Targeted Password Guessing Using *k*-Nearest Neighbors

Zhen Li, Ding Wang
College of Cryptology and Cyber Science, Nankai University, Tianjin 300350, China; wangding@nankai.edu.cn
Key Laboratory of Data and Intelligent System Security (NKU), Ministry of Education, Tianjin 300350, China
NDST, TBI Center, Nankai University, Tianjin 300350, China

***Abstract*—As the number of users' password accounts are constantly increasing, users are more and more inclined to reuse passwords. Recently, considerable efforts have been made to construct targeted password guessing models to characterize users' password reuse behaviors. However, existing studies mainly focus on characterizing slight modifications by training only on similar password pairs (e.g., `Shark0301` $\rightarrow$ `shark03`). This leads to overfitting and causes existing models to overlook users' large modification behaviors (e.g., `Shark0301` $\rightarrow$ `Bear03`). To fill this gap, this paper introduces a new non-parametric method named *k*-nearest-neighbors targeted password guessing (KNN-TPG). KNN-TPG builds a datastore that retains the context vector of all source passwords along with prefixes of the targeted passwords. During the generation of a new password, KNN-TPG retrieves *k* nearest neighbor vectors from the datastore to ensure that the generated passwords align better with realistic password distributions. By creatively combining KNN-TPG with our proposed Transformer-based password model, we propose a new targeted password guessing model, namely KNNGuess. At each step of generating a new password, KNNGuess predicts and utilizes three distinct distributions, aiming to comprehensively model users' password reuse behaviors.**

We demonstrate the effectiveness of our KNNGuess model and the KNN-TPG method through extensive experiments, which include 12 large-scale real-world password datasets, containing 4.8 billion passwords. More specifically, when the victim's password at site A is compromised (namely pw_A), within 100 guesses, the cracking success rate of KNNGuess for guessing her password at site B (namely pw_B, and $pw_B \neq pw_A$) is 25.40% (for common users) and 10.26% (for security-savvy users), which is 8.52%-119.0% (avg. 55.33%) higher than its foremost counterparts. When comparing with state-of-the-art password models (i.e., Pass2Edit and PointerGuess), this value is 8.52%-27.66% (avg. 18.09%) higher. Our results highlight that the threat of password tweaking attacks is higher than users expected.

I. Introduction

Text passwords are currently the most commonly used authentication method. Despite various alternative authentication methods being proposed continually (e.g., multi-factor authentication [1], [2] and hardware security key [3]), passwords have remained the primary method for user authentication [4], [5] due to their simplicity to use, easiness to change, and low deployment costs [6], [7].

Recent research indicates that 21%-33% of users tend to slightly modify their existing passwords when creating new ones, a practice frequently influenced by website password policies (e.g., minimum length and character composition requirements [8]–[11]). Moreover, 20%-59% of users directly reuse their existing passwords [12]–[14]. These behaviors represent two vulnerable password reuse patterns: indirect reuse (minor modifications) and direct reuse (exact repetition). Despite long-standing guidelines advising users to create passwords of varying strengths for accounts with different security levels [15]–[17], research consistently shows that users struggle to avoid password reuse [18], [19].

In reality, password reuse poses a significant threat to account security. Once attackers obtain passwords leaked from one service, they can exploit them to compromise the victim's other accounts through credential stuffing attacks [20], [21]. Such attacks result in severe economic and security consequences. For example, the retail industry alone suffers annual losses of approximately \$6 billion [22]. Recent large-scale breaches, including CAM4 (10.8 billion credentials) [23] and MOAB (26 billion credentials) [24], have provided attackers with ample material for credential stuffing campaigns. Worse still, recent research shows that when attackers exploit leaked passwords to capture users' *indirect* password reuse behavior, it poses a more severe threat [13], [25], [26]. These findings highlight the importance of modeling users' password reuse behavior to defend against attackers.

A. Motivations and design challenges

Considerable research [13], [25]–[28] has focused on developing targeted password guessing models to characterize users' password reuse behaviors and associated risks. Yet, most targeted password guessing models (e.g., Pass2Edit [25] and PointerGuess [29]) mainly capture two types of reuse behaviors: minor password modifications (Type-1) and the use of popular passwords (Type-2), while largely overlooking more subtle behaviors (Type-3) involving semantic similarity. Although Type-3 password pairs may appear dissimilar in character composition, they share intrinsic semantic connections, making them equally vulnerable to targeted attackers. Here, we describe the three reuse behaviors in more detail:

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.232077
www.ndss-symposium.org

- **Type-1**: The user makes simple or moderate changes to the source password, and the new password is similar in structure to the source password (e.g., `Shark0301` $\rightarrow$ `shark03`). This type of modification behavior can usually be captured explicitly by sequence-based metrics, such as cosine similarity>0.3 [25] or edit distance<5 [13], [26] between new and old password pairs.
- **Type-2**: The user chooses to use a popular password (e.g., `Shark0301` $\rightarrow$ `loveu4ever`). Such popular passwords often appear in publicly leaked dictionaries and are therefore insecure [25], [28].
- **Type-3**: The user creates a new password based on a partial pattern of the source password. While the two passwords may appear "dissimilar", they could be intrinsically similar in semantics and vulnerable to attackers (e.g., `Shark0301` $\rightarrow$ `Bear11`).

Several studies [13], [30], [31] have demonstrated that similar passwords share fine-grained features (e.g., semantic patterns) and have close distances (e.g., Euclidean distance) in latent space. While these findings focus on similar passwords, we observe that even dissimilar password pairs can show similar transformation patterns in latent space embeddings. For example, directly generating the target password `Bear11` from `Shark0301` can be challenging due to their substantial differences in character composition. However, the similar transformation patterns (e.g., `Shark0301` $\rightarrow$ `Shark03` and `Bear0311` $\rightarrow$ `Bear11`) project two source passwords closer in the latent space and link their vector representations. Consequently, during generation, the model can leverage this proximity to increase the likelihood of generating `Bear11`.

To capture semantic similarity in Type-3 reuse behavior, we introduce k-nearest-neighbors targeted password guessing (KNN-TPG), a novel non-parametric password guessing method based on KNN search [32]. KNN-TPG constructs a datastore that stores the context vectors of source passwords along with the prefixes of their corresponding target passwords. It learns to map a source password and the prefix of a target password to the next correct character in the latent space. When two passwords follow similar transformation patterns (i.e., users modify their original passwords in similar ways), their mappings tend to be similar. Consequently, even if the source and target passwords appear different at the character level, the vector representing the source password in the latent space will have a close distance. During password generation, this process is guided by retrieving information about the k nearest neighbors from the datastore, thereby facilitating the guessing of Type-3 password pairs.

By creatively integrating KNN-TPG with our proposed Transformer-based character-level model, we develop KNNGuess, which effectively captures both Type-3 and Type-1 behaviors. Moreover, to model users' Type-2 behaviors, we further propose a new method for mixing popular password lists. This method combines the guess list generated by KNNGuess with an external list of popular passwords to produce the final guess list. Particularly, within 100 guesses, the guessing success rates of our KNNGuess model outperform its foremost counterparts (i.e., Pass2Edit [25], PointerGuess [29], PassBERT [26], Pass2Path [13] and TarGuess-II [28]) by an average of 55.33%.

B. Our contributions

The contributions of this work are as follows:

- ***k*-nearest-neighbors targeted password guessing**. We propose the k-nearest-neighbors targeted password guessing (KNN-TPG) method, and integrate it with our Transformer-based character-level model to design KNNGuess, a novel targeted password guessing model that, *for the first time*, effectively captures all three types of password reuse behaviors (i.e., minor modifications, popular passwords, and semantic similarity). We further introduce a new popular-password mixing method to enhance overall cracking success rates.
- **Extensive evaluation**. We design 13 practical attack scenarios on 12 large real-world password datasets and conduct extensive experiments to demonstrate the effectiveness of our KNN-TPG method and the superior guessing success rate of our KNNGuess model. Results show that KNNGuess achieves an average improvement of 55.33% within 100 guesses over its foremost counterparts (excluding identical pairs) and outperforms state-of-the-art models (i.e., Pass2Edit [25], PointerGuess [29]) by 18.09%. Furthermore, incorporating KNN-TPG into the base model TransGuess boosts its cracking success rate by an additional 33.94% on average.
- **Some insights**. We analyze the stability of different models' guessing success rates as the training set size varies. Results show that, without adjusting the original hyperparameters, all models except KNNGuess exhibit the phenomenon that increasing the number of training password pairs leads to near-saturation in the model's effectiveness. This robustness of KNNGuess is attributed to the integration of our KNN-TPG method, which effectively mitigates this issue, consequently achieving a significantly later saturation point than its counterparts.

II. Background and Related Work

A. Password reuse behavior

In recent years, the number of password accounts users have to maintain continues to grow with the increase of internet service providers. Research findings show that the average user has 80-107 distinct online accounts [19], [33], [34]. As a result, users are likely to reuse existing passwords across different services. This behavior reduces the independence of accounts and increases users' security risks, making it commonly perceived as insecure. In 2014, Das et al. [27] conducted a survey revealing that approximately 43% of users reuse the same password across different services, and they summarized eight transformation rules to model reuse behavior. Subsequent studies [13], [25], [28], [35] have approached the issue from the attackers' perspective, aiming to model users' password reuse behavior by constructing more effective/threatening password guessing models. Some

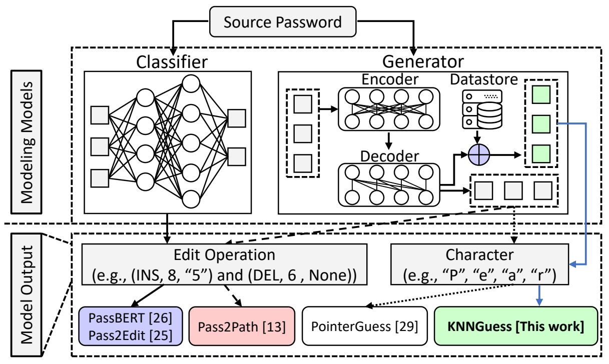


Fig. 1. Different modeling paradigms for existing-password-based targeted guessing models can be divided into three categories.

other studies [36]–[39], from defenders' perspective, detect users' reuse behavior to prevent credential stuffing attacks and analyze the relationship between passwords and services.

B. Password probability modeling

Classifiers and generators are commonly used in password guessability modeling and generation. The classifier is often associated with editing operations. By performing multiple classification tasks on a set of editing operations, it identifies several editing operations with the highest probabilities. These operations are then applied to an old password to produce a new password. The generator can either generate password characters or a sequence of editing operations. As shown in Fig. 1, there are three common modeling methods: The first category uses a classifier to output a sequence of editing operations (e.g., Pass2Edit [25], PassBERT [26]). The second category uses a generator to produce a sequence of editing operations (e.g., Pass2Path [13]). The third category uses a generator to directly generate password characters [29]. Our KNNGuess introduces offline datastore based on the third category. Character-based models model users' password reuse behavior as a series of estimates for conditional probability distributions. For instance, the probability of generating the password $pw = (c_0, c_1, ..., c_N)$ is shown below:

$$P(pw) = \prod_{i=1}^{N} P(c_i | c_0, ..., c_{i-1}), \tag{1}$$

where N is the total length, and c_i denotes the i-th character of pw. The probability of generating the password pw is a product of a series of conditional probabilities. Furthermore, when analyzing the behaviors of users reusing passwords, we can express the likelihood of a user creating a password $pw_B = (\widetilde{c}_0, ..., \widetilde{c}_N)$ for website B, based on the password $pw_A = (c_0, ..., c_M)$ used on website A as a conditional probability as follows:

$$\begin{aligned} P(pw_B | pw_A) &= P(\widetilde{c}_0, ..., \widetilde{c}_N | c_0, ..., c_M) \\ &= \prod_{i=1}^{N} P(\widetilde{c}_i | pw_A, \widetilde{c}_0, ..., \widetilde{c}_{i-1}). \end{aligned} \tag{2}$$

C. Related work

At CCS'16, Wang et al. [28] proposed a password reuse model called TarGuess-II, based on Probabilistic Context-Free Grammars (PCFG) [40] and the Markov model [41]. Its core idea is to describe users' password reuse behavior based on two levels of modification operations, namely structure-level and segment-level. In the structure-level process, TarGuess-II primarily considers the insertion and deletion operations at the beginning and end of different structures (i.e., letter segment L, digital segment D, and special character segment S). In the segment-level process, it considers the insertion and deletion operations within the same segment. Additionally, it heuristically mixes popular passwords to model the user's behavior of using popular passwords, significantly enhancing its cracking effectiveness.

At IEEE S&P'19, Pal et al. [13] introduced deep learning techniques to model user reuse behavior, namely Pass2Path. They trained a sequence-to-sequence (Seq2Seq [42]) model to predict the modification operations needed to transform one password to another. This approach results in higher cracking success rates, as it allows the model to focus better on users' typical password transformations.

At USENIX SEC'23, Wang et al. [25] for the first time introduced a multi-step mechanism, proposing a new targeted guessing model called Pass2Edit. They modeled the password reuse process as a multi-classification problem, establishing a direct connection between the editing operations and their corresponding editing effects, resulting in better cracking success rate than previous models. At the same time, Xu et al. [26] proposed a pre-trained model named PassBERT based on bidirectional Transformer. They used the pre-training and fine-tuning paradigm to model users' password reuse behavior. Through the sequence labeling mechanism [43], PassBERT can predict an editing operation for each character, thereby forming a complete editing path.

At USENIX SEC'24, Xiu-Wang [29] introduced the pointer mechanism to simulate users' reuse behaviors and proposed a new model, namely PointerGuess. PointerGuess defines password reuse from both individual and population-wide perspectives, thereby it can capture the complex password modification behaviors of users.

Our differences. Our work is essentially different from the above-mentioned in the following two key aspects.

Model architecture. We notice that only PassBERT [26] and our work use Transformers. The main difference is that: PassBERT's core architecture relies on Transformers, while in our work, Transformers is used only as the base model in combination with the KNN-TPG method. We show that only using Transformers doesn't lead to satisfactory results (see TransGuess's results in Fig. 4). Particularly, our base model can also be other Encoder-Decoder architecture, including RNN, LSTM, or GRU.

Modeling ideas. The underlying idea of KNN-TPG is that passwords with similar transformation patterns have related representation vectors in the latent space. In contrast, Pass2Edit [25] and Pass2Path [13] focus only on characterizing the editing sequence of password pairs. They differ significantly in both the information retrieval approach and the generation process.

III. KNNGuess: A targeted password guessing model combined with KNN-TPG

In this section, we first introduce our Transformer-based password guessing model TransGuess, which serves as the base model. Then, we introduce our proposed non-parametric method KNN-TPG. We combine it with the base model TransGuess to create the final targeted password guessing model, KNNGuess, to model users' password reuse behaviors.

A. Base password model of KNNGuess

Transformers [44], a prominent deep learning model for sequence-to-sequence tasks in the field of Natural Language Processing (NLP), has been demonstrated to have strong fitting ability. The structure of the Transformer includes an encoder and a decoder. Each of them consists of multiple layers, with each layer containing a self-attention mechanism and a feed forward neural network. Inputs are processed by these layers to produce the final output. Due to its powerful text learning capabilities, we employ the Transformer as our base model for targeted password guessing, namely TransGuess. TransGuess is a sequence-to-sequence neural password guessing model, it takes a leaked password from a user as input and outputs password variations to capture users' Type-1 behaviors.

Model architecture. TransGuess utilizes an encoder-decoder-based architecture. The input password is mapped to high-dimensional vector space $v \in \mathbb{R}^d$ through the embedding layer. The hyperparameterized dimension d is maintained consistently across all sub-layers of the model using residual connections. The encoder generates a sequence of continuous representations z from v, which includes contextual information among the input password characters. After encoding, the Decoder performs autoregressive decoding based on the encoding result z and the generated results at each step.

Additional configurations of TransGuess. We align the hyperparameters in the TransGuess model with those in the Transformer [44] to ensure the base model is simple and extensible (i.e., the dimension of the vector is 512, and the encoder and decoder each have six identical layers.). Pal et al. [13] demonstrated that using the key-sequence mechanism can capture the capitalization-related transformations better and improve the generation of special characters in passwords. We incorporate the consideration of shift key and caps-lock on the keyboard when processing password sequences. Specifically, we convert each password into a sequence of key-presses (e.g., `WANG123!` is converted to ⟨caps⟩`wang123`⟨shift⟩1). Hence, we exclude upper-case letters and some special characters from the vocabulary, as they can be formed by other tokens. The vocabulary, denoted as Σ, has a size of 54, comprises 48 types of characters on the EN-US standard keyboard, and six special identifiers: EOS, BOS, PAD, UNK, as well as ⟨caps⟩ and ⟨shift⟩. EOS stands for end-of-sequence, indicating the end of a password. BOS, or begin-of-sequence, marks the start of a password. PAD denotes padding, used to fill in spaces and ensure passwords input into TransGuess have the same length. UNK means "unknown token", which is a special symbol used to represent characters models don't recognize during password processing. We employ Adam as the model optimizer with a learning rate of 5×10^{-4} and a dropout rate of 0.2 to alleviate overfitting. Typically, we set the training epochs to 40 to ensure model convergence.

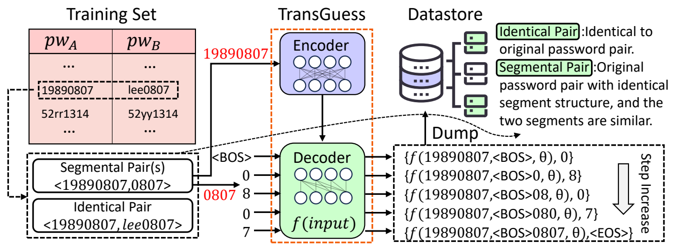

Training Set
pw_A
pw_B
...
...
19890807
lee0807
52rr1314
52yy1314
TransGuess
Encoder
Decoder
$f(input)$
Datastore
Identical Pair:Identical to original password pair.
Segmental Pair:Original password pair with identical segment structure, and the two segments are similar.
Dump
19890807
0807
Segmental Pair(s) <19890807,0807>
Identical Pair <19890807,lee0807>
<BOS>
0
8
0
7
{f(19890807,<BOS>, θ), 0}
{f(19890807,<BOS>0, θ), 8}
{f(19890807,<BOS>08, θ), 0}
{f(19890807,<BOS>080, θ), 7}
{f(19890807,<BOS>0807, θ),<EOS>}
Step Increase

Fig. 2. The process of building the datastore involves generating two types of password pairs: Segmental Pair(s) and Identical Pair from original password pairs in the training set. These pairs are input into the TransGuess model to produce a series of key-value pairs that form the datastore.

B. Non-parametric method: KNN-TPG

We introduce a non-parametric method, called k-nearest-neighbor targeted password guessing (KNN-TPG), which can be used in any trained neural password guessing models without additional training. It builds a large datastore from the training set. When generating a new password, we retrieve k-nearest neighbors from this datastore to provide additional information beyond the base model TransGuess. This additional information helps guide the generation of Type-3 passwords. Initially, we train TransGuess on the training dataset to achieve convergence of its loss. This ensures that our TransGuess has the ability to capture users' Type-1 reuse behaviors and map inputs to a high-dimensional vector space. When using the user's password $pw_A = (c_0, ...c_M)$ to generate the variant $pw_B = (\widetilde{c}_0, ..., \widetilde{c}_N)$, in each step of the generation process, the decoder of TransGuess predicts the conditional probability distribution for the next token, denoted as $P_{Basic}(\widetilde{c}_i | pw_A, \widetilde{c}_{<i}, \theta)$, conditioned on the leaked password pw_A, previously generated targeted tokens $\widetilde{c}_{<i}$, and the parameters θ of TransGuess.

Building datastore. A pair of passwords in the training set generates two types of inputs: "Segmental Pair(s)" and "Identical Pair". The Identical Pair is exactly the same as the original password pair in the training set. They include all dissimilar passwords, helping the model learn the latent semantic relationships between Type-3 type password pairs in a high-dimensional space. The Segmental Pair(s) consists of two similar segments within the original password pair. They strengthen TransGuess's ability to represent Type-1 reuse behavior. Both segments share the same structure-level tags (e.g., letter segment L, digital segment D, and special character segment S). Therefore, Segmental Pair(s) may consist of one or more pairs of password segments. Wang et al. [25] showed that using cosine similarity instead of edit distance provides a more accurate metric of similarity between passwords. Therefore, we use $sim(Seg_A, Seg_B) > 0.3$ to determine the similarity between two password segments, Seg_A and Seg_B. The threshold of 0.3 is based on experimental findings by Wang et al. [25]. The calculation method of similarity sim is as follows:

$$sim(Seg_A, Seg_B) = \frac{\sum\limits_{g\in\mathbb{G}} \left(cnt\left(Seg_A, g\right) * cnt\left(Seg_B, g\right)\right)}{\sqrt{\sum\limits_{g\in\mathbb{G}} cnt^2(Seg_A, g)}\ \sqrt{\sum\limits_{g\in\mathbb{G}} cnt^2(Seg_B, g)}},$$

where $\mathbb{G}$ denotes the collection of all 2-gram substrings within Seg_A and Seg_B. The function $cnt(Seg, g)$ signifies the frequency of the substring g in the password segment Seg. Both Identical Pair (X_I,Y_I) and Segmental Pair(s) (X_S,Y_S) will serve as inputs to build the datastore. In the password pairs, the source password is input into the Encoder of TransGuess, while the targeted password is sequentially input into the decoder. At each decoding step, a key-value pair is generated. The datastore contains a set of key-value pairs. TransGuess generates the key, a high-dimensional representation vector, through an input password and previously generated password tokens, denoted as $f(pw_A, \widetilde{c}_{<i}, \theta)$, where f denotes a mapping of the decoder that transforms the input into high-dimensional representation. We utilize the hidden layer vector from the last layer of the TransGuess decoder as the representation for the function f. The value corresponds to the ground truth $\widetilde{c}_i$, which represents each character in the targeted password. Formally, the entire datastore can be described as follows:

$$\{(Keys, Values)\} = \{(f(pw_A, \widetilde{c}_{<i}, \theta), \widetilde{c}_i), \quad \forall \widetilde{c}_i \in pw_B | (pw_A, pw_B) \in (X_I, Y_I) \cup (X_S, Y_S)\}. \tag{3}$$

During datastore construction, we process the training set through TransGuess in batches to generate 128×512-dimensional representation vectors (keys). These pair with the prefix of target password tensors (values) at each decoding step. We filter out padding symbols (values $\leq$1) before storing valid key-value pairs. The decoding continues until reaching maximum length or all-padding states, as formalized in Algo. 1. In Appendix B, we share some engineering tips to quickly implement KNN-TPG beyond the core algorithm.

As shown in Fig. 2, consider the password pair of pw_A =`19890807` and pw_B =`lee0807` in training set as an example. The original password pair generates Segmental Pair(s) $<$ `19890807`, `0807` $>$ because the cosine similarity score, sim(`19890807`,`0807`)=0.596>0.3. Besides, both `19890807` and `0807` are digital segments D. At each decoding step, a key-value pair is generated. The value consists of all the characters in the targeted password, namely: `0`, `8`, `0`, `7`, and the end-of-sequence symbol EOS. All key-value pairs are dumped into the datastore for subsequent retrieval processes.

Generating passwords. When generating reused passwords, we first input source password pw_A into the encoder of TransGuess. At each decoding step, TransGuess first outputs a conditional probability distribution, denoted as $P_{Basic}(\widetilde{c}_i|pw_A, \widetilde{c}_{<i}, \theta)$, referred to as the Basic Distribution, for the targeted password character $\widetilde{c}_i$. Moreover, TransGuess outputs a high-dimensional representation vector called Basic Decoder Representation, which is used as a query to retrieve the k nearest neighbors from the datastore, according to L^2 distance between the query and key. This allows obtaining k distances, denoted as $d = (d_1, ..., d_k)$. By employing the

Algorithm 1: Generate key-value pair algorithm.

```
Input: Datastore training set:
    $\mathcal{X} = \{(pw_A^0, pw_B^0), (pw_A^1, pw_B^1), ..., (pw_A^m, pw_B^m)\}$.
Output: Datastore with key-value pairs ($\mathcal{D}$).
1  $\mathcal{M} \leftarrow Trained\ TransGuess\ model$;
2  $K \leftarrow Decode\ max\ len$;
3  for $batch$ in $\mathcal{X}$ do
4      $memory \leftarrow \mathcal{M}.encode(batch)$;/* memory is the context
         vector generated by the encoder of TransGuess. */
5      $targeted \leftarrow batch.target$;
6      $tgt \leftarrow [BOS_1, ..., BOS_n]^T$;/* Initialize decoding password
         tensor, starting from begin-of-sequence. */
7      for $decode\_step\ i \leftarrow 1$ to $K$ do
8          $out \leftarrow \mathcal{M}.decode(tgt, memory)$;
9          $key \leftarrow out[:, -1]$;/* Use the last vector of the hidden
             layer as the key. */
10         $value \leftarrow targeted[:, i]$;
11         $mask \leftarrow value > 1$;
12         $tgt \leftarrow concat(tgt, value)$;
13         $key, value \leftarrow key[mask], value[mask]$;/* Exclude
             key-value pairs that have been generated. */
14         $\mathcal{D}.add(key, value)$;
15 return $\mathcal{D}$
```

following three operations: Temperature, Normalization and Aggregation, we ultimately obtain a conditional probability distribution for predicting the next token generated by the retrieval distance d. This distribution is referred to as the KNN Distribution, denoted as $P_{KNN}(\widetilde{c}_i|pw_A, \widetilde{c}_{<i}, d)$.

The Temperature operation involves dividing all the distances $(d_1, ..., d_k)$ by a certain value T, resulting in the new distances $d_T = d/T$. It adjusts the sensitivity and intensity of the distances. Selecting a large T helps reduce differences between various distances. This assists in preventing overfitting to similar retrieval results, avoiding excessive focus on a few items, and encouraging a more balanced distribution. The Normalization operation scales distance to the standard range of [0,1]. Specifically, we apply the softmax function to $-d_T$ because smaller distances indicate higher probabilities for the corresponding keys. The operation involves exponentiating each distance and normalizing the result, ultimately obtaining the probability distribution for each distance representing value, denoted as $P(value) \propto exp(-d_T)$. The Aggregation operation combines values corresponding to different distances by adding the probabilities of the same value. This results in the probability distribution generated by retrieval distances.

When users modify passwords based on their existing ones, there are instances where they need not consider the entire password but only focus on a specific segment within the passwords. We refer to this user's modification as users' "segmental attention behavior". For instance, whether the old password is `Wang123` or `lee123`, there is a high likelihood that users, in the practice of password reuse, might generate `wang123456` and `lee123456`. The transformation, in this case, is independent of the characters preceding the password segment `123`, but significantly influenced by the segment `123` itself. To model such observed reuse behaviors, we introduce a masking mechanism that simultaneously inputs the decoded characters $\widetilde{c}_{j:i-1}$ into the Encoder of TransGuess at each decoding step. During this input process, we mask out all

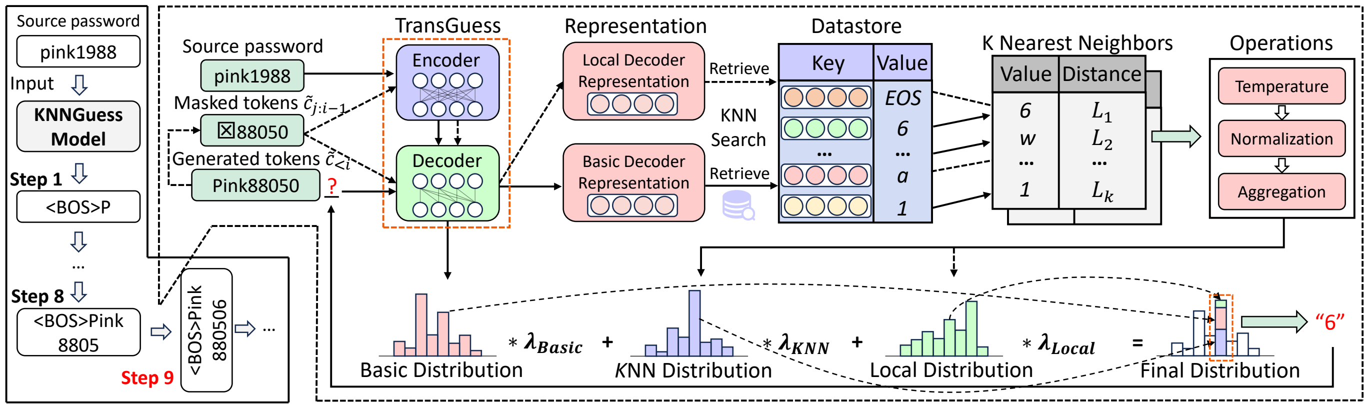


Fig. 3. An example of decoding at the 9th step (i=9) during password guess generation. Here, we suppose the source password is `pink1988` and the decoded generated password is `Pink88050`. The Final Distribution for predicting the next character is obtained through interpolation of three distributions. TransGuess generates the Basic Distribution and a high-dimensional vector called Basic Decoder Representation from the source password. By retrieving the k nearest neighbors from the datastore and applying three operations, it obtains the KNN Distribution. The solid line in this figure illustrates the generation process. The last segment of the generated tokens serves as the masked tokens, which are input into TransGuess to get Local Decoder Representation. Similarly, by retrieving k nearest neighbors from the datastore, the Local Distribution is obtained. The dashed line illustrates the process of generating Local Distribution.

characters before the segment $\widetilde{c}_{0:j-1}$ that is currently under generation. Therefore, all characters before position j are masked because $\widetilde{c}_{j-1}$ and $\widetilde{c}_j$ belong to different structure-level tags(e.g., $\widetilde{c}_{j-1}$ is letter and $\widetilde{c}_j$ is digit).

In Fig. 3, the input is source password `pink1988`. In the subsequent decoding step after generating `Pink88050`, the segment `88050` obtained by masking the generated tokens is input into both Encoder and Decoder. Thus, TransGuess also produces a representation vector, named Local Decoder Representation. It is derived from the masked tokens `88050`, which like the Base Decoder Representation, is used as a query to retrieve the k nearest neighbors from the datastore. After undergoing Temperature, Normalization, and Aggregation operations, the model yields a conditional probability distribution generated from local information, denoted as $P_{Local}(\widetilde{c}_i|\widetilde{c}_{j:i-1}, d)$, referred to as the Local Distribution.

We have three distinct distributions: Basic Distribution, denoted as $P_{Basic}(\widetilde{c}_i|pw_A, \widetilde{c}_{<i}, \theta)$, KNN Distribution, denoted as $P_{KNN}(\widetilde{c}_i|pw_A, \widetilde{c}_{<i}, d)$ and Local Distribution, denoted as $P_{Local}(\widetilde{c}_i|\widetilde{c}_{j:i-1}, d)$ for predicting the next character. The Basic Distribution and Local Distribution both capture users' Type-1 reuse behaviors, while the KNN Distribution targets Type-3 reuse. We use three parameters λ_{Basic}, λ_{KNN} and λ_{Local}, where the sum of these parameters is 1, to adjust the proportions of each distribution. The final decoding result is interpolated with these three distributions and three parameters. Consequently, we obtain the Final Distribution, that is:

$$\begin{aligned} P(\widetilde{c}_i|pw_A, \widetilde{c}_{<i}) = & P_{Basic}(\widetilde{c}_i|pw_A, \widetilde{c}_{<i}, \theta) \cdot \lambda_{Basic} \\ & + P_{KNN}(\widetilde{c}_i|pw_A, \widetilde{c}_{<i}, d) \cdot \lambda_{KNN} \\ & + P_{Local}(\widetilde{c}_i|\widetilde{c}_{j:i-1}, d) \cdot \lambda_{Local}. \end{aligned} \tag{4}$$

As shown in Fig. 3, after obtaining the Final Distribution for predicting the next token, the character '6' with the highest probability is selected and returned to the generated tokens. In the next decoding step, the generated tokens are `Pink880506` and the masked tokens are `880506`. The decoding process continues until the EOS symbol is generated.

For each input password, we need to generate q password guesses in decreasing order of likelihood. Thus, at each decoding step, we employ the beam search decoding algorithm [45] to obtain multiple password guesses. There is a parameter called beam width (w) in the beam search algorithm, which signifies retaining the top w sequences at each time step. Specifically, during each decoding step, w sequences are selected from $w{*}w$ sequences. Among these, sequences ending with EOS are considered as one of the potential guesses. Finally, we sort the results in descending order of probability to get the final outputs. To get at least 1,000 password guesses, we set the beam width to 1,000 in our model.

C. Mixing popular passwords

Unlike previous works that empirically mix popular passwords into the model-generated list (e.g., TarGuess-II [28], Pass2Edit [25], assuming that the probability of choosing a popular password is equal among all users while ignoring real-world user heterogeneity. To address the limitations in prior work, we propose a dynamic popular password interpolation method that depends on the source password to capture the users' vulnerable behaviors of using popular passwords, which will mix popular password list $List_p$ with the guess list $List_g$ generated by KNNGuess. First, we need to build this list. To ensure a fair comparison with previous work (e.g., Pass2Edit [25] and TarGuess-II [28]), we use the same list of popular passwords as before. Specifically, for the Chinese dataset, the popular password list is $\mathcal{L}_C$={pw|the value of $P_{\text{csdn}}(pw) * P_{126}(pw) * P_{\text{Dodonew}}(pw)$ ranks top-10^4}, while for the English dataset, it is $\mathcal{L}_E$={pw|the value of $P_{\text{000Webhost}}(pw) * P_{\text{Yahoo}}(pw) * P_{\text{LinkedIn}}(pw)$ ranks top-10^4}. In fact, using different popular lists for different datasets can increase the cracking success rate. The probability that password pw_S generates a new password pw_A is as follows:

$$\begin{aligned} P(pw_A|pw_S) = & Norm\{P_{Model}(pw_A|pw_S) * \beta\} * (1 - \alpha_{pop}) \\ & + Norm\{P_{Popular}(pw_A|pw_S) * \beta\} * \alpha_{pop}. \end{aligned} \tag{5}$$

We express the probability of generating a new password as two components: the probability of the user modifying the original password (including Type-1 and Type-3) and the probability of the user choosing a popular password (i.e., Type-2). The final probability is obtained through linear interpolation with a factor α_{pop}. The value of α_{pop} is calculated from the training set and represents the proportion of users who choose popular password. The interpolation factor $1-\alpha_{pop}$ represents the proportion of users who have Type-1 and Type-3 behaviors when constructing new passwords. P_{Model} denotes the probability of our KNNGuess generating the password pw_A using pw_S, and $P_{Popular}$ denotes the probability of pw_S becoming the popular password pw_A. Note that when generating popular passwords, we consider the influence of the original password pw_S. This means that different original passwords (e.g., `p@ssw0rd` and `summer0803`) may yield varying probabilities of generating the same popular password like `Passw0rd`. In previous works [25], [28], [29], this was treated as identical (i.e., $P_{Popular}(pw_A|pw_S) = P_{Popular}(pw_A)$). Finally, we normalize passwords probabilities in $List_p$ and $List_g$ to a common scale for summation.

To ensure that the probability of generating a popular password is influenced by the original password, we calculate the distance between the source password pw_S and each popular password $pw_{p_i} \in List_p$ within a high-dimensional space. This distance is multiplied by the probability of pw_{p_i} in $List_p$. After applying normalization, we obtain the final probability. To prevent small differences in distance from resulting in negligible differences in probability, we amplify the probability results by multiplying them with a parameter β. In this work, we set $\beta = 1,000$. We can express the $P_{Popular}(pw_A|pw_S)$ as: $P_{Popular}(pw_A|pw_S) = Distance_h(pw_A, pw_S) * List_p(pw_A)$.

IV. Experiments

We first elaborate on the experimental setup, including the password dataset used, the attack scenarios constructed, the hardware configuration and parameter selection. Then, we conduct a fair comparison of KNNGuess with its foremost counterparts (i.e., PointerGuess [29], Pass2Edit [25], Pass2Path [13], TarGuess-II [28] and PassBERT [26]).

A. Our datasets

Datasets. We conduct large-scale experiments on 12 password datasets containing 4.8 billion passwords to evaluate our KNNGuess model and its foremost counterparts. Our password datasets consist of 5 English datasets, 5 Chinese datasets, and two large-scale mixed leakage datasets (see Table I). These leaked password datasets were compromised by hackers and made publicly available on the Internet between 2011 and 2021, which are openly accessible. The password leakage datasets we utilized encompass websites of diverse service types (e.g., e-commerce, social forum, and email), to ensure the diversity of experimental scenarios and to test the robustness of the password guessing models. Both 4iQ [46] and COMB [47] are mixed password datasets, comprising leaked passwords from various websites of different countries and services. Various factors such as user types, language, service types, and service policies affect password strength. For example, 000Webhost users are primarily web administrators, thus the passwords created might be more secure than those created by common users. These leaked password datasets are utilized to construct password reuse attack scenarios by matching passwords that have the same corresponding email.
Datasets cleaning. For email addresses, we first removed items with empty email and those without the '@' symbol, as they can't match the same email address for the same user. For passwords, similar to previous works [13], [25], we only retained passwords containing 94 printable ASCII characters (from 33 to 126) and removed passwords with $lengths \geq 30$. These serve as initial cleaning strategies, with each leaked dataset in Table I undergoing the initial cleansing process. Subsequently, when constructing password reuse attack scenarios, specific cleaning methods will be applied based on the distinct website policies (see Table IV-B for details). For example, although the training sets for Scenarios #1&#2 are the same, we need to remove password pairs with len<8 from Tianya$\rightarrow$Dodonew in Scenario #2, as the password policy of the service CSDN to be guessed requires passwords with len$\geq$8 (while in Scenario #1, the password policy is none).
Open science. We will openly share our research artifacts to enhance the reproducibility and replicability of our work. We present our preliminary open-source solution in Sec. V-D.

B. Experimental setup

Attack scenarios. We consider modeling real-world tweaking attacks based on multiple passwords obtained from the same user through identical email matches. For different datasets, similar to the previous works [13], [25], if the same email appears in two datasets, we randomly select one of the passwords associated with this email as the source password pw_A and the other as the targeted password pw_B, forming an item $\langle pw_A, pw_B \rangle$ in the training set.

Previous research [48], [49] has shown that various real-world factors play a significant role in the characteristics and strength of passwords. To closely approximate the behavior of informed attackers in real-world scenarios, we constructed 13 meaningful attack scenarios, as shown in Table II. Specifically, Scenarios #1-#4 (Chinese) and Scenarios #5-#8 (English) compare same/different service types and strong/weak password policy transitions (e.g., CSDN's 8-character minimum). Scenarios #9-#10 test mixed attacks using combined datasets (e.g., LinkedIn+Twitter passwords to attack MathWay+000Webhost), while #11-#12 employ large leaked datasets (4iQ and COMB). Finally, scenario #13 specifically examines the cracking behavior on RedMart's MD5-salted hashes, with all other tests using plaintext passwords. See Table I for policy strength comparisons. In the training set, service B in A $\rightarrow$ B is known, aiming to enable the model to learn transformations from old passwords to new ones. However, in the test set, service C in A $\rightarrow$ C is unknown, representing a collection of passwords that attackers aim to crack through service A.

TABLE I
DETAILS OF PASSWORD DATASETS LEAKED FROM VARIOUS WEB SERVICES AND DATA CLEANSING ("PWS" STANDS FOR PASSWORDS).

Dataset	Language	Leaked Time	Original PWs	Unique PWs	Invalid emails	Invalid PWs	Removed %	After cleaning	Web service
LinkedIn	English	Jan. 2012	54,656,615	34,282,741	0	122,051	0.23%	54,534,564	Job hunting
000Webhost	English	Oct. 2015	15,299,907	10,526,769	49,061	67,401	0.76%	15,183,445	Web hosting
Twitter	English	May. 2016	25,575,929	16,249,287	3	287,548	1.12%	25,288,378	Social forum
RedMart[‡]	English	Oct. 2020	1,108,774	—	0	—	0	1,108,774	E-commerce
MathWay	English	Jan. 2020	16,051,087	10,054,873	168,819	40,907	1.31%	15,841,361	Education
126	Chinese	Dec. 2011	6,392,568	3,764,740	0	14,995	0.24%	6,377,573	Email
Tianya	Chinese	Dec. 2011	30,816,592	12,873,222	5,783	3,279	0.03%	30,807,530	Social forum
Dodonew	Chinese	Dec. 2011	16,282,286	10,010,744	225,931	30,085	1.57%	16,026,270	E-commerce & Gaming
Taobao	Chinese	Feb. 2016	15,072,418	11,633,759	1,176	90	0.01%	15,071,153	E-commerce
CSDN	Chinese	Dec. 2011	6,428,410	4,034,779	7	3,157	0.05%	6,425,246	Programmer forum
4iQ	Mixed	Dec. 2017	1,400,553,869	445,259,097	575,283	18,475,938	1.36%	1,381,502,648	Mixed
COMB	Mixed	Feb. 2021	3,279,064,312	855,833,811	81,542,117	15,718,941	2.97%	3,181,803,254	Mixed

[‡]RedMart, leaked from an online supermarket in Singapore, and all user passwords are in salted hash format. As we consider it as the real target, the data for statistical password characteristics is underscored (i.e.,"—" and the value of 0).

TABLE II
SETUPS OF 13 DIFFERENT ATTACKING SCENARIOS (FOR EVALUATION RESULTS, SEE FIG. 4, AND FIG. 5)[†]

Scenario	Language	Training set setup	Size (pairs)	Identical pairs	Test set setup	Size (pairs)	Identical pairs	Clean strategies*
1	Chinese	Tianya $\rightarrow$ Dodonew	624,925	28.71%	Tianya $\rightarrow$ Taobao	57,7017	26.87%	None
2		Tianya $\rightarrow$ Dodonew	434,255	23.33%	Tianya $\rightarrow$ CSDN	826,559	33.18%	$len{\geq}8$
3		126 $\rightarrow$ Dodonew	188,926	36.32%	126 $\rightarrow$ CSDN	86,104	31.55%	$len{\geq}8$
4		CSDN $\rightarrow$ Dodonew	211,385	24.21%	CSDN $\rightarrow$ 126	86,104	31.55%	None
5	English	000Webhost $\rightarrow$ Twitter	695,560	16.07%	000Webhost $\rightarrow$ LinkedIn	265,083	19.14%	$len{\geq}6$
6		LinkedIn $\rightarrow$ Twitter	944,451	34.26%	LinkedIn $\rightarrow$ MathWay	163,847	31.86%	$len{\geq}5$
7		Twitter $\rightarrow$ LinkedIn	316,388	34.83%	Twitter $\rightarrow$ 000Webhost	471,650	16.07%	LD, $len{\geq}6$
8		LinkedIn $\rightarrow$ Twitter	482,763	35.84%	LinkedIn $\rightarrow$ 000Webhost	259,175	19.55%	LD, $len{\geq}6$
9	Mixed	2 mixed English datasets	412,007	19.31%	2 mixed English Datasets	103,001	19.17%	None
10		3 mixed Chinese datasets	1,265,219	32.51%	2 mixed Chinese Datasets	316,304	31.28%	None
11		80% of 4iQ dataset	116,837,808	5.02%	20 % 4iQ dataset	29,209,452	4.94%	None
12		80% of COMB dataset	342,921,727	34.23%	20 % COMB dataset	85,730,432	34.44%	None
13 (hash)	English	000Webhost $\rightarrow$ Linkedin	213,697	19.26%	000Webhost $\rightarrow$ RedMart	6,858	16.70%	LD, $len{\geq}6$[‡]

[†]$A \rightarrow B$ means: the passwords leaked by users on the website A can be used by an attacker to attack the same user's account on the website B. Note that both the training and test sets contain identical password pairs. However, during the training and testing processes, we do not use identical password pairs(e.g., in attack scenario #1, the number of passwords inputted into the model for training is 624,925*(1-28.71%)=445,509).
*Clean strategies refer to additional cleaning strategies applied to password pairs in the training set, beyond the initial cleaning strategy(see Section IV-A). Different cleaning strategies can result in the same training set having different sizes (e.g., scenario #6 and scenario #8).
[‡](LD, $len{\geq}6$) means that we only retain passwords with a length greater than or equal to 6, and containing at least one letter and one digit.

TABLE III
CRACKING SUCCESS RATE CORRESPONDING TO DIFFERENT HYPERPARAMETERS.[†]

Parameter	Attack scenario #4: CSDN $\rightarrow$ 126							
λ_{Basic}[‡]	0.7	0.5	0.3	0.5	0.5	0.5	0.5	0.5
λ_{KNN}	0.2	0.4	0.6	0.4	0.4	0.4	0.4	0.4
k	32	32	32	8	16	48	32	32
T	300	300	300	300	300	300	10	1,000
Success rate	42.39%	**43.82%**	43.63%	43.49%	43.54%	43.63%	43.69%	43.67%

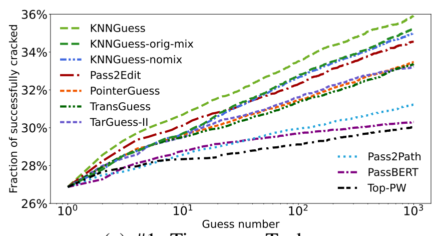
[†] We conduct tests under attack scenario #4, as detailed in Table II. The "Rate" represents the cracking success rate of the KNNGuess model under different parameter combinations, with 1,000 guessing attempts.
[‡] λ_{Basic} and λ_{KNN} represent the proportions of the basic distribution and KNN distribution in the final distribution respectively when predicting the next token. "k" denotes the number of retrievals from the datastore each time, and "T" represents the value of the Temperature operation, as detailed in Sec. III-B. The **bold** values indicate the highest cracking success rate. The hyperparameters we take achieve the optimal effect.

Hardware Configuration. To ensure the fairness of the experiment, we conducted training and testing on the same workstation to compare our KNNGuess model with its main competition models. Our workstation is equipped with the NVIDIA RTX 3090 GPU (including 24GB of VRAM), Intel Xeon Silver processor, 256GB of RAM, and a 4TB hard drive. We believe that such a configuration is not difficult to achieve for malicious attackers.

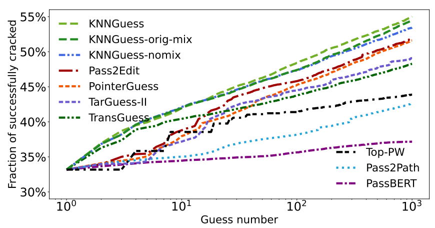
Key parameters selection. In KNNGuess, we find that the proportion hyperparameters λ_{Basic}, λ_{KNN} and λ_{Local} of three distributions within the final distribution will affect the model effectiveness. Through extensive experimentation, we determine $\lambda_{Basic} = 0.5$, $\lambda_{KNN} = 0.4$, $\lambda_{Local} = 0.1$. In theory, setting $\lambda_{Basic} = 0.5$ ensures the strong generalization ability of TransGuess, while $\lambda_{KNN} = 0.4$ allows KNN-TPG to learn the transformation methods for similar passwords in the latent space, and constrains the generated password by the distribution observed in the entire training set. Setting $\lambda_{Local} = 0.1$ models the users' segmental attention behavior (which means the users only focus on a certain segment of the passwords during modification). In Table III, we show experimental results for some different values to illustrate the rationality of our choice of parameters, and we discuss the reasons why different parameter values affect the effectiveness of KNNGuess in Appendix C.

Clearly, the effectiveness of KNN-TPG depends on two factors: the number of retrievals k from the datastore and the parameter T of the Temperature operation. We explore the impact of different parameter values on KNN-TPG and select the most suitable ones for targeted password guessing. We chose k=32 and T=300. Theoretically, choosing a large k may result in retrieving more noise, thereby reducing the effectiveness of the model, while selecting a small k may lead to retrieval results being biased towards a single outcome, thereby affecting the normal distribution of the next characters. For the parameter T in Temperature operation, the appropriate parameter can encourage a more balanced distribution. We present the reasons for choosing these values in Table III.

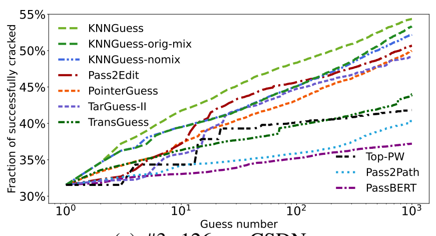
Guessing approaches for comparison. We compare KN-NGuess with its main counterparts (i.e., PointerGuess [29],



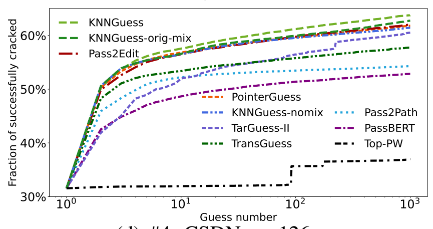
(a) #1: Tianya $\rightarrow$ Taobao



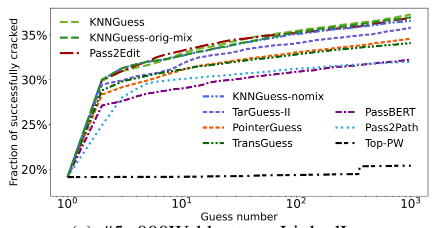
(b) #2: Tianya $\rightarrow$ CSDN



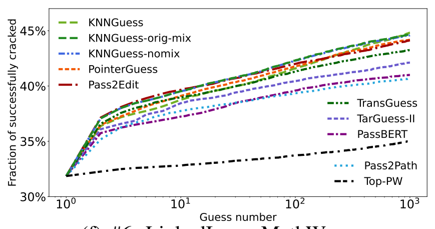
(c) #3: 126 $\rightarrow$ CSDN



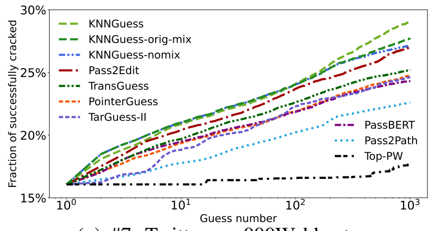
(d) #4: CSDN $\rightarrow$ 126



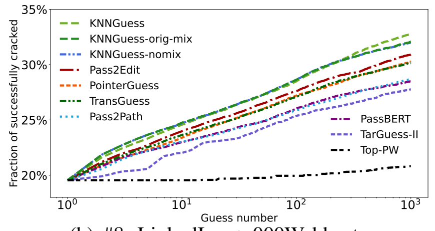
(e) #5: 000Webhost $\rightarrow$ LinkedIn



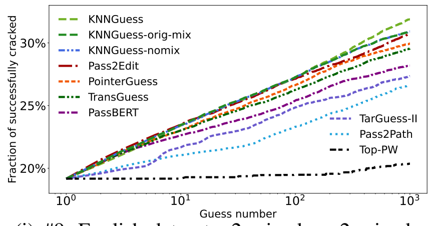
(f) #6: LinkedIn $\rightarrow$ MathWay



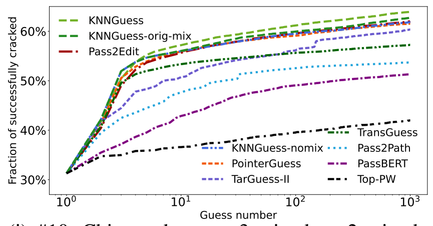
(g) #7: Twitter $\rightarrow$ 000Webhost



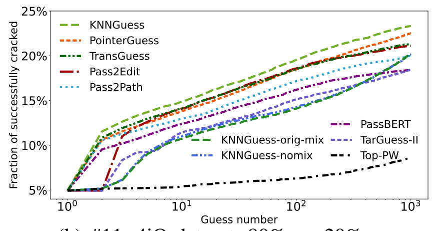
(h) #8: LinkedIn $\rightarrow$ 000Webhost



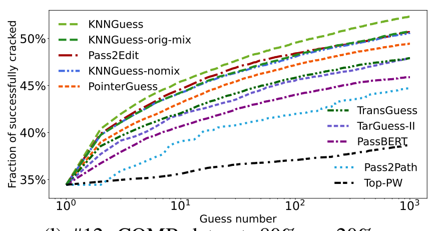
(i) #9: English datasets: 2 mixed $\rightarrow$ 2 mixed



(j) #10: Chinese datasets: 3 mixed $\rightarrow$ 2 mixed



(k) #11: 4iQ dataset: 80% $\rightarrow$ 20%



(l) #12: COMB dataset: 80% $\rightarrow$ 20%

Fig. 4. Experimental results of 12 attack scenarios. The training set is shown in Table II, and the test set is as the subtitle. The attack effect (i.e., cracking success rate) of KNNGuess is slightly/significantly better than that of our primary competitors: TarGuess-II [28], Pass2Path [13], PassBERT [26], Pass2Edit [25], and PointerGuess [29]. Note that in all attack scenarios, all password guessing models initially guess the original password, thus each figure starts from the same point. This starting point accurately reflects the proportion of identical password pairs present within the test set (see Table II for detailed data).

Pass2Edit [25], Pass2Path [13], TarGuess-II [28], PassBERT [26]) and their variants (e.g., mixing the popular password list) across the 13 attack scenarios we design. The source code of all models is open-source/shared by the original authors, and we adhere to all parameters specified in the original papers. To compare with our new popular password mixing method, we use the same popular password list and mixed method from previous work [25], [28], [29] on our KNNGuess model, namely *KNNGuess-orig-mix*. Further details about the other models can be found in Appendix A.

C. Evaluation results

The effectiveness of KNNGuess. Fig. 4 shows that our KNNGuess outperforms its primary counterparts in most attack scenarios (see Figs. 4(a)-4(d) and 4(g)-4(l)) and slightly surpasses the Pass2Edit in attack scenarios #5-#6 (where our model significantly outperforms all other models except Pass2Edit). Specifically, our KNNGuess model, with a budget of 1,000 guesses across the first 12 attack scenarios, outperforms TarGuess-II [28], Pass2Path [13], PassBERT [26], Pass2Edit [25] and PointerGuess [29] by an average of 33.79%, 73.25%, 112.4%, 11.82% and 19.16% (avg. 50.08%). In attack scenario #13, we evaluate the effectiveness of each model in cracking real-world passwords stored as salted hashes. As shown in Fig. 5, our KNNGuess model still achieves the highest cracking success rate. Specifically, our model improves by 18.84% and 111.61% compared to state-of-the-art Pass2Edit [25] and PointerGuess [29], respectively. We provide the cracking success rates of all models at some representative guess numbers (i.e., 10, 100, 1,000). Overall, see Table X and Table XI of the full version of our paper `http://bit.ly/41Lxub9` for details.

Additionally, KNNGuess achieves a greater improvement in cracking success rate for chinese users. Specifically, compared with Pass2Edit, which performs best among existing models in such scenarios, KNNGuess outperforms it by an average of 11.82% across 12 attack scenarios and achieves a 22.61% improvement on the chinese user datasets. We attribute this performance gap to the inclusion of three distinct user categories in our evaluation (i.e., common users, security-savvy users, and mixed-password datasets). Notably, security-savvy users tend to adopt more cautious password practices and exhibit lower rates of password reuse.

Moreover, KNNGuess shows significant improvement at low guess budgets. As shown in Table IV, when allowed 100 guesses and excluding identical password pairs (i.e., the targeted password is not the same as the original password), the cracking success rate of KNNGuess for common users (see Figs. 4(a)-4(f) and 4(j)) are 25.40%. While that of PointerGuess, Pass2Edit, PassBERT, Pass2Path and TarGuess-II are 22.12%, 23.46%, 13.39%, 15.28% and 20.92%. That is, the cracking success rate of our KNNGuess model is 14.83%, 8.27%, 89.69%, 66.23% and 21.41% higher than PointerGuess, Pass2Edit, PassBERT, Pass2Path and TarGuess-II. Analogously, the cracking success rate of the KNNGuess for security-savvy users is 10.26% (see Figs. 4(g)-4(i)). While that of PointerGuess, Pass2Edit, PassBERT, Pass2Path and TarGuess-II is 8.58%, 9.67%, 7.03%, 6.04% and 7.06%. The cracking success rate of KNNGuess is 19.58%, 6.10%, 45.95%, 69.87% and 45.33% higher than PointerGuess, Pass2Edit, PassBERT, Pass2Path and TarGuess-II.

The effectiveness of KNN-TPG. The TransGuess introduced in Sec. III-A obtains our KNNGuess model by integrating KNN-TPG. Therefore, we compare the guessing success rate of TransGuess with our KNNGuess model across 12 attack scenarios to show how KNN-TPG method improves the base model's effectiveness. The results show that our KNNGuess model outperforms TransGuess in all attack scenarios. Meanwhile, TransGuess performs relatively well at low guess numbers, but its guessing success rate improves only marginally as the guessing budget increases. However, KNN-TPG enables the base model to maintain considerable competitiveness even at high guess numbers. Specifically, our KNNGuess model shows an improvement in cracking success rate compared with TransGuess by 3.04%-110.28% (avg. 30.30%). When allowed 10 guesses, KNNGuess is on average 28.81% (absolutely) higher than TransGuess. Besides, with 1,000 guesses allowed, our KNNGuess model shows an average increase of 33.94% over TransGuess.

The effectiveness of the new popular password mixing method. In TarGuess-II [28] and Pass2Edit [25], they used the same method of mixing popular passwords for the model to achieve higher cracking success rate. To ensure a fair comparison with these two models, we use the same popular password lists but our new mixing method introduced in Sec. III-C. We multiply the probability of each password in the model's output guessing list by a factor α, representing the proportion of users who do not choose popular passwords, and this factor is derived through statistical analysis across different training sets. For the popular password list, we use the frequency of each password as its probability. These two lists are merged in descending order of probability to form the final guess set. When the same password appears in both lists, the maximum of the two probabilities is retained.

To demonstrate the effectiveness of the new mixing popular password method we proposed in Sec. III-C, we compared the guessing success rates of KNNGuess with KNNGuess-nomix (i.e., the KNNGuess model without mixing popular password) and KNNGuess-orig-mix (i.e., using the same mixing method from previous work [25], [28], see Sec. IV-B for details) in Fig. 4. Our KNNGuess model still achieves the highest cracking success rate. More specifically, KNNGuess improves by an average of 10.21% compared with KNNGuess-nomix, which shows that the vulnerability of users' Type-2 reuse behaviors has been underestimated in previous work (e.g., Pass2Path [13], PassBert [26] and PointerGuess [29] did not consider it), because capturing such behavior can effectively increase the cracking success rate. Additionally, our KNNGuess model improves by 9.21% compared with the KNNGuess-orig-mix model, indicating that our new mixed popular password method better simulates users' use of popular passwords. It provides a more accurate and comprehensive representation of users' password reuse behaviors (i.e., Type-2).

To explore the abilities of different models, we compare all models with mixed popular passwords in four scenarios. Notably, when mixing popular passwords, all models use the same mixing method and the same list of popular passwords, except for KNNGuess (using the new mixing popular password method proposed in this paper). As shown in Fig. 6, when all models mix popular passwords, KNNGuess improves by 14.42%, 17.70%, 42.19%, 37.32%, and 32.42% compared with Pass2Edit [25], PointerGuess [29], PassBERT [26], Pass2Path [13], and TarGuess-II [28], respectively. By comparing the results in Fig. 4, we find that the PassBERT and Pass2Path models can greatly improve effectiveness by mixing popular passwords, indicating that these two models are not strong in capturing users' Type-2 reuse behaviors.

We also completely remove the mixing popular passwords, as shown in Fig. 7, When all models do not mix popular passwords, KNNGuess improves by 34.0%, 21.09%, 114.5%, 92.42%, and 56.21% compared with Pass2Edit, PointerGuess, PassBERT, Pass2Path, and TarGuess-II, respectively. Mixing popular passwords does not improve PointerGuess as much as other models. This shows that PointerGuess can generate some popular passwords without mixing them, which explains why it performs better in attack scenarios without mixed popular passwords. Table IX shows detailed experimental data.

Overhead. We measure the attack speed of different deep learning models in scenario #13. For a fair comparison, all models are trained for 40 epochs and use only one process on the same workstation during generation, and none of the models used any accelerated generation methods, such as beam search in [29]. The results are shown in Table V. The training duration of KNNGuess comprises both model training time and datastore construction time. The results indicate that KNNGuess spends the most time in the generation. One of the reasons is that it generates three probability distributions and interpolates at each step for predicting the next character. Besides, the larger parameter k in KNN-TPG also contribute to increased generation time.

Fortunately, for online guessing attacks, the computational complexity is not particularly important, because the rate-limiting policy of websites forces attackers to focus on suc-

TABLE IV

A GRASP OF THE CRACKING EFFECTIVENESS OF OUR KNNGUESS COMPARED WITH THE SOTA MODELS PASS2EDIT [25] AND POINTERGUESS [29].

Attack settings	Guesses #	Against common users		Against security-savvy users†		Against mixed users	
		Success rate	Improvement rate‡	Success rate	Improvement rate	Success rate	Improvement rate
Excluding identical password pairs in the test set	10	19.72%	+ 13.50%	5.91%	+ 15.77%	13.83%	+ 12.90%
	100	**25.40%**	+ 11.45%	**10.26%**	+ 12.44%	**19.22%**	+ 10.71%
	1,000	30.62%	+ 11.49%	15.89%	+ 21.30%	23.32%	+ 12.12%
Including identical password pairs in the test set	10	42.98%	+ 3.94%	22.95%	+ 3.05%	30.15%	+ 4.13%
	100	**47.10%**	+ 4.04%	**26.61%**	+ 3.54%	**34.55%**	+ 4.14%
	1,000	50.72%	+ 4.53%	31.24%	+ 7.89%	37.83%	+ 5.24%

† 000Webhost is mainly used by web administrators, so its users are likely to be more security-savvy than common users (i.e., Scenarios #7-#9). Mixed users include both security-savvy users and common users, representing the average level of attack (i.e., Scenarios #11-#12).

‡ The improvement rate of our KNNGuess model compared with the State-Of-The-Art models Pass2Edit [25] and PointerGuess [29].

TABLE V

RUNNING TIME OF DIFFERENT ATTACK MODELS.†

Attack models	Train time	Test time	Speed (PW/s) ‡
KNNGuess	0:15:34	2:06:35	903
Pass2Edit [25]	0:16:12	1:53:10	1,010
PointerGuess [29]	0:16:07	1:01:36	1,840
Pass2Path [13]	0:10:12	0:53:12	2,148
PassBERT [26]	0:40:09	0:21:04	5,426

† The format of time is “hour:minute:second”. All models are trained for 40 epochs and generate 1,000 guesses for each password in the test set.

‡ The speed of generating guesses is determined by dividing the total number of guesses by the total testing time.

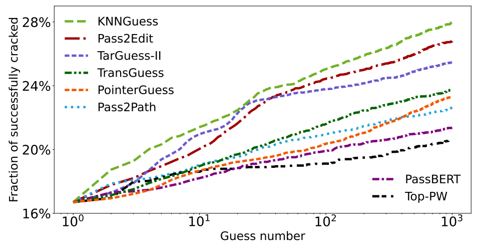


Fig. 5. The experimental results in the salted hash attack scenario #13. The training and test sets are listed in Table II. Our KNNGuess model still achieves the highest cracking success rate compare with other models.

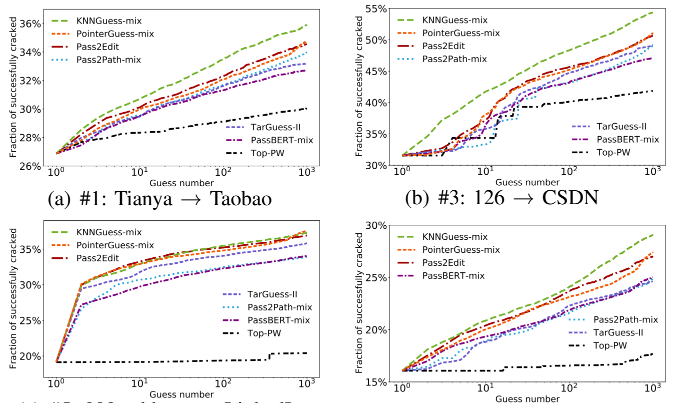
cessfully guessing passwords within as few guesses as possible [50]. Additionally, websites often impose longer login time restrictions on users than the time it takes for the model to generate a guessing list. For example, the Alexa top-10 websites allow 120-1,440 (e.g., 1,440 for Google, Baidu and Yahoo) online login attempts per day (see Table 7 of [51]), while the NIST recommended threshold is 100 attempts in 30 days [52]. Therefore, the generation time of KNNGuess is *acceptable* for online guessing attacks.

V. FURTHER EXPLORATION

We now provide a deeper understanding of the superiority of our KNNGuess model and KNN-TPG method by investigating (1) the passwords cracked by each model; (2) the password guessing lists generated by each model; and (3) the saturation in cracking success rates of each model. Furthermore, we demonstrate how KNNGuess's password-cracking capability can be transformed into a practical password strength meter.

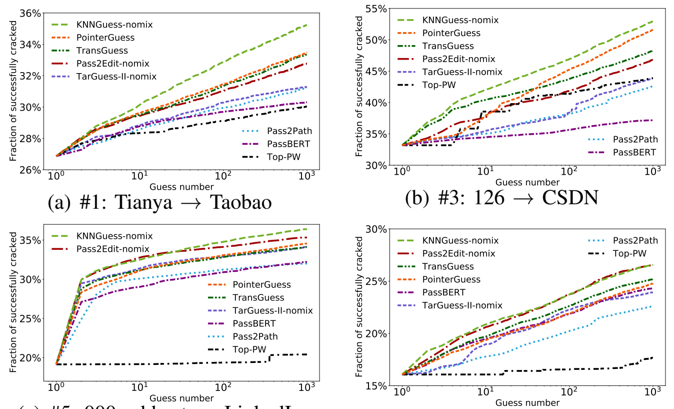
A. Analysis of cracked passwords

We examine password pairs cracked by different models in 12 attack scenarios, focusing on overlaps between the password cracking results of the models, similarities among cracked password pairs, the length of guessed passwords, and the occurrence of three types of users' reuse behavior.



(a) #1: Tianya → Taobao (b) #3: 126 → CSDN (c) #5: 000webhost → LinkedIn (d) #7: Twitter → 000webhost

Fig. 6. Experimental results of **all models mixed** with popular passwords in four attack scenarios. The training set is shown in Table II, and the test set is as the subtitle. After mixing popular passwords, our KNNGuess achieves the highest cracking success rate compared with its counterparts.



(a) #1: Tianya → Taobao (b) #3: 126 → CSDN (c) #5: 000webhost → LinkedIn (d) #7: Twitter → 000webhost

Fig. 7. Experimental results of **no-mixing** popular passwords in four attack scenarios. The training set is shown in Table II, and the test set is as the subtitle. Our model KNNGuess-nomix achieves the highest cracking success rate compared with other models that no-mixing popular passwords.

Overlap. To compare the cracking abilities of different models, we select three models: KNNGuess, Pass2Edit [25] (as the best-performing operation-sequence-based model), and PointerGuess [29] (as the best-performing character-based model). With a budget of 1,000 guesses, we analyze the overlap of password pairs cracked by three models across 12 attack scenarios, as well as the password pairs cracked *individually* by each model. As shown in Fig. 8, KNNGuess cracks passwords that the other two models cannot crack at a rate of 14.69% (4,630 pairs of 31,523 unique cracked pairs), while

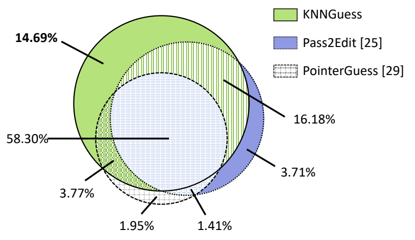


Fig. 8. The overlap ratio of passwords cracked by KNNGuess, Pass2Edit and PointerGuess. Our KNNGuess model has the highest number of cracked password pairs and the highest individual cracking ratio (i.e., 14.69%).

Pass2Edit [25] and PointerGuess [29] achieve only 3.71% (1,168 of 31,325) and 1.46% (461 of 31,523), respectively.

Similarity distribution. We analyze the similarity distribution of password pairs cracked by all models to understand their cracking capabilities. We consider using spatial distance metrics: edit distance [53] and cosine similarity [54]. Additionally, we use the Longest Common Subsequence algorithm [27], which considers the order and position of characters in a sequence, serving as a sequence alignment-based metrics method for comparing the similarity between sequences.

We evaluate the ability of different models to crack password pairs with different similarity scores, as shown in Fig. 9. Note that a smaller edit distance score indicates greater similarity between two passwords, while for cosine similarity and the longest common subsequence algorithm, a higher score signifies greater similarity between two passwords. The results show that our KNNGuess model exhibits the highest consistency in the distribution of cracked password pairs compared to all password pairs in the test set. In cases of similar password pairs (e.g., edit distance=1 or cosine similarity $\in$ [0.8,1.0]), KNNGuess exhibits the smallest proportion of cracked password pairs, closely resembling the distribution of password pairs in the test set. This trend also exists when the password pairs are dissimilar (e.g., edit distance $\geq$ 5).Moreover, we show the password length distribution for passwords cracked within 1,000 guesses by different models in Fig.11 of the full version of our paper `http://bit.ly/41Lxub9` .

Quantitative Comparison. As shown in Table VII, we provide the proportion and number of password pairs that belong to the three types of user reuse behaviors and are cracked by different models across all 12 attack scenarios. KNNGuess achieves the highest cracking of Type-3 reused password pairs (636 of all cracked 35,000 passwords), indicating that our KNN-TPG can effectively capture users' Type-3 reuse behavior. In addition, KNNGuess can guess the largest number of popular passwords (6,370 of 35,000) and outperforms other models (i.e., 6,284 of Pass2Edit [25] and 6,156 of TarGuess-II [28]) using the mixing popular password method, indicating the effectiveness of our new mixing popular password method. Pass2Path [13] and PassBERT [26] focus on Type-1 type reuse, so they rarely guess popular passwords. PointerGuess [29] directly fits the distribution of the entire password pairs space (i.e., Types 1-3). Although it can crack a certain number of Type-3 password pairs (355 of 25,958), it will affect the effectiveness of cracking other types of passwords. PointerGuess cracks significantly fewer Type-1 and Type-2 password pairs compared with both KNNGuess and Pass2Edit (e.g., 27.92% and 18.92% reduction respectively of Type-1).

B. Analysis of password guessing lists

To better understand why different models show different effectiveness for certain password pairs, we show examples of passwords that KNNGuess is able to crack but other models cannot (i.e., password pairs that belong to the 14.69% in Fig. 8). As shown in Table VI, the results show that KNNGuess exhibits the highest capability of "associative generation". This demonstrate that KNN-TPG provides additional information to the model during the generation process, enabling it to produce more diverse outputs. For instance, when generating the segment `zxc123`, it assigns more weight to generate subsequent numbers, allowing correct results to be guessed with fewer attempts. An interesting observation is that the KNNGuess model will associatively generate popular passwords related to the source password when it contains a popular password segment (e.g., deriving popular passwords such as `123456`, `123123` and `123456789` from the segment `123` contained in the source password `zxc123!@#`).

Meanwhile, the guessing lists effectively demonstrate the characteristics among different models, as shown in Table VI. TarGuess-II [28] exhibits characteristics of segment modification, resulting in the password guesses with only one segment in its guessing list (e.g., password `zxc`). PassBERT [26] excels in replacement operations, generating passwords similar to the source password (e.g., password `zxc123153`). Pass2Path [13] features the generation properties of the Seq2Seq [42] framework, enabling the insert/delete operation within passwords (e.g., password `zxc1231123`). Pass2Edit [25] can perceive changes in passwords during generation and the model will pay more attention to generating passwords under a specific structure (e.g., password `zxc12123`). PointerGuess [29] can generate popular passwords without mixing popular password lists (e.g., password `123456` and `123456789`).

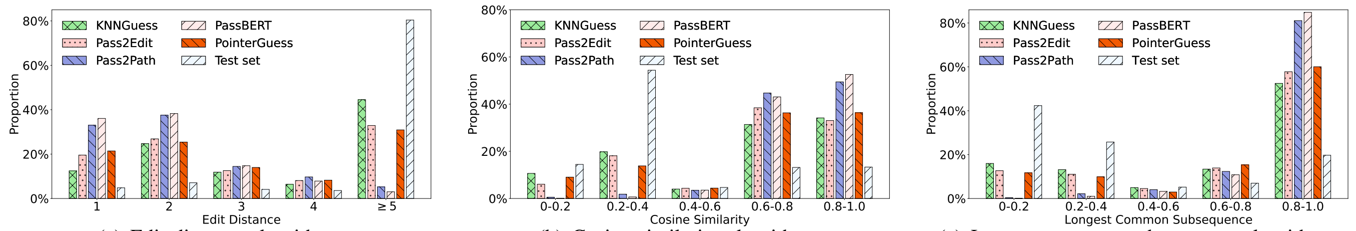
C. Analysis of the cracking-rate saturation

We now investigate how KNNGuess performs with varied sizes of the training set, and find it saturates much later than its counterparts. More specifically, when the training set size increases from 5×10^6 to 5×10^7, the guessing success-rates of all its counterparts show a clear saturation trend. That is, for targeted password guessing models, increasing the dataset size of password pairs moderately improves the guessing success-rate of the models, whereas inputting an excessive amount of password pairs as training data leads to a cracking rate saturation. We take the COMB dataset [47] as an example, which has the largest number of leaked passwords. From the training set of the COMB dataset, we randomly select password pairs of sizes $5*10^4$, $5*10^5$, $5*10^6$, $5*10^7$. The four different-sized password pair datasets are used as training inputs for all models. We show the comparison results with Pass2Edit [25] in Fig. 10. We can find that the effectiveness of Pass2Edit and PointerGuess is close to saturation after

TABLE VI
THE GUESSING LISTS GENERATED BY KNNGUESS AND THE OTHER COMPARED MODELS FOR AN EXAMPLE PASSWORD.†

Models	Examples of a password pairs: ((`zxc123!@#`, `zxcvbnm123`)									
Guesses	1	2	3	4	5	6	7	8	9	10
KNNGuess-nomix‡	zxc123!@#	zxc123	zxc123123	zxc123456	123456	123123	**zxcvbnm123**	zxc12345	123456789	zxcvbnm
Pass2Edit-nomix [25]	zxc123!@#	zxc123	zxc123123	zxc123!	zxc12123	zxc1123	ZXC123	zxc12323	zxc	zx123
PointerGuess [29]	zxc123!@#	zxc123	123	123456	123123	zxc123456	zxc	ZXC123	zxc123123	123456789
Pass2Path [13]	zxc123!@#	123123	1xc123123	c123123	xc123123	zxc1231123	zxc1231213	zxc123123	zxc12312a3	zxc12313
PassBERT [26]	zxc123!@#	zxc12313	zxc1233	zxc1231	zxc123113	zxc123153	zxc12373	zxc12323	zx12313	zxc12353
TarGuess-II-nomix [28]	zxc123!@#	zxc123	zxc	zxc123!@	zxc12!@#	123	zx123!@#	123!@#	zxc12	zx123

†All models guess the source password in the (source password, targeted password) pair as the first guess, modeling users' behavior of reusing directly. A password with dark gray represents a specific one that only one model can generate within 10 guesses, and **bold** passwords indicate successful guesses.
‡"*-nomix*" means that the model does not perform mixing popular password list operations, and represents the original guess generated by the model.



(a) Edit distance algorithm (b) Cosine similarity algorithm (c) Longest common subsequence algorithm

Fig. 9. The similarity distribution of password pairs cracked by 5 attack models. Figs. 9(a)-9(c) show the results using spatial distance metrics and sequence alignment-based metrics (i.e., edit distance, cosine similarity, and longest common subsequence (LCS) algorithm), to measure the similarity distribution of cracked passwords. The "Test set" represents all password pairs in the test set. KNNGuess is particularly good at predicting distant password reuse behaviors.

TABLE VII
THE NUMBER OF PASSWORD PAIRS CRACKED BY EACH PASSWORD MODEL FOR THREE TYPES OF REUSE BEHAVIORS.†

Model name	Type-1	Type-2	Type-3
KNNGuess (This work)	27994 (79.98%)	6370 (18.2%)	636 (1.82%)
Pass2Edit [25]	26025 (79.85%)	6284 (19.28%)	283 (0.87%)
PointerGuess [29]	21884 (84.31%)	3719 (14.33%)	355 (1.4%)
Pass2Path [13]	15368 (99.11%)	7 (0.05%)	131 (0.84%)
PassBERT [26]	19242 (99.73%)	3 (0.02%)	49 (0.25%)
TarGuess-II [28]	21394 (76.95%)	6156 (22.14%)	254 (0.91%)

†We defined three types of user reuse behaviors in Sec. I-A. For example, Type-1 means that user makes simple changes to the source password.

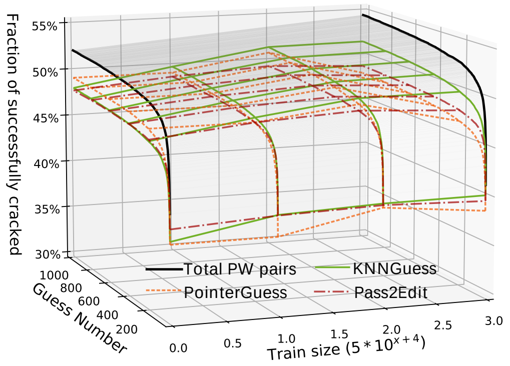


Fig. 10. KNNGuess, PointerGuess and Pass2Edit's [25] effectiveness in scenario #12 (see Table II) when the training set size changes. The x-axis represents the size of the training set, the y-axis indicates the number of guesses, and the vertical axis represents the cracking success rate of guessing. "Total PW pairs" represents the highest guess success rate achieved by combining the guess lists of the three models.

the training set size exceeds $5 * 10^6$, and the effectiveness has a downward trend, while our KNNGuess model still has an increasing trend. We put the results of other models in Appendix D.

There are two possible reasons for the saturation of the guessing success rate: (1) *The oversized training set introduces more noise*. The excessive noise affects the model's generalization ability, especially when the password pairs increase from $5 * 10^6$ to $5 * 10^7$ (45 million additional pairs). All models use filtering methods (e.g., edit distance [53] and cosine similarity [54]) to screen the training set. There are differences between different filtering methods, and they cannot ensure the filtering of all bad data that may affect the convergence of the model (which is also why KNN-TPG can demonstrate its advantages). Therefore, the oversized training set brings more noise and affects the model's fitting ability; (2) *The data distribution changes*. The training process of the model involves fitting the data distribution of the training set. The COMB dataset contains leaked passwords from multiple languages, websites with different policies, and different types of services. Increasing the size of the training set too much introduces more types of password pair transformations, which makes it difficult for the model to generalize. A significant difference in distribution between the new and old training sets can cause the model to perform poorly on the new training set. Our KNNGuess mitigates this limitation through its KNN-TPG component, which retrieves semantically relevant candidates from the datastore when the model's generalization capacity is insufficient. This retrieval mechanism enhances training robustness by compensating for the model's coverage gaps.

D. Application to a targeted password strength meter

Password strength meters (PSMs) help users assess the strength of their passwords when creating new ones, aiming to prevent users from generating weak passwords and assist in creating strong ones. Previous works [14], [30], [31], [55] proposed methods to measure password strength accurately, but faced limitations. For instance, guessing-based PSMs depend on the algorithm's effectiveness and only assess overall strength, without providing modification suggestions.

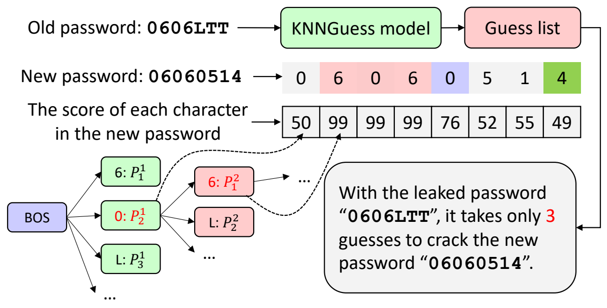


Fig. 11. Example of inputting a password pair into KNN-PSM, it provides two pieces of information: how many guesses are needed to correctly guess the new password in the event of an old password leak, and the predictability score for each character in the password, where a higher score indicates that users are more likely to choose that character.

Therefore, we devise KNN-PSM based on the KNNGuess, which not only offers users macro-level password strength advice from the perspective of the entire password, but also provides modification suggestions from the perspective of each individual character in the password. As shown in Fig. 11, we present the results of inputting a pair of password pair into KNN-PSM. KNN-PSM outputs two results: the predictive score for each character (e.g., the first character "0" has a score of 50), and the number of attackers required to crack the new password in the event of the old password being leaked. For example, when the attacker knows the leaked password `0606LTT`, it only takes 3 times to crack `06060514`.

At each decoding step, as we generate each character of the new password correctly, we obtain the probability of generating the character in the final distribution. This distribution is generated jointly by the Basic Distribution, KNN Distribution, and Local Distribution. In Sec. IV-C, we demonstrate that its predictive probabilities are more accurate than only using the Basic Distribution to predict the next character. By normalizing the probabilities, we obtain scores for each character in the new password:

$$score_i = \frac{prob_i[targeted_i]}{\sum_{j=0}^{vocab\_size-1} prob_i[targeted_j]}, \tag{6}$$

where the $score_i$ represents the score of the i-th character in the new password, $targeted_i$ denotes the i-th character in the new password, and prob represents the final distribution, i.e., the prediction of probabilities for all characters. The normalized score (Eq. 6) quantifies the model's confidence in predicting each character. For the predictive score, a higher score indicates that attackers are more likely to guess this character, implying that choosing this character at this position is less secure. KNN-PSM warns users to modify the character at this position (e.g., the red color in Fig. 11) to reduce the predictive score and improve password security. Therefore, users can refer to the number of guesses needed to crack to dynamically modify the old password or choose a completely new one. Further, we release the trained model as a password strength meter. This approach evaluates input passwords by estimating their guessability or assigning strength levels, preventing weak ones. A preliminary anonymous version is available at https://github.com/KNNGuess/KNNGuess-code .

VI. Conclusion

This paper introduces a new non-parametric method, k-nearest-neighbors targeted password guessing (KNN-TPG), for modeling users' password reuse behaviors. By creatively integrating it with our Transformer-based model, we obtain a new password guessing model, named KNNGuess. The KNN-TPG method maps all passwords in the training set to high dimensional vectors and builds a datastore offline, from which the KNN-TPG retrieves k neighbor information during password generation to enhance its guessing capability. KNNGuess effectively characterizes users' password reuse behaviors, and accurately models attackers' password tweaking attacks, providing a better understanding of password security.

Ethics Considerations

Although these datasets are publicly available on the Internet and the dark web, and widely used in previous password research [13], [25]–[28], [56], we treat them as private. In order to prevent potential damage to the individuals, we handle these datasets with caution and minimize the risk of secondary leakage. During the experimental process, we implement the following precautionary measures: (1) After completing the matching of identical email addresses for the same user, we promptly delete all email information to ensure that personally identifiable information (PII) is not leaked (meaning that passwords and users' corresponding email addresses do not appear simultaneously). (2) We ensure that all datasets are processed and stored on computers and hard drives not connected to the Internet. We only report macro-level aggregated statistical information and show some typical passwords. (3) After completing the experiments and analysis, we promptly delete all processed data (e.g., multiple passwords corresponding to the same user on different service websites) to prevent attackers from exploiting them for further password guessing attacks. Note that while we use leaked password datasets to characterize more sophisticated attackers, our fundamental goal is to assist security administrators/users in accurately assessing their password strength from the perspective of guessing attacks (as the number of guesses can serve as a good indicator of password strength [14], [51], [55], [57]). All the leaked datasets we used are obtainable from various publicly available sources on the Internet, ensuring the reproducibility of experimental results.

Acknowledgment

The authors are grateful to the shepherd and anonymous reviewers for their invaluable comments. Ding Wang is the corresponding author. This research was in part supported by the National Natural Science Foundation of China under Grants Nos. 62172240, 62222208 and 62572259, and by the Fundamental Research Funds for the Central Universities, Nankai University (Grant No. 63243154). See the full version of this paper at http://bit.ly/41Lxub9 .

References

[1] M. Shirvanian and S. Agrawal, "{2D-2FA}: A new dimension in two-factor authentication," in *Proc. ACM ACSAC 2021*, pp. 482–496.

[2] P. Shrestha, A. T. Mahdad, and N. Saxena, "Sound-based two-factor authentication: Vulnerabilities and redesign," *ACM Trans. Priv. Secur.*, vol. 27, no. 1, pp. 1–27, 2024.

[3] S. G. Lyastani, M. Schilling, M. Neumayr, M. Backes, and S. Bugiel, "Is FIDO2 the kingslayer of user authentication? A comparative usability study of FIDO2 passwordless authentication," in *Proc. IEEE S&P 2020*, pp. 268–285.

[4] J. Bonneau, C. Herley, P. Oorschot, and F. Stajano, "The quest to replace passwords: A framework for comparative evaluation of web authentication schemes," in *Proc. IEEE S&P 2012*, pp. 553–567.

[5] J. Yan, A. Blackwell, R. Anderson, and A. Grant, "Password memorability and security: Empirical results," *IEEE Secur. Priva.*, vol. 2, pp. 25–31, 2004.

[6] J. Bonneau, C. Herley, P. C. Van Oorschot, and F. Stajano, "The request to replace passwords: A framework for comparative evaluation of web authentication schemes," in *Proc. IEEE S&P 2012*, pp. 553–567.

[7] J. Bonneau, C. Herley, P. van Oorschot, and F. Stajano, "Passwords and the evolution of imperfect authentication," *Commun. ACM*, vol. 58, no. 7, pp. 78–87, 2015.

[8] M. L. Mazurek, S. Komanduri, T. Vidas, L. F. Cranor, P. G. Kelley, R. Shay, and B. Ur, "Measuring password guessability for an entire university," in *Proc. ACM CCS 2013*, pp. 173–186.

[9] D. Wang and P. Wang, "The emperor's new password creation policies," in *Proc. ESORICS 2015*, pp. 456–477.

[10] K. Lee, S. Sjöberg, and A. Narayanan, "Password policies of most top websites fail to follow best practices," in *Proc. SOUPS 2022*.

[11] *LastPass Is Making Account Updates. Here's Why*, Jan. 2024, https://blog.lastpass.com/posts/2024/01/lastpass-is-making-account-updates-heres-why.

[12] H. P. Google, *Online Security Survey*, Feb. 2019, https://services.google.com/fh/files/blogs/google_security_infographic.pdf.

[13] B. Pal, T. Daniel, R. Chatterjee, and T. Ristenpart, "Beyond credential stuffing: Password similarity models using neural networks," in *Proc. IEEE S&P 2019*, pp. 417–434.

[14] D. Wang, D. He, H. Cheng, and P. Wang, "fuzzyPSM: A new password strength meter using fuzzy probabilistic context-free grammars," in *Proc. IEEE/IFIP DSN 2016*, pp. 595–606.

[15] *Password administration for system owners*, Nov. 2018, https://www.ncsc.gov.uk/collection/passwords/updating-your-approach.

[16] *Stick with Security: Require secure passwords and authentication*, Aug. 2017, https://www.ftc.gov/business-guidance/blog/2017/08/stick-security-require-secure-passwords-and-authentication.

[17] US-CERT, *Choosing and Protecting Passwords*, Nov. 2019, https://us-cert.cisa.gov/ncas/tips/ST04-002.

[18] B. Ur, F. Noma, J. Bees, S. M. Segreti, R. Shay, L. Bauer, N. Christin, and L. F. Cranor, "" i added'!'at the end to make it secure": Observing password creation in the lab," in *Proc. SOUPS 2015*, pp. 123–140.

[19] S. Pearman, J. Thomas, P. E. Naeini, H. Habib, L. Bauer, N. Christin, L. F. Cranor, S. Egelman, and A. Forget, "Let's go in for a closer look: Observing passwords in their natural habitat," in *Proc. ACM CCS 2017*.

[20] E. M. Redmiles, S. Kross, and M. L. Mazurek, "How I learned to be secure: a Census-representative survey of security advice sources and behavior," in *Proc. ACM CCS 2016*, pp. 666–677.

[21] *Data Breach Investigations Report*, June 2023, https://www.verizon.com/business/resources/reports/2024/dbir/2024-dbir-data-breach-investigations-report.pdf.

[22] *2021 Credential Spill Report*, Feb. 2021, https://www.f5.com/company/news/features/credential-spill-incidents-double-as-hacker-sophistication-conti.

[23] *Hack Brief: An Adult Cam Site Exposed 10.88 Billion Records*, May 2020, https://www.wired.com/story/cam4-adult-cam-data-leak-7tb/.

[24] *Mother of all breaches reveals 26 billion records: what we know so far*, Jan. 2024, https://cybernews.com/security/billions-passwords-credentials-leaked-mother-of-all-breaches/.

[25] D. Wang, Y. Zou, Y.-A. Xiao, S. Ma, and X. Chen, "PASS2EDIT: A multi-step generative model for guessing edited passwords," in *Proc. USENIX SEC 2023*, pp. 9803–1000.

[26] M. Xu, J. Yu, X. Zhang, C. Wang, S. Zhang, H. Wu, and W. Han, "Improving real-world password guessing attacks via bi-directional transformers," in *Proc. USENIX SEC 2023*, pp. 1001–1018.

[27] A. Das, J. Bonneau, M. Caesar, N. Borisov, and X. Wang, "The tangled web of password reuse," in *Proc. NDSS 2014*, pp. 1–15.

[28] D. Wang, Z. Zhang, P. Wang, J. Yan, and X. Huang, "Targeted online password guessing: An underestimated threat," in *Proc. ACM CCS 2016*, pp. 1242–1254.

[29] K. Xiu and D. Wang, "Pointerguess: Targeted password guessing model using pointer mechanism," in *Proc. USENIX SEC 2024*, pp. 5555–5572.

[30] W. Melicher, B. Ur, S. Komanduri, L. Bauer, N. Christin, and L. F. Cranor, "Fast, lean and accurate: Modeling password guessability using neural networks," in *Proc. USENIX SEC 2017*, pp. 175–191.

[31] D. Pasquini, G. Ateniese, and M. Bernaschi, "Interpretable probabilistic password strength meters via deep learning," in *Proc. ESORICS 2020*, pp. 502–522.

[32] U. Khandelwal, A. Fan, D. Jurafsky, L. Zettlemoyer, and M. Lewis, "Nearest neighbor machine translation," in *Proc. ICLR 2021*.

[33] M. Nicholas, *68 Million Reasons Why Your Small Business Needs a Password Manager*, Jan. 2017, https://blog.dashlane.com/68-million-reasons-why-your-small-business-needs-a-password-manager/.

[34] A. Hanamsagar, S. S. Woo, C. Kanich, and J. Mirkovic, "Leveraging semantic transformation to investigate password habits and their causes," in *Proc. ACM CHI 2018*, pp. 1–10.

[35] S. Li, Z. Wang, R. Zhang, C. Wu, and H. Luo, "Mangling rules generation with density-based clustering for password guessing," *IEEE Trans. Depend. Secur. Comput.*, vol. 20, no. 5, pp. 3588–3600, 2022.

[36] L. Li, B. Pal, J. Ali, N. Sullivan, R. Chatterjee, and T. Ristenpart, "Protocols for checking compromised credentials," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1387–1403.

[37] B. Pal, M. Islam, M. S. Bohuk, N. Sullivan, L. Valenta, T. Whalen, C. Wood, T. Ristenpart, and R. Chatterjee, "Might i get pwned: A second generation compromised credential checking service," in *Proc. USENIX SEC 2022*, pp. 1831–1848.

[38] J. Kim, M. Song, M. Seo, Y. Jin, and S. Shin, "Passrefinder: Credential stuffing risk prediction by representing password reuse between websites on a graph," in *Proc. IEEE S&P 2024*, pp. 1–20.

[39] M. Islam, M. S. Bohuk, P. Chung, T. Ristenpart, and R. Chatterjee, "Araña: Discovering and characterizing password guessing attacks in practice," in *Proc. USENIX SEC 2023*, 2023, pp. 1019–1036.

[40] M. Weir, S. Aggarwal, B. de Medeiros, and B. Glodek, "Password cracking using probabilistic context-free grammars," in *Proc. IEEE S&P 2009*, pp. 391–405.

[41] J. Ma, W. Yang, M. Luo, and N. Li, "A study of probabilistic password models," in *Proc. IEEE S&P 2014*, pp. 689–704.

[42] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in neural information processing systems*, vol. 27, 2014.

[43] Y. Shen, X. Ma, Z. Tan, S. Zhang, W. Wang, and W. Lu, "Locate and label: A two-stage identifier for nested named entity recognition," in *Proc. ACL 2021*, pp. 2782–2794.

[44] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[45] T. Wolf, L. Debut, V. Sanh, and et al., "Transformers: State-of-the-art natural language processing," in *Proc. ACL 2020*, pp. 38–45.

[46] *Identities in the Wild: The Tsunami of Breached Identities Continues*, May 2018, https://4iq.com/wp-content/uploads/2018/05/2018IdentityBreachReport4iQ.pdf/.

[47] *COMB: largest breach of all time leaked online with 3.2 billion records*, July 2022, https://cybernews.com/news/largest-compilation-of-emails-and-passwords-leaked-free/.

[48] Z. Li, W. Han, and W. Xu, "A large-scale empirical analysis on chinese web passwords," in *Proc. USENIX SEC 2014*, pp. 559–574.

[49] D. Wang, P. Wang, D. He, and Y. Tian, "Birthday, name and bifacial-security: Understanding passwords of Chinese web users," in *Proc. USENIX SEC 2019*, pp. 1537–1555.

[50] M. Dürmuth, D. Freeman, S. Jain, B. Biggio, and G. Giacinto, "Who are you? A statistical approach to measuring user authenticity," in *Proc. NDSS 2016*, pp. 1–15.

[51] D. Wang, X. Shan, Q. Dong, Y. Shen, and C. Jia, "No single silver bullet: Measuring the accuracy of password strength meters," in *Proc. USENIX SEC 2023*, pp. 947–964.

[52] P. A. Grassi, E. M. Newton, R. A. Perlner, and et al., "NIST 800-63B digital identity guidelines: Authentication and lifecycle management,"

McLean, VA, Tech. Rep., Mar. 2020, https://pages.nist.gov/800-63-3/sp800-63b.html.

[53] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," *Soviet physics. Doklady*, vol. 10, pp. 707–710, 1965.

[54] W. W. Cohen, P. Ravikumar, and S. E. Fienberg, "A comparison of string distance metrics for name-matching tasks," in *Proc. IIWEB 2003*, pp. 73–78.

[55] C. Castelluccia, M. Dürmuth, and D. Perito, "Adaptive password-strength meters from Markov models," in *Proc. NDSS 2012*.

[56] D. Pasquini, A. Gangwal, G. Ateniese, M. Bernaschi, and M. Conti, "Improving password guessing via representation learning," in *Proc. IEEE S&P 2021*, pp. 265–282.

[57] P. G. Kelley, S. Komanduri, M. L. Mazurek, R. Shay, T. Vidas, L. Bauer, N. Christin, L. F. Cranor, and J. Lopez, "Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms," in *Proc. IEEE S&P 2012*, pp. 523–537.

[58] *SecLists: the pentester's companion*, 2024, https://github.com/danielmiessler/SecLists/tree/master/Passwords.

[59] *Password Similarity Models using Neural Networks*, 2019, https://github.com/Bijeeta/credtweak.

[60] *PassBertStrengthMeter*, 2023, https://github.com/snow0011/PassBertStrengthMeter.

[61] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with GPUs," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2019.

Appendix

A. Supplementary information for comparison models

We introduce the comparative models to our work, including PointerGuess [29], Pass2Edit [25], Pass2Path [13], TarGuess-II [28], and PassBERT [26], which were proposed in previous studies. We introduce detailed information about these benchmark models for comparison.

TarGuess-II. Wang et al. [28] proposed a probabilistic model called TarGuess-II in 2016, which is based on a rigorous mathematical probability model, departing from purely heuristic designs. This model relies on PCFG model [40]and Markov model [41], and describes user password reuse behavior through modification operations at two levels (i.e., structural level and token level), such as head deletion, tail insertion, etc. Additionally, this model incorporates popular passwords, significantly enhancing its effectiveness. Despite the availability of many pre-compiled lists of popular/weak passwords on the Internet (e.g., OWASP maintains a available list of common passwords [58]), for the sake of ensuring a fair comparison with the behavior of mixing popular passwords as proposed by Wang et al. [25], [28], the popular password lists constructed in this paper are consistent with TarGuess-II. For the Chinese dataset, the popular password dictionary is $\mathcal{L}_C$=$\{pw|$the value of $P_{\text{csdn}}(pw) * P_{126}(pw) * P_{\text{Dodonew}}(pw)$ ranks top-$10^4\}$, while for the English dataset, it is $\mathcal{L}_E$=$\{pw|$the value of $P_{000\text{Webhost}}(pw) * P_{\text{Yahoo}}(pw) * P_{\text{LinkedIn}}(pw)$ ranks top-$10^4\}$.

Pass2Path. In 2019, Pal et al. [13] proposed the Pass2Path model based on the Seq2Seq framework, which models the process of user password modification as a series of atomic edit operations. They trained the model and predicted the sequence of editing operations using the Seq2Seq [42] framework. We use parameters from its open-source code [59]. Specifically, we trained for 80 epochs, with the 3 layers of RNN. The learning rate is 0.0003, hidden layer size is 128, embedded layer size is 200, and dropout probability is 0.2, filtering the training set with edit distance ≤ 4. Note that Wang et al. [25] proposed the "Pass2Path-bugfix" model to fix the original model, but there is no significant improvement in the effectiveness. Therefore, we adopt the original "Pass2Path" model in our experiments.

PassBERT. Xu et al. [26] proposed the pre-trained model PassBERT based on bidirectional Transformers in 2023. In the field of password guessing, they employed pre-training and fine-tuning paradigms to model user password reuse behavior. By utilizing a sequence labeling mechanism [43], PassBERT predicts an editing operation for each character in the password, forming a sequence of editing operations, known as an editing path. We use their open-source code [60] and a pre-trained model on 4iQ, fine-tuning it in our 13 attack scenarios (see Table II) and generating a list of guesses. Note that PassBERT generates passwords with spaces during the generation process. We remove the spaces from these passwords and ensure that they are different from previously generated passwords to generate 1,000 distinct guesses.

Pass2Edit. Wang et al. [25] found that the models utilizing editing operations cannot perceive changes that occur during the password modification process. So they introduced, for the first time in 2023, a multi-step decision mechanism. They modeled password generation as a multi-step decision classification task and developed a model called Pass2Edit. In the model's input stage, it doesn't only input the original password PW_{org}, but also requires inputting the current password PW_{cur} (which equals the original password at the first time step), concatenating the two before inputting into the GRU layer. Unlike the Seq2Seq framework of the Pass2Path model, the Pass2Edit-nomix model is essentially a classifier, generating an edit operation at each step until the password generation is complete. Additionally, Pass2Edit-nomix also employs the same popular password list as TarGuess-II-mix to mix the guessing dictionary, aiming for a higher cracking rate. The model after mixing popular passwords is named *Pass2Edit*. We utilize the source code generously provided by the authors and retained the model's default parameters, only modifying the datasets used as input.

PointerGuess. Xiu and Wang [29] introduced the pointer mechanism to simulate users' password reuse behaviors. They did this by asking two questions: what the users want to copy/keep and what the users want to tweak. This led to the development of a targeted password guessing model called PointerGuess. Specifically, PointerGuess defines password reuse from both individual and population-wide perspectives, thereby covering the complex password modification behaviors of users. Note that PointerGuess can capture popular passwords used by users without mixing password lists. Therefore, when no mixing of popular password lists is performed, PointerGuess achieves a higher cracking success rate.

Top-PWs. In different scenarios, we calculate the frequency of different passwords in the training set and arrange passwords in descending order of frequency to generate a guessing dictionary composed of popular passwords. It represents the attacker's minimal strategy, that is, guessing passwords that have already been seen, without utilizing any model. Note that this guessing dictionary is distinct from the mixed popular dictionaries (i.e., $\mathcal{L}_C$ and $\mathcal{L}_E$) used by the "*-mix*" models (e.g., TarGuess-II and Pass2Edit).

TransGuess. To demonstrate the effectiveness of the KNN-TPG method, we introduce the TransGuess model as our base model in this paper. It is a character-to-character model based on the Transformer architecture, utilizing the key-sequence mechanism. The parameters remain consistent with the default parameters of the Transformer [44], ensuring simplicity and easy extendibility of KNNGuess model. Specific parameters are detailed in Sec. III-A.

KNNGuess-orig-mix. To demonstrate our new method of mixing popular passwords, we use the same mixing method and list from previous work [25], [28]. We apply them to the KNNGuess model without mixed popular passwords. The new model is called KNNGuess-orig-mix. Compared with the KNNGuess model presented in this paper, the only difference is the method used to mix popular passwords. By comparing KNNGuess-orig-mix with KNNGuess, we can understand the advantages of our new method for mixing popular passwords.

KNNGuess-nomix. We build a Transformer model based on password character levels, combined with our proposed non-parametric method, KNN-TPG. Without manually mixing any popular passwords, we get the KNNGuess-nomix model. By comparing this model with the one that mixes popular passwords, we can understand the weak behavior of users using popular passwords in different attack scenarios.

B. Specific implementation of KNN-TPG

In Sec. III-B, we introduce how to utilize the KNN-TPG method to assist in generating password guesses. Now, we provide some supplementary information and specific implementations regarding KNN-TPG. In the process of building the datastore, we input the processed training set into TransGuess in batch form, obtaining a multi-dimensional representation vector (e.g., The dimension is 128*512, where 128 is the batch size, and 512 is the size of TransGuess hidden layer). We use the representation vectors as the key, and the value is a tensor, namely $tensor_{tar}$, representing the i-th dimension (i is the current decoding step) of the targeted password vector in batch form. We dump them into the datastore in the form of key-value pairs. Note that since the targeted passwords are not fully aligned, we need to check the values of $tensor_{tar}$. We only consider vectors with values greater than 1 (because 1 represents the padding symbol PAD). Finally, we concatenate the $tensor_{tar}$ with the representation vectors and continue the decoding process until reaching the maximum decoding length or $tensor_{tar}$ are all filled with PAD. We precisely formalize this process in Algo. 1 for clarity.

To perform vector search in large-scale high-dimensional vector spaces, we utilize the Faiss [61] toolkit[1], designed specifically for efficient similarity search and dense vector clustering. Its core functionality involves encapsulating vectors and building indexes. We employ the IndexIVFPQ index type, which combines the Inverted File (IVF) index and Product Quantization (PQ) compression technique, to achieve efficient approximate nearest neighbor search on large-scale datasets. The inverted file index is a type of index based on vector quantization. It partitions the vector space into a series of non-overlapping regions called clusters, and maintains a list containing indices of vectors for each region. This structure significantly reduces the computational cost required during the search, as the search is conducted only within a subset of clusters that are similar to the query vector. PQ is a compression technique used for vector quantization. It divides high-dimensional vectors into several lower-dimensional sub-vectors and assigns a discrete codeword (codebook) to each sub-vector. These sub-codewords are quantized and compressed, allowing for efficient storage and processing. During search, it can calculate distances between vectors with lower memory consumption and faster speed.

[1]https://github.com/facebookresearch/faiss/

TABLE VIII
MEMORY CONSUMPTION OF KNN-TPG METHOD.

Training set size	$5*10^4$	$5*10^5$	$1*10^6$	$5*10^6$	$1*10^7$
Keys	1.2GB	11.7GB	23.4GB	116.7GB	166.7GB
Values	4.6MB	45.6MB	91.2MB	455.9MB	651.3MB
Faiss index[†]	49.9MB	419.0MB	830.1MB	4.1GB	5.9GB

[†]Faiss index is an index built by the Faiss toolkit based on key-value pairs, which is ultimately used in the retrieval process.

The KNN-TPG method requires constructing an offline datastore, which inevitably results in memory consumption when the number of password pairs is large. We test the memory consumption of the KNN-TPG method under different training set sizes, and the results are shown in Table VIII. The Faiss index is built jointly by keys and values and is only used in subsequent high-dimensional vector retrieval processes. Due to the storage of a large number of high-dimensional vectors, keys incur the greatest memory consumption. The index files constructed by Faiss also have noticeable memory consumption. We acknowledge that this is one of the side effects of increasing the cracking success rate. However, for malicious attackers, this is entirely acceptable. For example, we use a 4TB hard drive during the experimental process. In addition, we set the dimension of the latent space to 512 during the experiment, which results in high memory storage consumption for a single vector. Thus, memory consumption can be reduced by reducing the latent space dimension.

C. Selection of hyperparameters in KNN-TPG

For our method KNN-TPG, different hyperparameter choices affect the guessing success rate of KNNGuess. We test the impact of different hyperparameter combinations on the model in attack scenario #4. First, we determine the value of λ_{Local}, which reveals the proportion of users exhibiting segmental attention behavior (i.e., users only focusing on a segment modification, see Sec. III-B). By analyzing the training set, we find that this portion of users accounts for approximately 10%. Therefore, we determine $\lambda_{Local} = 0.1$.

TABLE IX

COMPARISON OF THE CRACKING SUCCESS RATE OF DIFFERENT METHODS WHEN MIXING/NOT MIXING POPULAR PASSWORDS (EXPERIMENTAL RESULTS EXCLUDE ALL IDENTICAL PASSWORD PAIRS IN THE TEST SET).†

Experimental Setup		KNNGuess		Pass2Edit [25]		PointerGuess [29]		TarGuess-II [28]		Pass2Path [13]		PassBERT [26]	
Attack scenarios	Guesses #	-mix	-nomix	-mix‡	-nomix	-mix	-nomix	-mix	-nomix	-mix	-nomix	-mix	-nomix
#1 Tianya → Taobao	10	**5.43%**	**3.99%**	4.63%	3.61%	4.41%	3.87%	3.76%	2.79%	3.80%	2.42%	3.74%	2.65%
	100	**9.09%**	**7.93%**	7.44%	5.69%	7.12%	6.30%	6.48%	4.67%	6.62%	4.22%	6.09%	3.87%
	1,000	**12.36%**	**11.11%**	10.50%	8.06%	10.80%	9.03%	8.68%	6.03%	9.70%	5.95%	7.97%	4.70%
#3 126 → CSDN	10	**15.25%**	**11.87%**	9.02%	6.53%	10.03%	8.83%	6.41%	3.61%	2.95%	4.08%	8.25%	2.26%
	100	**24.53%**	**19.83%**	20.65%	9.41%	19.99%	17.03%	19.31%	8.87%	17.04%	6.37%	16.99%	5.36%
	1,000	**33.31%**	**30.19%**	27.97%	17.85%	28.63%	26.91%	25.73%	16.59%	25.53%	13.17%	22.74%	8.26%
#5 000Webhost → LinkedIn	10	17.26	16.89%	**17.62%**	**17.00%**	17.16%	14.97%	15.93%	15.28%	14.40%	13.58%	13.29%	12.34%
	100	**20.20%**	**19.75%**	19.90%	18.55%	19.38%	17.23%	18.49%	17.01%	16.50%	14.96%	16.27%	14.54%
	1,000	22.42%	**21.57%**	21.99%	20.01%	**22.80%**	19.08%	20.63%	18.57%	18.33%	15.90%	18.44%	16.16%
#7 Twitter → 000Webhost	10	**5.85%**	**6.01%**	5.32%	5.51%	4.86%	4.07%	3.45%	3.47%	3.58%	2.10%	3.99%	4.18%
	100	**9.58%**	**9.53%**	9.10%	9.13%	8.44%	6.96%	7.43%	7.38%	6.57%	4.96%	6.95%	6.92%
	1,000	**15.50%**	**13.18%**	13.06%	12.49%	13.53%	10.39%	10.18%	9.35%	10.48%	7.78%	10.67%	9.82%

† The detailed data of the attack scenarios are shown in Table II. All passwords in the test sets (pw_A,pw_B) satisfy $pw_A \neq pw_B$. All models include both mixed popular passwords (-mix) and no mixed popular (-nomix) methods. We compare their cracking success rates respectively. The value of **bold** in each row indicates the highest cracking success rate of the model in the same method (i.e. -mix or -nomix). In all 12 attack cases, our KNNGuess model achieved 10 best results when mixing popular passwords and 11 best results when not mixing popular passwords.

‡ Note that in order to distinguish whether different models mix popular passwords, each model has two methods, -mix and -nomix. If the model does not originally have a method to mix popular passwords, then model-nomix corresponds to the model name in Appendix A (includes PassBERT [26], Pass2Path [13], and PointerGuess [29], i.e., PassBERT-nomix means PassBERT in Appendix A). If the model originally has a method to mix popular passwords, then model-mix corresponds to the model in Appendix A (i.e. KNNGuess, Pass2Edit [25], TarGuess-II [28])

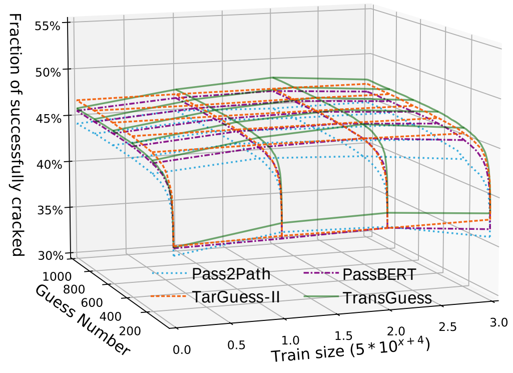


Fig. 12. The effectiveness of other models when the training set size varies in Scenario #12 (see table II). Here, we show the results of Pass2Path [13], PassBERT [26], TarGuess-II [28] and TransGuess (in this work). The x-axis represents the size of the training set, the y-axis indicates the number of guesses, and the vertical axis represents the cracking success rate of guessing.

Based on this, we select several sets of representative hyperparameter values, as shown in Table III. The results indicate that when the value of λ_{Basic} is too large, it causes KNNGuess to gradually degrade into the TransGuess model. Additionally, excessively large or small values of k slightly affect the model's effectiveness: a too-small k prevents it from fully retrieving more results, while an overly large k retrieves more noise. Overall, we chose the $\lambda_{Basic} = 0.5$, $\lambda_{KNN} = 0.4$, $k = 32$, $T = 300$ hyperparameter set because it achieves the best model effectiveness among the various combinations.

From Table III, an interesting observation is that, when λ_{Basic} takes an appropriate value (e.g., 0.5), changes in other hyperparameters have minimal impact on the KNNGuess's effectiveness. Even significant alterations in the values of KNN-TPG hyperparameters (e.g., $\lambda_{Basic} = 0.7$) result in superior attack effectiveness under attack scenario #4 compared with all other models. This demonstrates a degree of robustness in our method, indicating insensitivity to hyperparameter variations and resilience to minor perturbations in input data. This inherent stability allows the KNN-TPG to consistently maintain its robust effectiveness when confronted with various changes in the input dataset or different types of noise.

D. Supplementary analysis of training set changes

In real-world attack scenarios, attackers train password guessing models based on existing password pairs [56], although the number of password pairs that different attackers possess varies. In our experiments, we use password pairs obtained through email matching as the training set (e.g., Tianya → Dodonew). The sizes of training sets matched between two websites typically range from 10^5 to 10^6 (scenarios #1-#8), while mixed datasets generally range from 10^6 to 10^8. as shown in Fig. 10 and Fig. 12. With the increase in training set size, KNNGuess shows the steepest growth, which gradually slowed down after the training set size exceeded $5*10^6$. The slowest growth is observed in the TarGuess-II model, even when the training set size increases by a factor of 1,000, the model's effectiveness does not change significantly (deviation in cracking success rate under 1,000 guesses < 0.01%).

Comparing the effectiveness of the KNNGuess model with the TransGuess model (see in Fig. 12) reveals the advantage of KNN-TPG. TransGuess tends to exhibit the "Password overload drop effect" when dealing with large datasets, whereas the addition of the KNN-TPG method effectively mitigates this issue. When the model exhibits weak generalization or inadequate fitting, KNN-TPG effectively provides more training information to assist generation. The impact of KNN-TPG on model training is minimal because we only utilize the model's ability to map passwords into a high-dimensional vector space. This underscores the advantage of non-parametric methods: they possess stronger robustness and are not constrained by specific distribution assumptions.