00174	+0.00000
phpBB	+0.00113	+0.00004	+0.00000	−0.00075	−0.00568	−0.00571	−0.00176	−0.00319	−0.00391	−0.00216	−0.00359	−0.00391
Myspace	−0.01024	−0.01307	−0.01304	−0.00246	−0.02172	−0.02124	−0.00407	−0.02383	−0.02361	−0.00408	−0.02418	−0.02433

$q = 10$ using checker `Chk-All`. This can be explained by the fact that a proximity typo often leads to low probability passwords, which do not increase the overall aggregate probability of the attacker's ball. For example, when checking the password '`password`', `rep-prox-rs` will derive a huge ball containing candidate passwords such as '`oassword`' and '`psssword`', which are rarely used by humans. On the other hand, applying `swc-all` will obtain '`PASSWORD`', which is also a frequently used password. In the worst case, the security loss is 0.427% ($\lambda_q^{greedy} = 0.03279$ and $\lambda_q = 0.02852$) when $q = 10$ and using checker `Chk-All` under the transformation function set $\mathcal{C}_{top4}$.

Attacker with Incorrect Password Distribution. In practice, when calculating the trial passwords, a greedy attacker does not necessarily hold the same password distribution used by the checker. The password distribution used by the attacker is called an attacker distribution while the distribution used by the checker is called a challenge distribution. We ran the same experiment as above except that we used different attacker distributions (based on the phpBB [12] and Myspace [22] password leakage datasets).

The security loss for attackers who use an estimated password distribution is shown in Table 4. The maximum success rate an attacker could reach under the exact checker `ExChk` is measured by calculating the aggregate possibility of

the top q passwords in the RockYou password distribution, and is shown in the second row. These figures serve as the baseline for calculating security loss under different checkers and attackers. As shown in the table, when an attacker choses a different password distribution, he would gain negative security loss, meaning that he did even worse under a typo-tolerant checker than under an exact checker. This can be explained by the fact that the wrong password distribution may mislead the attacker to choose a suboptimal trial set.

6 Implementation and Evaluation

We have implemented a proof-of-concept prototype of VaultIME for the Android OS. A user is able to customize the transformation function set ranging from $\mathcal{C}_{top1}$ to $\mathcal{C}_{top4}$, and the checking algorithms among `Chk-All`, `Chk-wBL`, and `Chk-AOp`.

The prototype uses the standard QWERTY US keyboard layout. It automatically detects the attribute of the current `TextView`, and inserts an "`AuCo`" key in the bottom right of the keyboard for the `TYPE_TEXT_VARIATION_PASSWORD` and `TYPE_TEXT_VARIATION_VISIBLE_PASSWORD` input types. VaultIME records a new password entry when the "`AuCo`" key is pressed. We use the package name of a login app and the account information as the key to index the password. Once a correct password has been recorded, subsequent login attempts will go through the typo-tolerant checker to auto-correct possible typos. As with traditional password vaults, the file storing passwords is encrypted by a secure master key [3]. The master key is randomly generated, and managed by the Android KeyStore provider.

Performance Evaluation. We measured the average time needed to run each checker. In particular, 1000 password-typing records from the dataset mentioned in Section 4.1 are randomly chosen and tested. We normalized the results based on the time consumption for `Chk-wBL` under $\mathcal{C}_{top1}$, because it is the most efficient checker in our implementation. As shown in Table 5, `Chk-All` and `Chk-wBL` perform much better than `Chk-AOp`. This is because running `Chk-AOp` requires a brute-force calculation to obtain all the subsets of the ball, which is time consuming. In fact, data measured in Table 5 is the result of an optimized implementation of `Chk-AOp`, that is, only passwords with non-zero probability in the ball are considered. If there are still more than 20 passwords in the ball, running the brute-force algorithm for finding all subsets of the ball becomes impractical. In this case, we abort auto-correction. In our experiments,

Table 5. Normalized time consumption for each checker.

	$\mathcal{C}_{top1}$	$\mathcal{C}_{top2}$	$\mathcal{C}_{top3}$	$\mathcal{C}_{top4}$
`Chk-All`	1.03	1.17	1.14	1.25
`Chk-wBL`	1.00	1.08	1.10	1.20
`Chk-AOp`	2.21	2.55	3.30	5.14

such a case rarely happened, because human passwords are sparely distributed. In all settings, no obvious delay can be observed.

Optimization. In the proof-of-concept implementation, we ran the auto-correction algorithm for every login operation. However, many of the intermediate results are common in each run for the same app. Therefore, as a trade-off between time and storage, one potential optimization could be to calculate all the acceptable passwords for each app beforehand in the background. In particular, after obtaining the correct password w, VaultIME runs a reverse operation of each corrector in $\mathcal{C}_{topx}(x = 1 \dots 4)$, which gets a set of neighboring passwords $\tilde{w}$ that can be auto-corrected. Then, each $\tilde{w}$ is tested by the activated typo-tolerant checker. Only those that are accepted remain. With this precomputed acceptance set, performance can be further improved. Note since the acceptance set is calculated in the background (potentially during a low-intensity usage period), users likely will not perceive these computations as disturbing. Further optimization in the form of human-in-the-loop experimentation is left for future work.

7 Future Work

In the future, we plan to conduct user studies to investigate the usability of the VaultIME app as well as adoption intentions in detail. Specifically, by empirically evaluating how users interact with our system, we aim to deliver a more usable and secure user experience for mobile phone users. Moreover, we are interested to learn to which degree users prefer our method to the traditional auto-fill password manager, whether users feel less threatened, have less frustration, and whether the correction process fits users' habituated login process.

In evaluating the security loss imposed by VaultIME, we mainly focus on a brute-force attacker who attempts to maximize the possibility coverage in each guessing. However, given that some personal data is publicly available (e.g., user name, birthday, etc.), particularly to family members or close friends, a targeted guessing attack could be more efficient [36]. Building an attack model which incorporates personal information into the on-line guessing and designing a new free auto-correction schema specific to this model constitutes an interesting research topic.

8 Conclusion

In this paper, we present VaultIME, a new password auto-correction scheme for mobile platforms. Our work ameliorates concerns of password manager users that they lack control over the use of their credentials. We achieve this by requiring the user to type a "near correct" password, which is automatically replaced with the correct one.

In designing the auto-correction policies, we conduct a mobile-centric password typo analysis, and are able to categorize the observed typos occurring while using virtual keyboards. Based on these empirical observations, we are able to develop a customized set of password correctors, which can cover as much as 47.8% of the detected password typos on mobile systems. This substantial coverage is made possible through a client-side implementation of our password-correction scheme as an app which allows for the treatment of the most common typographical errors, i.e., proximity typos. Moreover, the proposed auto-correction scheme is secure against a brute-force attacker under the formal model proposed in [6]. Our experimental results reveal that in the worst case, our scheme causes a security loss of 0.43%, indicating our auto-correction scheme has a high level of security robustness.

Acknowledgments: We would like to thank the anonymous reviewers for their insightful comments that helped to improve our paper. This work was supported by NSF CNS-1422594, ARO W911NF-13-1-0421 (MURI), and the German Institute for Trust and Safety on the Internet (DIVSI).

References

1. Changes on waiting time for wrong input on Galaxy S6. `http://gadgetguideonline.com/galaxys6/galaxy-s6-android-marshmallow-update-guide/changes-on-waiting-time-for-wrong-input-on-galaxy-s6-lock-screen-in-android-marshmallow-update/`.
2. AgileBits, Inc. 1password. `https://1password.com/`.
3. AgileBits, Inc. 1password security. `https://support.1password.com/1password-security/`.
4. Apple Inc. If you forgot the passcode for your iPhone, iPad, or iPod touch, or your device is disabled. `https://support.apple.com/en-us/HT204306`.
5. R. Butler and M. Butler. The password practices applied by South African online consumers: Perception versus reality. *South African Journal of Information Management*, 17(1):1–11, 2015.
6. R. Chatterjee, A. Athalye, D. Akhawe, A. Juels, and T. Ristenpart. pASSWORD tYPOS and how to correct them securely. In *IEEE Security and Privacy (S&P)*, 2016.
7. M. Fagan and M. Khan. Why do they do what they do?: A study of what motivates users to (not) follow computer security advice. In *SOUPS '16*, 2016.
8. P. Gasti and K. Rasmussen. On the security of password manager database formats. In *ESORICS '12*, 2012.
9. D. Gelormini and B. Bishop. Optimizing the android virtual keyboard: A study of user experience. In *IEEE International Conference on Multimedia and Expo Workshops*, 2013.
10. Google Inc. Chrome. `https://www.google.com/chrome/`.
11. A. Gott. Important security updates for our users, 2017. `https://blog.lastpass.com/2017/03/important-security-updates-for-our-users.html/`.
12. R. Graham. PHPBB password analysis. `http://www.darkreading.com/risk/phpbb-password-analysis/d/d-id/1130335`, 2009.

13. K. Grant. Identity theft victims: You might know the culprit, 2015. http://www.cnbc.com/2015/07/21/identity-theft-victims-may-know-the-culprit.html.
14. H. Hakoda, B. Shizuki, and J. Tanaka. QAZ keyboard: QWERTY based portrait soft keyboard. In *International Conference of Design, User Experience, and Usability*, pages 24–35. Springer, 2016.
15. KeePassDroid. Dropbox and KeePassDroid. http://blog.keepassdroid.com/2010/06/dropbox-and-keepassdroid.html.
16. M. Keith, B. Shao, and P. Steinbart. The usability of passphrases for authentication: An empirical field study. *International Journal of Human-Computer Studies*, 65(1):17–28, 2007.
17. M. Keith, B. Shao, and P. Steinbart. A behavioral analysis of passphrase design and effectiveness. *Journal of the Association for Information Systems*, 10(2):63–89, 2009.
18. B. Knijnenburg, A. Kobsa, and H. Jin. Counteracting the negative effect of form auto-completion on the privacy calculus. In *ICIS '13*, 2013.
19. S. Komanduri, R. Shay, P. Kelley, M. Mazurek, L. Bauer, N. Christin, L. Cranor, and S. Egelman. Of passwords and people: Measuring the effect of password-composition policies. In *ACM CHI '11*, 2011.
20. S. Kossman. Familiar fraud: When family and friends steal your identity, 2014. http://www.creditcards.com/credit-card-news/familiar_fraud-damage-1282.php.
21. LastPass Inc. LastPass. https://lastpass.com/.
22. LeakedSource. LeakedSource analysis of MySpace.com hack, 2013. https://www.leakedsource.com/blog/myspace.
23. S. Lee and S. Zhai. The performance of touch screen soft buttons. In *CHI '09*, 2009.
24. Z. Li, W. He, D. Akhawe, and D. Song. The emperor's new password manager: Security analysis of web-based password managers. In *USENIX Security '14*, 2014.
25. R. C. Ltd. USB password manager: When your password database is right where you need it. http://www.anypassword.com/password-database-in-usb-password-manager.html.
26. S. Maydebura, D. Jeong, and B. Yu. Understanding environmental influences on performing password-based mobile authentication. In *IRI '13*, 2013.
27. Mozilla. Firefox. https://www.mozilla.org/en-US/firefox.
28. P. Owusu-Agyeman, W. Xie, and Y. Yeboah. A robust alternative virtual key input scheme for virtual keyboard systems. *Journal of Computer and Communications*, 2016.
29. Y. Park, S. Han, J. Park, and Y. Cho. Touch key design for target selection on a mobile phone. In *MobileHCI '08*, 2008.
30. B. Pellin. KeePassDroid. http://www.keepassdroid.com/.
31. F. Schaub, R. Deyhle, and M. Weber. Password entry usability and shoulder surfing susceptibility on different smartphone platforms. In *MUM '12*, 2012.
32. R. Shay, S. Komanduri, A. Durity, P. Huh, M. Mazurek, S. Segreti, B. Ur, L. Bauer, N. Christin, and L. Cranor. Can long passwords be secure and usable? In *ACM CHI '14*, 2014.
33. M. Siegler. One of the 32 million with a RockYou account? You may want to change all your passwords. Like now. *TechCrunch, http://techcrunch.com/2009/12/14/rockyou-hacked*, 2009.
34. J. Stroup. Who Commits Identity Theft?, 2016. https://www.thebalance.com/who-commits-identity-theft-1947637.

35. M. Tabini. Review: Lastpass takes your passwords to the cloud, 2013. http://www.macworld.com/article/2032046/review-lastpass-takes-your-passwords-to-the-cloud.html.
36. D. Wang, Z. Zhang, P. Wang, J. Yan, and X. Huang. Targeted online password guessing: An underestimated threat. In *ACM CCS '16*, 2016.
37. S. Zhai, M. Hunter, and B. A. Smith. The metropolis keyboard - An exploration of quantitative techniques for virtual keyboard design. In *UIST '00*, 2000.