This is a full version of our work that will appear in the proceedings of USENIX Security 2023.

Password Guessing Using Random Forest

Ding Wang, Yunkai Zou
Nankai University
{wangding, zouyunkai}@nankai.edu.cn

Zijian Zhang
Peking University
zhangzj@pku.edu.cn

Kedong Xiu
Nankai University
kedongxiu@nankai.edu.cn

Abstract

Passwords are the most widely used authentication method, and guessing attacks are the most effective method for password strength evaluation. However, existing password guessing models are generally built on traditional statistics or deep learning, and there has been no research on password guessing that employs classical machine learning.

To fill this gap, this paper provides *a brand new technical route* for password guessing. More specifically, we re-encode the password characters and make it possible for a series of classical machine learning techniques that tackle multi-class classification problems (such as random forest, boosting algorithms and their variants) to be used for password guessing. Further, we propose RFGuess, a random-forest based framework that characterizes the three most representative password guessing scenarios (i.e., trawling guessing, targeted guessing based on personally identifiable information (PII) and on users' password reuse behaviors).

Besides its theoretical significance, this work is also of practical value. Experiments using 13 large real-world password datasets demonstrate that our random-forest based guessing models are effective: (1) RFGuess for trawling guessing scenarios, whose guessing success rates are comparable to its foremost counterparts; (2) RFGuess-PII for targeted guessing based on PII, which guesses 20%∼28% of common users within 100 guesses, outperforming its foremost counterpart by 7%∼13%; (3) RFGuess-Reuse for targeted guessing based on users' password reuse/modification behaviors, which performs the best or 2nd best among related models. We believe this work makes a substantial step toward introducing classical machine learning techniques into password guessing.

1 Introduction

Passwords are likely to remain the dominant method in the foreseeable future due to its simplicity to use, easiness to change and low cost to deploy [12,13,22,30]. However, users tend choose popular strings, employ personally identifiable information (PII), and reuse an existing password. Such behaviors make passwords vulnerable to guessing attacks (including trawling guessing [11,47] and targeted guessing [44,63]).

To address this issue, service providers often employ a password strength meter (PSM) [15,66] to detect weak passwords, and research shows that well-designed PSMs do help users improve their password strength [54]. In practice, guess number is found to be a good metric to evaluate password strength [15,39], and those easily guessed by an attacker are considered weak passwords. Thus, it is imperative to study password strength from the attacker's perspective. While unending password data breaches [3, 6, 8] provide favorable material for attackers, there are still realistic attack scenarios (e.g., for e-banking sites, and for passwords from sites beyond USA, China, and Russia) where training data is scarce (e.g., size$\leq 10^6$), and/or when little is known about the target. Therefore, it is equally imperative to understand guessing threats when the available training data is not abundant.

In 1979, Morris and Thompson [40] designed several heuristic transformation rules to generate variants of dictionary words, and exploited them to perform password guessing. Since then, a series of trawling password guessing approaches that employ users' behavior of choosing popular passwords have been proposed, major ones are probabilistic context-free grammar (PCFG [65]), and Markov-based models [38,41]. Besides, frequent large-scale PII leaks (e.g., 240 million Deezer leak [53], 553 million Facebook leak [9] and 77 million Nitro PDF leak [25]) make targeted password guessing (e.g., Targeted-Markov [61] and TarGuess-I [63] that employ users' PII, and TarGuess-II [63] that employ users' sister passwords) more and more realistic. All these password guessing algorithms are *statistics-based* models that crack passwords by counting the frequency of elements in the training set (such as the letter segments in PCFG and the n-gram strings in Markov). These "simply counting" models have the inherent limitations of data sparseness and overfitting [38].

To address these limitations, *deep learning based* guessing models (e.g., RNN [39], PassGAN [31], Adams [46] and CPG/DPG [47]) have been proposed. They mainly use complex neural networks to process the short length and small feature dimension texts (i.e., passwords). While the model training only happens once for these models, they usually require extremely large training set (e.g., $>10^8$ for dynamic dictionaries [46]), but have no significant success-rate advantages over statistics-based guessing models (e.g., PCFG [65] and Markov [38]) within 10^{10} guesses [57].

Since Melicher et al. [39] first modeled password guessability using long short-term memory (LSTM) in 2016, sustained attention has been attached to applying deep learning to password guessing research [31, 46, 47]. It turns out that *password guessing research has bypassed classical machine learning and entered into the era of deep learning directly from the statistics-based period*, leaving a huge gap. In reality, as the development of statistical learning and also the foundation of deep learning, classical machine learning[1] techniques (e.g., support vector machine [42] and random forest [14]) have shown extensive applications in various fields like natural language processing (NLP), speech recognition and computer vision [33]. Compared with traditional statistical methods, classical machine learning algorithms usually have stronger fitting and predictive abilities; Compared with deep learning techniques, classical machine learning techniques usually have more concise models, entail easier parameter tuning, and require less training data to achieve satisfactory results.

However, to the best of our knowledge, no attention has been given to designing password guessing models based on classical machine learning techniques. Particularly, there has been no satisfactory answer to the following key questions: (1) Can classical machine learning techniques be used to design password guessing models? (2) If it is possible, how can these techniques be used for typical guessing scenarios? (3) Whether password guessing models based on classical machine learning techniques can improve the attacking success rate while reducing the computational overhead? In this paper, we aim to provide concrete answers to these key questions. Though applying classical machine learning techniques to password guessing looks deceptively simple, it is actually rather challenging. Now we explain why.

Firstly, passwords are essentially short texts and have the following characteristics that differ significantly from traditional NLP tasks: (1) A password is usually composed of 6∼30 characters [38, 59], which is much shorter than standard NLP texts; (2) A password is a piece of artificially constructed sensitive text, which may contain rich semantics, but it is not limited by (and often deliberately deviated from) the syntactic structure of the ordinary text, such as the password `loveu4ever` (with the semantic love you forever); (3) For password guessing, it is required that the generated passwords can *precisely* match the target. This means any inconsistency will lead to the failure of password cracking. For example, we take `P@ssword123` as the target and generate a series of guesses that are very close to it but different, such as `password123`, `p@sswrod123`, `Password123`, etc. Though they are all similar to the targeted password, none of them constitutes a correct guess. This is particularly concerned in guessing scenarios where the number of guesses allowed is limited, e.g., online guessing [63], while online guessing is the primary security threat that normal users need to devote efforts to mitigate (see [12, 24, 63]). In contrast, some amount of ambiguity is allowed in traditional NLP tasks, as long as the ambiguity does not significantly impair understanding. Hence, classical machine learning techniques originally designed for NLP tasks (or computer vision) cannot be directly or easily used for password guessing.

Secondly, password guessing models based on deep learning ([31, 39, 47]) usually use one-hot encoding for password characters, and use neural networks to learn the internal connections of these characters automatically. However, classical machine learning techniques usually require manually extracting and constructing features (i.e., feature engineering). Thus, it is a considerable challenge to tackle the question of how to accurately characterize passwords, so that we can not only reflect the inherent properties of the characters, but also ensure the effectiveness of the machine learning algorithm.

We summarize our contributions as follows:

- **A new technical route**. We represent each password character in an *n*-order (e.g., *n*=4, 5, 6) string in four dimensions: ⟨character type, the rank of the character (e.g., letter `a` is the first lower letter in `a`∼`z`), keyboard row number, keyboard column number⟩, and represent the entire *n*-order string in two additional dimensions: ⟨position of the character in a password, position of the character in the current segment⟩. These representations are *generic* and make the classic machine learning techniques (e.g., Random Forest and Boosting), for the first time, be successfully applied to password guessing.
- **A new PII matching algorithm**. To overcome the limitations of existing PII matching algorithms (i.e., using heuristic tags to represent PII usages in passwords [63]), we propose a new approximately optimal PII matching algorithm that more accurately captures users' PII usages, and can improve the success rates of leading guessing models by 7%∼13%. We show the effectiveness of our algorithm through both theory and experiments.
- **Extensive evaluation**. We perform a series of experiments to demonstrate the effectiveness and general applicability of our models. Results show that the guessing success rate of our RFGuess is comparable to its foremost counterparts in trawling guessing scenarios, and is 7.03%∼27.54% higher than its counterparts in targeted guessing scenarios based on PII.
- **Some insights**. When predicting the next character after the *n*-order strings in a password, RFGuess can clearly show the importance of each character in different dimensions (e.g., type/continuity/position-information). Such knowledge can help us optimize the model training and

[1]For simplicity of presentation, the term "machine learning" that appears in this work stands for the "classical machine learning".

password generation time by making easier the detection (and elimination) of password features with low importance, and also sheds light on the design of new machine learning based guessing models.

2 Background and related work

2.1 Three guessing scenarios

Trawling guessing. Trawling guessing means that the attacker does not care who the specific target is, and its only goal is to guess more passwords under the guess number allowed. In 2009, Weir et al. [65] proposed a fully automated password guessing algorithm based on a rigorous probabilistic context-free grammar (PCFG). First, the algorithm divides a password string into three categories: letter segment L, digit segment D, and special character segment S. Then, the password is transformed into a template structure (e.g., `Password123!`$\rightarrow$ $L_8D_3S_1$) and the corresponding terminals that fit into the structure (e.g., $L_8$$\rightarrow$Password). Finally, the probability of a generated password is calculated according to the probability of its structure multiplied by those of its terminals. In this context, researchers have proposed a series of improved techniques, such as performing further semantic mining in passwords [56], adding keyboard and multiword patterns [32] and adaptive improvement for long passwords [27].

Unlike PCFG [65], which divides passwords into different segments according to the character type, the Markov model proposed by Narayanan and Shmatikov [41] trains the whole characters in a password, and calculates the probability of passwords through the connection between the characters from left to right. Particularly, the n-gram Markov needs to record the frequency of a character followed by a string of length n-1. Like PCFG, many researchers have conducted follow-up research on the Markov model. For example, Ma et al. [38] smoothed and normalized the Markov model to alleviate the problem of data sparseness and overfitting; Markus et al. [21] enumerated the passwords in descending probability order to improve the guessing speed.

At USENIX'16, Melicher et al. [39] first introduced deep learning techniques to password guessing. More specifically, they build a neural network composed of LSTMs (which are denoted as FLA, i.e., Fast, Lean, and Accurate). Compared with the traditional statistical password guessing models (e.g., PCFG [65] and Markov [38]), FLA has better cracking rate under relatively large guesses (i.e., $>10^{10}$). In 2019, Hitaj et al. [31] introduced generative adversarial networks (GAN) to password guessing, and proposed the PassGAN, which shows the potential of GAN's application in this field. After that, Pasquini et al. [47] alleviated the mode collapse problem of GAN during training, so that the cracking rate of GAN-based approaches under large guesses has been significantly improved. On this basis, they constructed two password guessing frameworks, that is, conditional password guessing (CPG) and dynamic password guessing (DPG). However, compared with statistics-based models (e.g., PCFG [65] and Markov [38]), CPG/DPG [47] generally requires extremely large training data (e.g., size$>10^7$), consumes longer training time, and suffers cumbersome parameter tuning.

Targeted guessing based on PII. The goal of a targeted password guessing is to crack the password of a given user in a given service (e.g., an online banking account, and personal mobile phone) as quickly as possible [63]. Thus, the attacker would use PII related to the target victim to enhance the pertinence of cracking. Overall, the current research on targeted password guessing is still in its infancy, mainly focusing on how to use demographic information (such as name, birthday and mobile phone number). In 2015, Wang and Wang [61] first proposed a targeted guessing model based on Markov [41] (namely Targeted Markov). Their basic idea is that the frequency of names in the training sets reveals the likehood of the targeted user choosing a name-based password. In 2016, Li et al. [36] proposed a targeted guessing model based on PCFG [65]. The difference with trawling PCFG is that some PII segments representing different lengths have been added to the original LDS segments. At CCS'16, Wang et al. [63] revealed the inherent limitation of length-based PII matching method, and proposed a new targeted guessing model with a *type-based* PII matching method, namely TarGuess-I.

Targeted guessing based on password reuse. At NDSS'14, Das et al. [19] proposed the first cross-site password-guessing algorithm based on transformation rules. This algorithm performs several artificially defined transformations (e.g., delete, insert, and leet) on users' existing passwords, and then generates guesses in a pre-defined order. However, users would hardly reuse/modify passwords in such a pre-defined unified approach, hence limiting its performance in the real world.

At CCS'16, Wang et al. [63] proposed a PCFG-based model for password reuse, namely TarGuess-II. The core idea is to characterize users' password reuse behaviors in two levels of modification operations (i.e., structure- and segment-level). During training, it first learns the probability of the two-level transformation path of sister password pairs to build a PCFG. Second, the guess with the highest probability in the PCFG is output each time through the priority queue, and then they are transformed and inserted into the priority queue again. In this way, the guesses sorted by probability can be obtained.

At IEEE S&P'19, Pal et al. [44] proposed Pass2Path, a targeted guessing model for password reuse based on deep learning. Specifically, it employs a sequence-to-sequence (seq2seq) model [52] to predict the path of modifications needed to transform one password into its sister password. Its guessing success rate is better than that where the input and output of the model are directly the user's original password and the new password, respectively. In other words, this way of training focuses the model better on learning common transformations found in password datasets. We have analyzed the problems in existing password models in Appendix D.

2.2 Password guessing modeling

The Markov n-gram model was originally introduced at CCS'05 [41] and improved at IEEE S&P'14 [38]. Generally, n is recommended to be 3, 4, or 5 [38,62]. Its core assumption is that: Each character is only related to the first d characters in front of it and has nothing to do with other characters, where $d(=n+1)$ is the order of the Markov model. The conditional probability for character c_i following the string $c_1c_2\ldots c_{i-1}$ is

$$\begin{aligned}\Pr(c_i|c_1c_2\cdots c_{i-2}c_{i-1}) &= \Pr(c_i|c_{i-d}\cdots c_{i-1})\\ &= \frac{\texttt{Count}(c_{i-d}\cdots c_{i-1}c_i)}{\texttt{Count}(c_{i-d}\cdots c_{i-1}\cdot)},\end{aligned} \tag{1}$$

where $\texttt{Count}(c_{i-d}\cdots c_{i-1}c_i)$ denotes the number of occurrences of the string $c_{i-d}\cdots c_{i-1}c_i$, and $\texttt{Count}(c_{i-d}\cdots c_{i-1}\cdot)$ denotes the number of occurrences of the string $c_{i-d}\cdots c_{i-1}$ where it is followed by an undetermined character (i.e., where it is not at the end of a string). Then the probability of the string $s{=}c_1c_2\cdots c_n$ is given by:

$$\begin{aligned}\Pr(s) &= \Pr(c_1)\Pr(c_2|c_1)\cdots\Pr(c_n|c_{n-1}c_{n-2}\cdots c_1)\\ &= \prod_{i=1}^{n}\Pr(c_i|c_{i-d}\cdots c_{i-1}).\end{aligned} \tag{2}$$

In reality, while each character in a password may have varying degrees of security impact on other characters [45], this paper assumes that the order in which users create passwords is from left to right (i.e., the same order with how they type passwords), and each character is only related to a few characters before it (This means our model makes the same assumption with the well-known Markov model [38,41]). Under this assumption, the password generation process can be regarded as a *multi-class classification problem.*

More specifically, given a password, the n-order string in the front of its character can be seen as the target to be classified (and features can be extracted from this n-order string), and the character itself can be seen as the classification label corresponding to the string to be classified. From this perspective, all supervised machine learning algorithms that tackle multi-classification problems can be applied to password guessing. Considering that when the data dimension is low and the task accuracy required is high (which are exactly the characteristics of password guessing tasks), ensemble learning methods generally performs well (see the potential applicability of some representative classification algorithms like SVM [42] and Boosting [50] in Appendix A). Without loss of generality, in what follows, we take Random Forest as a typical case study to show how to employ classical machine learning techniques for password guessing.

Assume that $\mathbf{T}=\{(x_1,y_1),(x_2,y_2),\ldots,(x_n,y_n)\}$ is the training set, then we can build a mapping f from the input space $\mathbf{X}$ to the output space $\mathbf{Y}$ by learning $\mathbf{T}$. Here $\mathbf{X}=\{n\text{-order strings of a password set}\}$, $\mathbf{Y}=\{95 \text{ printable ASCII codes}\}\cup\{E_s\}$, i.e., 96 different categories, where E_s denotes the end symbol.

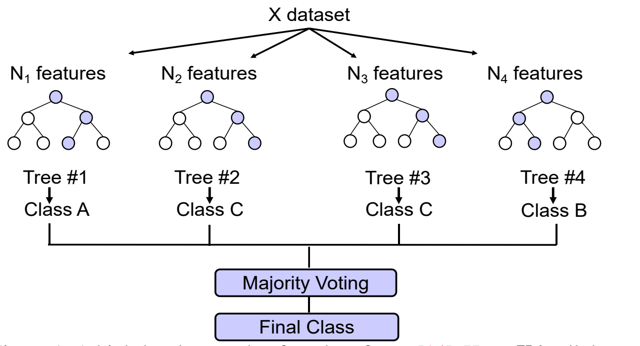


Figure 1: A high-level example of random forest [14]. Here, $\mathbf{X}$ is all the n-order strings of a training dataset, $N=\{N_1,N_2,\ldots,\}$ is a randomly extracted feature subset, and class {A, B,...} represents the category (95 printable ASCII codes and the end-symbol) to which each n-order string belongs.

2.3 Introduction of random forest

Random forest [14] is a an ensemble learning method for classification (and regression) that consists of multiple decision trees [49]. When predicting the category of a sample, the algorithm counts the prediction results of each tree in the forest, and then selects the final result by voting (see Fig. 1). "Randomness" lies in two aspects: the random selection of features and the random selection of samples. Hence, each tree in the forest has both similarities and differences.

Formally, we denote the decision tree model as $\{h_k(\mathbf{X}), k=1,2,3,\ldots\}$. Given an independent variable x in $\mathbf{X}$ dataset, each decision tree has one vote to select the optimal classification result. The final classification decision is:

$$\mathrm{H}(x) = \arg\max_{y\in\mathrm{Y}} \sum_{i=1}^{k}\mathrm{I}(h_i(x)=y), \tag{3}$$

where $\mathrm{H}(x)$ represents the combined classification model (i.e., the random forest), h_i is a single decision tree, $y\in\mathrm{Y}$ is the output, and $\mathrm{I}(\cdot)$ is the characteristic function.

The decision tree [49] has three mainstream node splitting algorithms: ID3, C4.5 and CART. These algorithms use different feature selection criteria, namely, information gain, gain ratio and Gini impurity. Among them, the Gini impurity represents the probability that two samples are randomly selected from the dataset and their categories are different. It has a relatively small calculation cost and is easy to understand. Therefore, this paper uses the CART decision tree (see Fig. 2). For a dataset D (composed of n-order strings), the calculation formula of the Gini impurity is as follows:

$$\mathrm{Gini}(D) = \sum_{k=1}^{|y|}\sum_{k'\neq k} p_k p_{k'} = 1-\sum_{k=1}^{|y|} p_k^2, \tag{4}$$

where $|y|$ represents 96 classification categories (i.e., 95 printable characters and the end-symbol E_s), and p_k represents the proportion of the category k in D. When dividing features, the Gini impurity of feature a (the detailed feature construction method of n-order strings can be seen in Sec. 3.1) is:

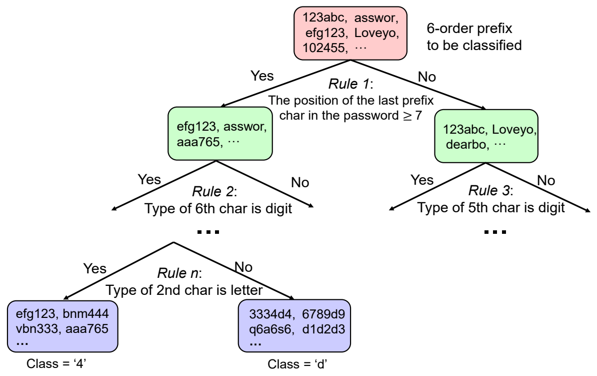

123abc, asswor, efg123, Loveyo, 102455, ⋯
6-order prefix to be classified
Yes
No
Rule 1: The position of the last prefix char in the password ≥ 7
efg123, asswor, aaa765, ⋯
123abc, Loveyo, dearbo, ⋯
Yes
No
Rule 2: Type of 6th char is digit
Yes
No
Rule 3: Type of 5th char is digit
Yes
No
Rule n: Type of 2nd char is letter
efg123, bnm444 vbn333, aaa765 …
3334d4, 6789d9 q6a6s6, d1d2d3 …
Class = '4'
Class = 'd'

Figure 2: A high-level example of a decision tree for password prefix classification. The node division is determined according to the corresponding rules through the if-else logic, and finally, each prefix is divided into the character category to which it belongs. For example, class=`4` means that all prefixes in this leaf node are followed by the character `4` like `efg1234`.

$$\mathrm{Gini}(D|a) = \sum_{v=1}^{V} \frac{|D^v|}{|D|} \mathrm{Gini}(D^v), \tag{5}$$

where v represents each value of feature a, and D^v represents the subset of D divided according to the value v. Formula 5 indicates that when selecting features, the weighted average method is used to calculate the total Gini impurity, and finally, the feature that minimizes the Gini impurity after division is selected as the optimal division feature.

2.4 Analysis of random forest

Now we explain why the random forest model [14] can solve the shortcomings of the Markov n-gram model [41] when applying to password guessing from three aspects.

The fitting principle. Fig. 3 shows that the Markov n-gram model [41] can essentially be seen as a decision tree [49] divided by prefix string of height one. It divides all strings with the same prefix into the same leaf node, resulting in that when the prefix string appears very rarely in the training set (i.e., data sparseness issue), it can only be classified according to the few samples that appear in the training set. In comparison, the decision tree divides its node according to the *impurity* (representing how well the trees split the data, and there are several impurity measures like the Gini impurity as defined in Sec. 2.3.) of each feature (i.e., division rules) in the prefix. It selects the feature with the least impurity as the rule of feature division, which makes the final sample meeting the same division rule fall to the same leaf node. These samples can be regarded as similar samples with the same classification results (see Fig. 2 for a high-level example). Thus, the prefixes that appear less frequently or do not appear in the training set can also be divided into leaf nodes composed of similar samples.

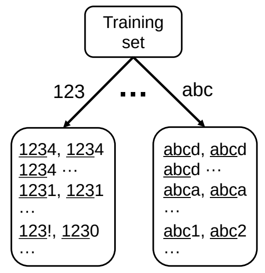


Figure 3: Tree diagram of the Markov model [41]. Here we take 3-order as an example.

Automatic feature screening. The most critical parameter in the Markov model [41] is the order d, which is the length of the prefix that needs to be considered. When the order is too high, the model is easy to overfit [38]. However, this is not a problem for the random forest [14]. Specifically, the decision tree [49] selects the feature with the smallest impurity (after splitting) as the division rule. That is, it will select features with a higher degree of importance for division, and will not be affected by those with poor division effects. For example, the string `1234` followed by the character `5` is a natural law and should not be changed due to the training set, so the substring `1234` is more important than the whole string `#1234` when predicting the next character, while the 5-order Markov model only considers the frequency of `#1234`.

Minimum number of samples in each leaf node. For samples that appear less frequently or have never appeared before, the decision tree [49] will divide the samples into a specific leaf node according to the training rules. Since the samples that meet the same set of rules have the same classifications, the probability of each category will be obtained according to the distribution of the sample categories in the leaf nodes. When the number of samples in the leaf node is large, the decision tree [49] can smooth the samples well; when the number of leaf nodes is small, or even if there is only one sample in a leaf node, this situation degenerates to the Markov model with the low-frequency prefix problem. Fortunately, the decision tree can reasonably solve this problem by limiting the minimum number of samples in leaf nodes.

3 RFGuess: A new trawling guessing model based on random forest

3.1 Password character feature construction

To construct a password guessing model based on classical machine learning techniques, feature engineering is an essential step. A password is usually composed of characters and has two important characteristics: the type of characters and the continuity of characters. Particularly, there are three types of characters used in passwords: digits, letters, and special characters. Each type of character has a certain internal continuity, such as 0∼9 and a∼z. Therefore, to well represent these two characteristics of characters in a password, we need to *re-encode* the password characters.

We first represent the password characters in two dimensions. The first is the character type. Here we use 0, 1, 2, and 3 to represent special characters, digits, uppercase letters, and lowercase letters, respectively; the second is the serial number of the characters in each type. For example, letters `a`∼`z` are represented by 1∼26, digits `1`∼`9` represented by number 1∼9, and digit `0` represented by 10 (since 0 stands for the beginning symbol). In this way, the type and continuous characteristics of password characters can be displayed explicitly.

Secondly, keyboard pattern is a popular way to create passwords [63, 68], so the characteristics of keyboard pattern are also considered in the feature construction. The keyboard pattern usually means creating passwords through adjacent keyboard positions, such as `1qa2ws` and `123qwe`. Thus, we also use two-dimensional features to represent the keyboard characteristics of password characters: the row and column position of the keyboard in the form of coordinates. For example, 1 is represented as $(1,1)$, q is represented as $(2,1)$, s is represented as $(3,2)$. Thus, the position coordinate representation can clearly show the continuous characteristics of the characters and improve the model fitting ability.

The last consideration is the length characteristic of the string. More specifically, we construct two length features: position of the character relative to the entire password (i.e., trained length) and position of the character relative to the current segment (i.e., trained length in the current L/D/S segment). Considering that the password length of most users is at least six [38, 59], we set the order of our model to six. That is, we use a 6-order prefix to predict the next character.

As a result, every 6-order prefix can be represented by a 26-dimensional feature vector (26=4×6+2), because there are 6 prefix characters, 4 feature dimensions for each character and 2 additional feature dimensions for the length information of the entire prefix. We take the 6-order prefix `wer654` of password `qwer654321` as an example. First, each character in `wer654` can be *uniquely* represented as a 4-dimensional feature vector. For instance, character `r` is represented as (3, 18, 2, 4), where 3 represents the character type of lowercase letter, 18 is the rank of `r` in the lower letter sequence `a`∼`z`, 2 and 4 are the keyboard row and column positions of `r`, respectively. Now, `wer654` can be represented by a 24-dimensional feature vector (24=4×6). Then, we add 2-dimensional length feature (7, 3) of prefix `wer654`, where 7 represents the position information relative to the entire password (i.e., digit `4` in `wer654` is the 7th character of `qwer654321`), and 3 represents the position information relative to the current digit segment (i.e., digit `4` in `wer654` is the 3th character of segment `654321`).

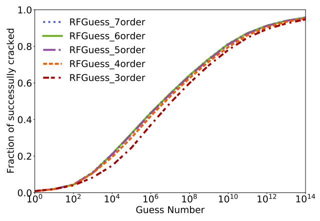


Figure 4: Impacts of varied orders (1M Rockyou→Rockyou_rest).

Note that, we have tested a number of different order values (i.e., n=3, 4, 5, 6, 7) of our RFGuess, and found that when $n \geq 4$, RFGuess can achieve similar cracking success rates (as shown in Fig. 4). Generally, when the order decreases (e.g., n=3), the number of features (i.e., $4n+2$) decreases accordingly, which may make RFGuess underfit. While when the order is too large (e.g., $\geq$7), passwords whose length is smaller than this value cannot be well modeled. Since the cracking success rate is slightly better when n=6, we set n=6 in the following experiments.

3.2 Feature importance analysis

To verify the effectiveness of the constructed features, the feature importance scores are calculated by random forest in different training datasets. According to the results shown in Figs. 11, 12, and 14 (in Appendix H), we find that the trained length feature (i.e., position of the last prefix character in the password) has the highest score, indicating its significant impact on fitting RFGuess. Other top-ranked features are mainly characters that are closer to the target character, which is intuitive. Among the four different dimensional features of the same character, the serial number feature (e.g., `a` is the first in alphabetic types `a`∼`z`, 0 is the first of digits 0∼9) and the keyboard column number feature are more effective, while the type of character (letter/digit/special character) and the trained length of current segment features (position of the character in the L/D/S segment) are relatively unimportant, and keyboard row number offers little gain on the model fitting. A plausible reason is that when building passwords, users create more horizontal keyboard modes (e.g., `qwerty`) than vertical modes (e.g., `1qaz`) [68]. Particularly, we have counted the Top-10 most frequency keyboard patterns of CSDN, Dodonew, Taobao, and Rockyou, and found that the Top-10 patterns are either horizontal keyboard modes or just the repetition of a single character (e.g., `aaaaaa`).

3.3 Model training and password generation

At a high level, our RFGuess is similar to the Markov 7-gram model [41]. More specifically, it first processes the password into the form of 6-order character prefixes and their corresponding characters (e.g., the resulting 6-order set of password `abc123` is $\{(B_sB_sB_sB_sB_sB_s,a),(B_sB_sB_sB_sB_sa,b),...,(abc123,E_s)\}$, where B_s and E_s stand for the beginning and ending symbol respectively). Then, it represents the 6-order prefix as a 26-dimensional vector (each character is represented by 4 dimensions, plus two additional length features for the entire prefix), the single character following this prefix in an ASCII value, and the beginning and ending symbol are represented by 0 and -1, respectively. When training the model, RFGuess traverses the 6-order set of each password in the training set, takes the 26-dimensional prefix feature vectors as training input, and takes the numerical label of the corresponding characters as training output. Fig. 2 shows a high-level example of the decision tree classification process.

The process of guess generation is quite similar to the Markov n-gram model [38]. The key difference is that we use the trained random forest (but not the Bayes formula) to calculate the conditional probability. More specifically, each decision tree will vote on which category (one of the 95 characters and end-symbol) the input sample (i.e., the 6-order string prefix) belongs to, and its probability is the proportion of the number of votes obtained by this category to the total number of trees. For example, suppose there are 10 decision trees in the random forest voting on the string prefix

Table 1: Basic information about our 13 password datasets.†

Dataset	Web service	Language	When leaked	Total PWs	Length>30	Removed %	Unique PWs	With PII
Taobao	E-commerce	Chinese	Feb., 2016	15,072,418	88	0.01%	11,633,759	
126	Email	Chinese	Oct., 2015	6,392,568	621	0.23%	3,764,740	
Dodonew	E-commerce	Chinese	Dec., 2011	16,283,140	13,4758	0.15%	10,135,260	
CSDN	Programmer	Chinese	Dec., 2011	6,428,632	0	0.01%	4,037,605	
Wishbone	Social	English	Jan., 2020	10,092,037	250	0.01%	5,933,902	
Mate1	Dating website	English	Mar., 2016	27,401,505	12,430	0.06%	11,916,080	
000Webhost	Web hosting	English	Oct., 2015	15,299,907	4,159	0.76%	10,526,769	
Yahoo	Web portal	English	July, 2012	453,491	0	2.35%	342,510	
LinkedIn	Job hunting	English	Jan., 2012	54,656,615	17,162	0.22%	34,282,741	
Rockyou	Social forum	English	Dec., 2009	32,603,387	3,140	0.07%	14,326,970	
12306	Train ticketing	Chinese	Dec., 2014	129,303	129,303	0	117,808	✓
ClixSense	Paid task platform	English	Sep., 2016	2,222,045	0	0	1,628,018	✓
Rootkit	Hacker forum	English	Feb., 2011	69,330	5	0.01%	56,835	✓

†PW stands for password, and PII for personally identifiable information. We clean up passwords longer than 30 or containing non-ASCII characters.

"123456". Among them, 6 votes are cast for the character "7", 3 votes for the character "a", and 1 vote for the character "6". Then the probabilities of the entire string (6-order prefix plus a single character) are {"1234567": 0.6; "123456a": 0.3; "1234566": 0.1}. See more details in Appendix E.

Our exploratory experiment shows that in the process of generating 1.8 million (M) guesses (training with 6M CSDN passwords using 30 decision trees), an average of 70% of the characters do not get a vote in one-step prediction, constituting the majority of the alphabet. This indicates that RFGuess may not be good at generating previously unseen characters.To address this issue, we employ the add-δ smoothing technique [38] to smooth characters that do not get a vote. For example, $\Pr(\#|123456) = \frac{\Pr_{\text{RFGuess}}(\#|123456)+\delta}{1+\delta\cdot|\Sigma|}$ (where $\Pr_{\text{RFGuess}}(\#|123456)$ means the probability of # calculated by RFGuess when the input string is 123456, and Σ is the character table of the training set). We have tested a number of values of δ (e.g., 0, 0.01, 0.02, 0.001), and found δ=0.001 is the best among all (see Fig. 5).

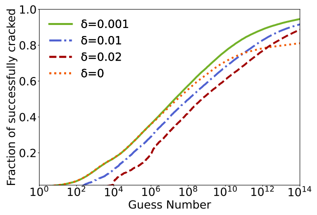


Figure 5: Impacts of varied values of δ (0.1M Rockyou→Rockyou_rest).

3.4 Experimental setups and results

Datasets. We evaluate the existing password guessing approaches and our RFGuess model based on 13 large real-world password datasets (see Table 1), a total of 241.27 million(M) passwords. Eight of our password datasets are from English sites and five from Chinese sites. As Table 1 shows, three datasets (i.e., 12306, ClixSense and Rootkit) are originally associated with various kinds of PII (e.g., name, birthday, email). To enable extensive targeted guessing evaluation, we match the *non*-PII-associated password datasets with these three PII-associated ones through email, and this produces a total of six PII-associated password datasets (see Table 2). For targeted guessing based on password reuse, we obtain eight password pair datasets by matching email (see Table 4).

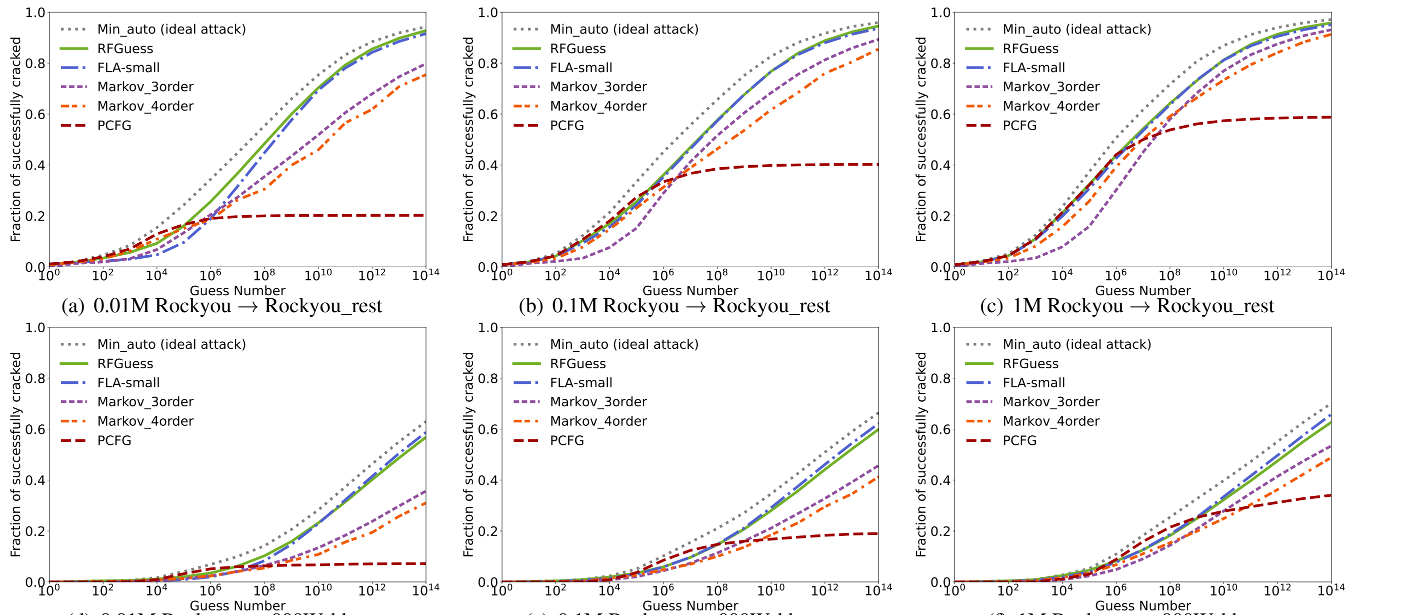
Ethical considerations. Though ever publicly available on the Internet and widely utilized in existing studies [19, 44, 46, 47, 63], these datasets are private data. Hence, we only illustrate the aggregated statistical information and keep each individual account as confidential in order to avoid bringing additional risks to the corresponding victim. While these datasets may be misused by attackers for cracking, our use is both beneficial for the academic community to understand the strength of users' password choices and for security administrators to prevent creating weak passwords. As our datasets are widely used and publicly downloadable on the Internet, this facilitates fair comparison and good reproducibility.

Experimental setup. To well establish the generality and effectiveness of our RFGuess, we evaluate it on both one-site (intra-site) and cross-site guessing scenarios. For intra-site scenarios, we randomly select 0.01M, 0.1M, and 1M (M=million) passwords from Rockyou as the training set, respectively, and randomly select 100,000 passwords from the remaining dataset as the test set. Since the attacker is smart and will constantly improve her training set to make it as close as possible to the test set (to improve her success-rates), our intra-site experimental methodology just reflects this situation (this methodology is quite routine in password research [60, 63, 65]). Particularly, many sites (e.g., Yahoo [43], Flipboard [4], Twitter [29] and Anthem [23]) have leaked their user passwords more than once, and thus it's practical/realistic to conduct/consider the intra-site guessing scenarios. For cross-site scenarios, we apply the trained model (on an older leaked dataset) to crack a newer leaked dataset (i.e., Rockyou→000Webhost and 000Webhost→Wishbone). Note that we do not remove the identical password pairs (i.e., direct reuse) that occur in the training sets from the test sets, because the attacker has no prior knowledge of which passwords are used by the target account, and excluding duplicate passwords from the test set hinders the evaluation of a guessing model's fitting ability. We discuss this point in detail in Sec. 6.

We compare RFGuess with three leading guessers (i.e., PCFG [65], Markov [38], and FLA [39]). Also, we introduce the Min_auto approach [55] to avoid the bias of a single approach. The setups of each approach are as follows.

PCFG. The PCFG we use is consistent with [38], that is, the probability of the L segment comes from the training set, which is better than the original version in [65].

Markov. For the Markov model, due to the great influence of order, this paper carries out the 3-order and 4-order experiments at the same time, and adopts the Laplace smoothing



(a) 0.01M Rockyou → Rockyou_rest (b) 0.1M Rockyou → Rockyou_rest (c) 1M Rockyou → Rockyou_rest

(d) 0.01M Rockyou → 000Webhost (e) 0.1M Rockyou → 000Webhost (f) 1M Rockyou → 000Webhost

Figure 6: Guessing performance of our RFGuess in comparison with other approaches (i.e., PCFG [65], 3/4-order Markov [38], FLA [39] and Min_auto [55]) in the intra-site and cross-site trawling guessing scenarios. Note that Min_auto [55] represents an *idealized* strategy: A password is considered cracked as long as any of these five real-world password models cracks it. Rockyou_rest means the original Rockyou dataset excluding the corresponding training set.

and end symbol regularization as used in [38].

FLA. We use the source code of FLA [39], and follow its recommended parameters in our experiments. More specifically, we train a model consisting of three LSTM layers with 200 cells (namely the "small" model in [39]) in each layer and two fully connected layers, a total of 20 epochs.

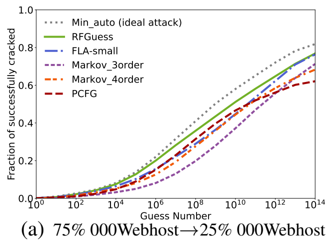
RFGuess. As detailed in Appendix B, we train a random forest with 30 decision trees. Its minimum number of leaf nodes is 10, the maximum ratio of features is 80%, and the rest are in default of the scikit-learn framework [2].

Min_auto. It represents an idealized guessing approach [55], in which a password is considered cracked as long as any of these real-world guessing models cracks it.

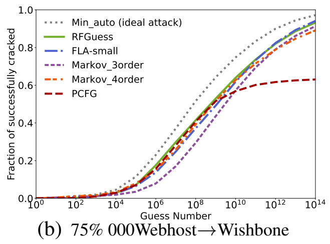
Experimental results. Considering that the way of enumerating large guesses is computationally intensive, we use the Monte-Carlo algorithm [20] to estimate a password's guess number. That is, how many guesses it would take for an attacker to arrive at that password when password guesses are attempted in descending order of likelihood. Fig. 6 shows the results. To accurately show the attack success rates of different approaches, we give the concrete result values at some specific guess numbers (i.e., 10^7 and 10^{14}; see Table 10 in Appendix C). In intra-site guessing scenarios, our RFGuess performs slightly better than FLA [39], and beats PCFG [65] and Markov [38] beginning at around 10^7 guesses. In cross-site guessing scenarios, the guessing success rates of our RFGuess are slightly worse than FLA [39] within 10^{14} guesses, but are significantly higher than PCFG [65] and Markov [38].

To demonstrate the generality of RFGuess, we evaluate it with larger training datasets (i.e., 75% 000Webhost of size 11,474,930). Fig. 7 shows that, when using a ten million-sized training set, RFGuess outperforms all its counterparts in intra-site guessing scenarios, and is slightly better than (or comparable to) its counterparts in cross-site guessing scenarios. This suggests that RFGuess is better at modeling the guessability of passwords from the same (or similar) distribution. By employing the same training and test set (i.e., 75% of 000Webhost→25% of 000Webhost), we also compare RFGuess with dynamic dictionaries [46]. However, the success rate of dynamic password guessing (DPG) [46] is only 0.13% within 5×10^9 guesses (which are the maximum number of guesses that can be reached using 75% of 000Webhost). A plausible reason is that DPG is more suitable for extremely large training sets, and this partially explains why the original paper [46] uses the 1.4 billion-sized 4iQ as its training set. Our RFGuess is just on the opposite: It is particularly suitable for guessing scenarios where the training data is not abundant (e.g., passwords from sites beyond USA, China, and Russia).

We further make an apples-to-apples performance comparison of these approaches in three key criteria (i.e., training time, model size, and time to generate guesses), and summarize the comparison results in Table 7 (see Appendix C for details). In all, RFGuess has relatively high training efficiency (it only takes 0.3 hours to train five million data), but it has relatively large model size (i.e., 4.5G when the compress parameter in the joblib tool is set to three), and its guess generation is relatively slow (about 130∼677 passwords/s). This makes RFGuess particularly suitable for online password guessing attacks where the number of guesses allowed is small. In practice, online password guessing is the



(a) 75% 000Webhost→25% 000Webhost



(b) 75% 000Webhost→Wishbone

Figure 7: Evaluate our RFGuess using 75% 000Webhost of size 11,474,930.

most concerning (and unmatured) scenario regarding password security [7,44,63], because offline guessing can be well eliminated by slow/memory-hard hashes (e.g., Bcrypt and Argon2), but online guessing is unavoidable and its success-rate is rather high (see Tables 3 and 5) even if there are rate-limiting/blocking mechanisms. This is because the guess number allowed for an attacker cannot be too small, otherwise the system will suffer from DoS attacks, which explains why 100 in one month is recommended by NIST-SP800-63B [26].

If one wants to improve the password generation efficiency of RFGuess, she can set the number of trees to one (i.e., use the decision tree model). At this time, the password generation speed can be increased to 1,520 passwords/s, while the attack success rate is reduced by about 0.4%∼2% (see Fig. 9).

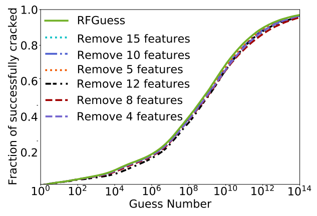


Figure 8: Impacts of varied features on RFGuess (Taobao→Taobao_rest).

Insights. To understand the impacts of features, we remove the relatively unimportant 5-, 10-, and 15-dimensional features according to the feature importance ranking, and remove 4-, 8-, and 12-dimensional features according to the character position information (e.g., the 4-dimensional features of character `123` in prefix `123456` are removed in turn). Results show that the training time and password generation speed of our RFGuess are improved by up to 35%, while the success rates remain stable (see Fig. 8).

Thus, when designing new password guessing models based on classical machine learning techniques, one can create as many new features as possible (e.g., the number of character types contained in the prefix, the Shannon entropy of the prefix, etc.) to explore more effective password representation. Then, the most effective features can be figured out by measuring the feature importance score and/or success rates. This improves the training efficiency while maintaining the success rates, which makes our RFGuess highly scalable.

4 RFGuess-PII: A targeted password guessing model based on PII

We now use random forest [14] to design a targeted password guessing model based on PII, called RFGuess-PII. We first analyze the limitations of the PII matching strategy used in current targeted guessing models, and then propose a more effective PII matching algorithm. Based on this algorithm and the RFGuess model in Sec. 3, we propose RFGuess-PII and demonstrate its effectiveness through large-scale experiments.

4.1 Problems in mainstream methods

Previous PII matching methods. Li et al. [36] first proposed a PII matching method similar to PCFG [65] (for example, N_4 represents name information with a length of four like `Wang`). At CCS'16, Wang et al. [63] pointed out that this method has severe limitations. Instead, they introduced a series of *type-based* PII tags and achieved drastically better results. More specifically, they use N standing for name usages, while N_1 for the usage of full name, N_2 for the abbr. of full name,$\cdots$; U stands for username usages, U_1 for full username, U_2 for the letter segment of the user name,$\cdots$. We summarize these notations in Table 6 of Appendix C.

In the process of training, the leftmost and longest matching strategy is adopted for disambiguation when matching the PII contained in the passwords. For example, if a user's username is `Alice0102`, name is `Alicexxx`, birthday is `19930102`, and password is `Alice01021993`, then according to the leftmost and longest matching strategy used in [63], it should be represented as U_1B_5 instead of N_3B_2 (where B_5 represents the birthday year, N_3 represents the full name of the surname, and B_2 represents the birthday in the MY format), because the username `Alice0102` will be matched first.

This matching strategy uses a greedy strategy to first match the longest PII at the leftmost position, and it is not optimal. "Optimal" here refers to the *global* optimum for the entire training password set rather than the local optimal for a single password. To explain the concept of global optima more clearly, we introduce information entropy for analysis. The Shannon Entropy [51] metric is proposed in 1948 to measure the uncertainty of a distribution. The greater the information entropy, the more random the password distribution, and the more secure the password set. Thus, for the same password set, the feature extraction and representation method that makes the password set's information entropy lower can better make use of the characteristics of the training set.

4.2 New PII matching algorithm

The current strategy for PII matching is not optimal because there will be ambiguities (multiple representations for the same password) when matching, and as in the above example, using the leftmost and longest matching strategy would result in heuristically selecting one option for PII tagging. This cannot minimize the information entropy. In other words, it cannot entirely and accurately extract the PII usage behavior of the entire user group. To address this issue, we propose an approximately optimal PII matching algorithm.

The first step of our proposed algorithm is similar to the type-based PII matching method [63], which subdivides the various possible transformations of PII and use different tags

Table 2: Basic information about our PII datasets.

Dataset	Language	Items num	Types of PII useful for this work
PII-12306	Chinese	129,303	Email, User name, Name, Birthday, Phone
PII-CSDN	Chinese	77,216	Email, User name, Name, Birthday, Phone
PII-Dodonew	Chinese	161,517	Email, User name, Name, Birthday, Phone
PII-ClixSense	English	2,222,045	Email, User name, Name, Birthday
PII-000Webhost	English	79,580	Email, User name, Name, Birthday
PII-Rootkit	English	69,418	Email, User name, Name, Birthday

to represent them. Notably, we use digital tags instead of letter tags (e.g., $N_1 \sim N_7, B_1 \sim B_{10}$ in TarGuess-I [63]), and summarize these notations in Table 6 of Appendix C. Thus, they can be conveniently used as input to the machine learning model for training. For example, starting from 1,000 to stand for name usages, where 1,000 for the usage of full name, 1,001 for the lowercase letter of last name,$\cdots$.

The second step is to list all the possible representations with PII tags for each of the passwords in the training set (e.g., three representations {4000, 2001}, {4001, 2003, 2004, 2001} and {1002, 2003, 2004, 2001} for `Alice01021993`). After that, we sort the representations by frequency from high to low. Specifically, the most frequent representation (e.g., {4000, 2001}) is denoted as R_1, the second is denoted as R_2 (e.g., {4001, 2003, 2004, 2001}),$\cdots$. Then, we use R_1 to represent all passwords that can be represented as R_1, and the frequency of each of their remaining representations (e.g., {4001, 2003, 2004, 2001} and {1002, 2003, 2004, 2001}) subtracts one. Next, the remaining passwords (remove those already represented by R_1) that can be represented as R_2 are all represented by R_2, and their frequency of the remaining representations continues to subtract one. The process repeats until the frequency of all remaining representations is less than or equal to one. Finally, the password whose representation has not been determined is represented by the shortest structure, and the algorithm ends. We formalize this process in Algorithm 2, and demonstrate its generality and effectiveness both theoretically and experimentally (see Appendix F).

4.3 New targeted guessing model based on PII

Based on RFGuess proposed in Sec. 3 and the approximately optimal PII matching algorithm, we now propose a new targeted password guessing model RFGuess-PII. The password training and generating process is similar to the trawling guessing scenario. The difference is that the PII string in the password is replaced with the corresponding digital tag through PII matching, and then the password set containing PII tags is used for training. Also, the generated guesses may have PII tags, and they need to be replaced with the corresponding PII string of the target user to obtain a final guess.

Similar to the construction of character features in trawling guessing scenarios, we also use four-dimensional features to represent PII tags in targeted guessing scenarios. Specifically, we have used ⟨character type, the rank of this character in its type, keyboard row number, and keyboard column number⟩ to represent an ordinary character. For PII tags, they are similar to ordinary characters except for the lack of keyboard features. Therefore, we use ⟨PII type, PII serial number, 0, 0⟩ to represent PII tags. The last two 0s are to align with the four-dimensional features of ordinary characters.

4.4 Experimental setups and results

Datasets. In Table 1, only 12306, ClixSense and Rootkit datasets are with PII (name, email, birthday, etc.). To enable extensive targeted guessing evaluation, we match the non-PII-associated datasets with these PII-associated ones through *email*, and this produces six PII-associated password datasets (i.e., PII-12306, PII-CSDN and PII-Dodonew, PII-ClixSense, PII-000Webhost and PII-Rootkit; See Table 2). Among them, Rootkit is a hacker forum, and 000Webhost is a free web hosting site and is mainly used by web administrators. Therefore, the users of both sites are likely to be more security-savvy than normal users, and this has been observed in [63]. We use these six PII-associated datasets to conduct six comparative experiments. In each experiment, half of each dataset is used as the training set, and the other half is used as the test set as recommended in [15, 60, 63] (see Table 3).

Approaches for comparison. The current mainstream targeted guessing models employing PII mainly include the TarGuess-I [63] based on PCFG [65] and the Targeted-Markov [61] based on the Markov model [38]. Note that the original Targeted-Markov proposed by Wang et al. [61] exploits only name information, but it can be easily extended to incorporate user name, birthday, email, etc. For a more comprehensive comparison, we apply our proposed PII matching algorithm to FLA [39], leading to FLA-PII. To our knowledge, this is the first time that FLA can capture PII semantics.

More specifically, we first identify the PII in a password, and encode it to a one-dimensional array based on the dictionary order (e.g., `wang666`→[1001, 6, 6, 6], where 1001 and 6 are the numerical labels corresponding to the surname and the digit 6 in the dictionary, respectively.). Here, we use *an embedding layer* rather than the canonical one-hot encoding layer to reduce the sparsity of the embedding vector due to the large size of PII tags. Then, the embedded vector is fed into LSTM neural networks. Finally, the dense layer converts the hidden layers into the output size. The output is the possible subsequent labels with probabilities, and FLA-PII chooses the next label with the highest probability. Here we set the embedding size to 128, and the remaining parameters are completely consistent with trawling FLA [39] in Sec. 3.4.

Note that, theoretically, an online guessing attacker can only perform very limited guessing attempts if the protection measures (e.g., lockout, rate-limiting [22]) are deployed on the server. For instance, NIST requires that "the verifier (server) *shall* limit consecutive failed authentication attempts on a single account to no more than 100" [26]. However, in reality, as revealed in [37], 72% of the top 182 websites "allow frequent, unsuccessful login attempts without account lockout or login throttling". Overall, the system has to balance

Table 3: Comparison of four PII-based models.[†]

Experimental setup		RFGuess-PII	4-order Tar-Markov [61]	Tar-Guess-I [63]	FLA [39]-PII
Guessing scenario	Guess #				
50% PII-12306 ↓ 50% PII-12306	10	**11.19%**	11.00%	10.60%	8.41%
	10^2	**21.37%**	20.91%	20.30%	17.47%
	10^3	**28.89%**	28.20%	26.30%	24.01%
	10^7	**52.75%**	42.00%	44.79%	50.51%
	10^{14}	**98.42%**	87.68%	48.12%	97.50%
50% PII-CSDN ↓ 50% PII-CSDN	10	**21.24%**	20.13%	21.20%	15.94%
	10^2	**28.23%**	27.01%	27.90%	21.96%
	10^3	**33.30%**	32.96%	33.00%	26.97%
	10^7	**53.14%**	46.94%	42.23%	52.85%
	10^{14}	**94.68%**	80.74%	44.00%	94.51%
50% PII-Dodonew ↓ 50% PII-Dodonew	10	**9.54%**	9.52%	9.40%	6.07%
	10^2	**20.45%**	20.33%	19.10%	16.00%
	10^3	30.21%	**30.29%**	26.50%	24.93%
	10^7	**61.21%**	59.62%	59.45%	60.72%
	10^{14}	**99.12%**	92.61%	64.86%	93.80%
50% PII-Clixsense ↓ 50% PII-Clixsense	10	**5.99%**	5.87%	4.90%	4.12%
	10^2	**9.51%**	9.05%	7.70%	7.67%
	10^3	**13.48%**	12.06%	11.70%	11.15%
	10^7	**48.30%**	41.01%	43.48%	33.75%
	10^{14}	**92.38%**	85.33%	56.38%	82.60%
50% PII-Rootkit ↓ 50% PII-Rootkit	10	**6.96%**	6.77%	6.77%	3.97%
	10^2	**11.40%**	11.07%	10.46%	8.21%
	10^3	14.88%	**15.17%**	14.59%	12.45%
	10^7	**39.45%**	35.73%	27.73%	38.70%
	10^{14}	**89.81%**	76.01%	33.24%	86.91%
50% PII-000Webhost ↓ 50% PII-000Webhost	10	**3.86%**	3.75%	0.90%	1.76%
	10^2	**7.31%**	6.89%	6.10%	4.64%
	10^3	**10.88%**	10.52%	9.54%	7.71%
	10^7	25.56%	22.26%	**26.17%**	25.73%
	10^{14}	**77.10%**	60.45%	36.43%	70.60%

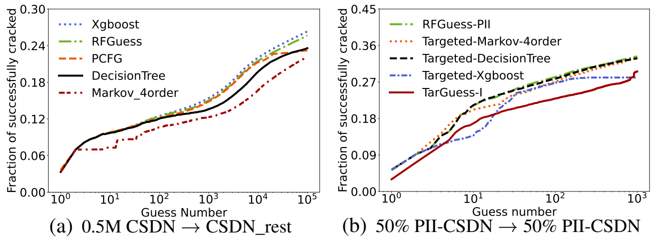
[†]A **bold** value (attack success rate) means that it is the highest one in each row.

online guessing attacks and denial-of-service (DoS) attacks. Without loss of generality, we set $T = 10^3$ as with mainstream online-guessing literature [44, 63] in our experiments.

In reality, there also exist offline attack scenarios that target specific users. For example, after obtaining a leaked password file, attackers will focus on some specific, most valuable accounts (such as celebrities, politicians, or specific common users deemed valuable/profitable), and devote more effort to them. In this case, the number of guesses will be limited only by the cost the attacker is willing to pay, which can be extremely large (e.g., $>10^{10}$). Thus, as recommended by [20], we also evaluate all the PII-models under larger guesses (i.e., 10^{14}) through the Monte-Carlo algorithm.

Experimental results. We design six targeted guessing scenarios, and the results are summarized in Table 3. For a more comprehensive comparison, we further use the guess-number-graph to evaluate the effectiveness of our RFGuess-PII with its counterparts, and put the details in Appendix C (see Figs. 10 and 13). For a fair comparison, all three counterpart targeted models (i.e., TarGuess-I [63], Targeted-Markov [61] and FLA [39]-PII) employ our improved PII matching algorithm. Results show that RFGuess-PII achieves a slightly better attack success rate in most cases within $10{\sim}10^3$ guesses. As the number of guesses increases, the superiorities of RFGuess-PII over its counterparts are enhanced. More specifically, RFGuess-PII outperforms its foremost counterpart (i.e., FLA-PII [39]) by 5.20%∼8.36% within $10^7{\sim}10^{14}$ guesses.

Further exploration. We now show that our representation of passwords can be easily transferred to other machine learning algorithms. More specifically, we replace the random forest with Xgboost [16]/DecisionTree (We simply replace the `RandomForestClassifier` class in our script with `Xgboost` and `DecisionTreeClassifier` with all the remaining pro-



(a) 0.5M CSDN → CSDN_rest (b) 50% PII-CSDN → 50% PII-CSDN

Figure 9: Using Xgboost [16] and decision tree for password guessing: (a) trawling guessing; (b) targeted guessing based on PII.

cessing flows unchanged), and perform two exploratory experiments in both trawling and targeted guessing scenarios. Fig. 9 show that attack success rates of Xgboost and DecisionTree (and also Targeted-Xgboost and Targeted-DecisionTree) are comparable to state-of-the-art models. Notably, their parameters can be better tuned for potential optimization, and we leave further exploration as future work.

5 RFGuess-Reuse: A new targeted guessing model based on reuse

We now focus on modeling users' password reuse behavior. Based on our RFGuess in Sec. 3, we first design a targeted guessing model called RFGuess-Reuse, and then conduct large-scale experiments to demonstrate its effectiveness.

5.1 New targeted password guessing model based on reuse

We now describe how the random forest model can be used for password reuse-based scenarios. Inspired by TarGuess-II [63], we also consider both structure-level and segment-level transformations. First, we count structure-level transformations like $\mathsf{L}_8\mathsf{S}_2{\rightarrow}\mathsf{L}_7\mathsf{D}_3$) by calculating the editing matrix for each password pair in the training set. Then we train a segment-level transformation (i.e., a transformation within a string of the same type, e.g., `password`→`passwor` in letter segment) model based on random forest. The formula for calculating the probability of generating a new password is

$$\Pr(pw_1 \rightarrow pw_2) = \left(\prod_{i=1}^{n} \Pr(Pt^i_{pw_1 \rightarrow pw_2})\right) * p_n, \qquad (6)$$

where $Pt^i_{pw_1 \rightarrow pw_2}$ stands for a specific transformation operation (e.g., inserting the digital structure `123`) from pw_1 to pw_2, and p_n represents the probability of ending after n operations. For example, given a password `password!!`, Pr(`password!!`→`p@sswor123`)=Pr(`password!!`→`passwor d`)∗Pr(`password`→`passwor`)∗Pr(`passwor`→`passwor123`)∗ Pr(`passwor123`→`p@sswor123`)$*p_4$, where Pr(`password!!` →`password`) (i.e., $\mathsf{L}_8\mathsf{S}_2{\rightarrow}\mathsf{L}_8$) and Pr(`passwor`→`passwor12 3`) (i.e., $\mathsf{L}_7{\rightarrow}\mathsf{L}_7\mathsf{D}_3$) are the probability of structure-level transformation, and can be obtained by statistics of password pairs in the training set; Pr(`password`→`passwor`) (i.e., delete a single character `d`) and Pr(`passwor123`→`p@sswor123`) (i.e., a$\rightarrow$@) are the probability of segment-level transformation, and can be obtained by the trained random forest model; p_4 is the probability of ending after four transformations, and can also be obtained by statistics of the training set.

For structure-level transformation, we take the insertion of the structure 123 (i.e., D_3) at the tail of passwor as an example. Its probability is Pr(passwor$\rightarrow$passwor123)= $\Pr(T_1) * \Pr(T_2) * \Pr(\texttt{123}|\mathsf{D}_3)$, where T_1 denotes the event "Insert structures at the tail of passwor", and $\Pr(T_1)$ can be obtained by counting the reuse behaviors in the training set according to the length distribution of the training set (see Table 13 in Appendix G); T_2 denotes the event "Insert the specific structure D_3", and both $\Pr(T_2)$ and $\Pr(\texttt{123}|\mathsf{D}_3)$ can be obtained by training a PCFG model [38].

For segment-level transformation, we consider four atomic transformations based on [63]: head insertion, head deletion, tail insertion, and tail deletion. For three types of segments (i.e., letters, digits, and special character), we train random forests in positive order and reverse order, respectively, and this generates 3×2 models in total. For example, when determining the probability of performing the tail insertion operation of passwor, we input passwor into the positive order letter random forest to obtain this conditional probability; when determining the probability of performing the head insertion operation, we input rowssap into the reverse order letter random forest to obtain this conditional probability.

We take the *positive order letter* random forest as an example, and consider the operations related to the last character. When training the password password!!, three behavioral characteristics need to be trained for the letter string password: inserting characters, unchanged, and deleting characters. For inserting characters, our model uses assword as the training input, and uses d as the training output; for unchanged, our model uses ssword as the training input, and uses the end character E_s as the training output; for deleting characters, our model uses ssword plus any letter as the training input and uses -1 as the training output (i.e., the input is sword* and the output -1, where * can be any letter).

Here we give a toy example of how to calculate the probability of a segment-level transformation. Given a password password!!, it can be divided into two segments L_8 and S_2 (denoted as p_1,p_2), and we calculate the probability of deleting d at the tail of the first segment p_1 (denoted as event P_1^t) as $\Pr(P_1^t) = \Pr(S_1) * \Pr(S_2) * \Pr(S_3) * \Pr(S_4)$, where S_1 denotes the event "Perform segment-level transformation" , and $\Pr(S_1)$ can be obtained by counting the reuse behavior of the training set; S_3 and S_4 denote the event "Perform tail deletion operation on p_1" and the event "Delete character d at the end of p_1", respectively, and both $\Pr(S_3)$ and $\Pr(S_4)$ are calculated by the trained random forest model; S_2 denotes the event "Perform operation on p_1", and $\Pr(S_2)$ is calculated by $\frac{1-\Pr(E_s|p_1)+1-\Pr(E_s|\overline{p_1})}{\sum_{i=1}^{2}(1-\Pr(E_s|p_i)+1-\Pr(E_s|\overline{p_i}))}$, where $\overline{p_1}$ represents the inversion of p_1 (i.e., password$\rightarrow$drowssap), and $\Pr(E_s|p_1)/\Pr(E_s|\overline{p_1})$ is obtained by the positive/reverse order random forest model; "$1-\Pr(E_s|p_1)$" represents the probability of performing the *tail* operation (because $\Pr(E_s|p_1)$ represents the probability of unchanged operation), and "$1-\Pr(E_s|\overline{p_1})$" represents the probability of performing the *head* operation.

Table 4: Basic information about password reuse datasets.

Dataset	Language	Items	# Same password pair	# Similar password pair†
CSDN→126	Chinese	195,832	62,686	47,690
CSDN→12306	Chinese	12,635	7,079	2,815
12306→Dodonew	Chinese	49,775	35,395	9,386
CSDN→Dodonew	Chinese	5,997	2,040	1,597
000Webhost→Clixsense	English	150,273	35,470	41,731
000Webhost→LinkedIn	English	231,452	50,875	52,731
000Webhost→Yahoo	English	36,936	5,960	6,303
000Webhost→Mate1	English	51,942	7,613	25,504

† Similar means that the similarity score s is within [0.5, 1.0], and it is calculated as $s = 1 - \text{EditDistance}(pw1, pw2)/\max(|pw1|, |pw2|)$.

The formula of calculating $\Pr(S_2)$ is used to solve the problem of unequal operation probability of each segment. For instance, the structure of password!! is $\mathsf{L}_8\mathsf{S}_2$, and the operation probability (e.g., insertions or deletions) on different segments (i.e., L and S) is not equal in practice, while TarGuess-II [63] regards it as equal in the structure-level. To address this issue, we treat the probability of each segment (take L segment as an example) be expressed by the ratio of "the sum of the operation probabilities of L segment (i.e., password) to that of all the segments (L and S segments)".

In the guess-generation phase as with [63], after each operation performed on the original password, the corresponding probability is calculated and inserted into a priority queue, and the guess with the highest probability is output. Then we repeat this process until the number of generated guesses reaches the predefined threshold (e.g., 10^3).

5.2 Experimental setups and results

Datasets. We select four English and four Chinese datasets to conduct experiments on password-reuse guessing scenarios (see Table 4). Among them, 000Webhost→ClixSense and CSDN→126 are selected as the training set for English and Chinese guessing scenarios. We take the dataset "000Webhost→ClixSense" as an example. It is obtained by matching two datasets (000Webhost and ClixSense) through *email* and consists of password pairs like $(email_{U_i}, pw_{i1}, pw_{i2})$ for user U_i. In the training phase, U_i's password pw_{i1} comes from the 1st dataset (000Webhost), and the attacker $\mathcal{A}$ learns/trains how it can be used to guess pw_{i2} from the 2nd dataset (ClixSense). Then, suppose the dataset "000Webhost→Yahoo" is used for testing. $\mathcal{A}$ exploits pw_{j1} from 000Webhost as victim j's leaked password, and uses the trained password model to generate guesses until pw_{j2} from Yahoo is generated.

We compare our proposed model with TarGuess-II [63] and Pass2Path [44]. TarGuess-II and our RFGuess-Reuse require additional PCFG structure dictionaries (see Sec. 5.1) and popular password dictionaries (see Sec 4.2 in [63]), and we maintain the same datasets for these two models. For Pass2Path, we use the recommended parameters in [5] to train the model. Similar to the targeted guessing scenarios

Table 5: Comparison of three password reuse models.†

Experimental setup		RFGuess-	Pass2-	TarGuess-
Guessing scenario	Guess number	Reuse	Path [44]	II [63]
CSDN → 12306	10	68.41%	68.80%	68.13%
	100	73.09%	70.72%	73.19%
	1,000	75.86%	72.16%	75.57%
CSDN → Dodonew	10	48.59%	48.82%	48.44%
	100	53.86%	51.79%	54.56%
	1,000	57.71%	53.84%	57.58%
12306 → Dodonew	10	84.14%	83.44%	84.11%
	100	86.00%	85.69%	86.34%
	1,000	87.65%	86.78%	87.58%
000webhost → Mate1	10	27.70%	25.11%	30.17%
	100	31.29%	26.42%	32.14%
	1,000	33.77%	27.73%	34.37%
000webhost → LinkedIn	10	35.67%	32.65%	36.17%
	100	37.77%	34.06%	38.16%
	1,000	39.52%	35.69%	39.72%
000webhost → Yahoo	10	26.53%	24.84%	27.12%
	100	28.59%	25.87%	28.69%
	1,000	30.13%	26.99%	30.19%

†A value with dark gray (resp. light gray) represents the highest one (resp. 2nd one).

based on PII, we also generate 10^3 guesses for each model.

Table 5 shows that RFGuess-Reuse achieves the best or 2nd best results among three models. In particular, within 10^3 guesses, the attack success rates of TarGuess-II [63] and our RFGuess-Reuse are about 1%∼7% higher than that of Pass2Path [44]. For English datasets, although the attack success rate of RFGuess-Reuse is slightly lower than that of TarGuess-II, it is still 7%∼22% higher than Pass2Path.

6 Discussion

We now discuss the security implications of this work and our insights on online/offline password guessing.

Honeywords. At CCS'13, Juels and Rivest [34] proposed a decoy password mechanism to timely detect password file compromises, called honeywords. This mechanism can generate $k-1$ (e.g., k=20 in [34]) honeywords for each account, and both the real password and its corresponding honeywords are stored together. In addition, the index of each real password is stored in another server named honeychecker. When an attacker tries to log in with a honeyword, the system signals a possible leak. As a leading password model, RFGuess can be potentially employed to generate honeywords to timely detect password leakage. In this application scenario, the model size and password generation speed are not particularly important since the server only needs to generate 20∼40 honeywords (as recommended by [64]) for each account, and such generation is conducted only once for an account. For example, the Markov/TarMarkov model employed by the hybrid method proposed in [64] can simply be replaced by our RFGuess/RFGuess-PII to generate flatter honeywords (that are harder to be differentiated from real passwords).

Feature importance score. As shown in Sec. 3.2, RFGuess can efficiently identify the dominant factors of password security through the feature importance score to resist against data-driven guessing. For example, we only need to set the number of character classes as a password prefix feature, and RFGuess can automatically show if it is one of the dominant factors impacting password security through the feature importance score. This can help administrators enforce more effective password policies. For example, more character classes contribute marginally improvement in password security due to the imbalanced use of symbol strings, while more segments (i.e. a continuous string whose characters have a strong correlation) can significantly help resist against guessing [58]. Also, the feature importance score allows users to understand which dimensions of a character (e.g., type, continuity, and position-information) impact the password security to what extent, thus helping them create more secure passwords.

Online/Offline password guessing. Before cracking, the offline guessing attacker has the salted-password accounts, but generally has no prior knowledge of which passwords are used by the target accounts, and thus it is more realistic/reasonable to do *not* exclude duplicate passwords in the training set from the test set when evaluating a guessing algorithm, as done in Sec. 3.4 (and [38,39,57]) and opposed to [31,65,67]. Besides, excluding duplicate passwords can only evaluate/simulate the generalization ability, but overlooks the evaluation of fitting ability. In practice, the *generalization* ability corresponds to offline guessing scenarios with relatively large guess numbers (e.g., $>10^7$). Although previous work [24] suggested that 10^{14} could be a lower boundary for offline guessing, the size of guessing dictionaries explicitly generated by existing password guessing literature (e.g., [38,39,47,65]) generally does not exceed 10^{11} (due to the limitation of generation speed and computing resources). This implies that the practical significance of guessing algorithms' generalization ability is mainly highlighted in 10^7∼10^{10} guesses. In contrast, the *fitting* ability mainly corresponds to online guessing scenarios with relatively small guess numbers (e.g., $<10^7$), while online guessing is the most concerning threat that normal users need to devote efforts to mitigate [7,24,63]. Thus, when evaluating a guessing model/algorithm, it is of practical significance to consider both the fitting ability and generalization ability.

In all, it is more realistic to do *not* exclude duplicate passwords in the training set from the test set. Actually, this practice has been preferred in password research (see [38,39,57]), but we *for the first time* explain why it is acceptable.

7 Conclusion

This paper, for the first time, introduces classical machine learning techniques for password guessing, and designs three new guessing models for the three most representative guessing scenarios: trawling guessing, targeted guessing based on PII and on reuse. Extensive experiments with 13 real-world datasets demonstrate the effectiveness and scalability of our models. This work provides a brand new technical route for modeling users' password guessability and opens up new directions for designing effective password guessing models.

Acknowledgement

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 and 62222208, and Natural Science Foundation of Tianjin, China under Grant No. 21JCZDJC00190.

References

[1] *cuML documentation for RandomForestClassifier*, https://docs.rapids.ai/api/cuml/stable/.

[2] *sklearn documentation for RandomForestClassifier*, https://scikit-learn.org/stable/about.html.

[3] *Cracking passwords from the Mall.cz dump*, Jan. 2018, https://www.michalspacek.com/cracking-passwords-from-the-mall.cz-dump.

[4] *Flipboard Confirms It Was Hacked Twice: 150M Users At Risk As Passwords Stolen*, May 2019, https://bit.ly/3ICqJ0S.

[5] *credTweak*, Sept. 2020, https://github.com/Bijeeta/credtweak.

[6] *Hacker leaks full database of 77 million Nitro PDF user records*, Jan. 2021, https://www.bleepingcomputer.com/news/security/hacker-leaks-full-database-of-77-million-nitro-pdf-user-records/.

[7] *Identity is the new battleground*, 2022, https://news.microsoft.com/wp-content/uploads/prod/sites/626/2022/02/Cyber-Signals-E-1.pdf.

[8] *Recently added breaches.*, 2022, https://haveibeenpwned.com/.

[9] L. Abrams, *533 million Facebook users' phone numbers leaked on hacker forum*, April 2021, https://www.bleepingcomputer.com/news/security/533-million-facebook-users-phone-numbers-leaked-on-hacker-forum/.

[10] J. Blocki, B. Harsha, S. Kang, S. Lee, L. Xing, and S. Zhou, "Data-independent memory hard functions: New attacks and stronger constructions," in *Proc. CRYPTO 2019*, pp. 573–607.

[11] J. Bonneau, "The science of guessing: Analyzing an anonymized corpus of 70 million passwords," in *Proc. IEEE S&P 2012*, pp. 538–552.

[12] 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.

[13] 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.

[14] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[15] C. Castelluccia, M. Dürmuth, and D. Perito, "Adaptive password-strength meters from Markov models," in *Proc. NDSS 2012*, pp. 1–14.

[16] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proc. ACM SIGKDD 2016*, pp. 785–794.

[17] K. Cho, B. van Merrienboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in *Proc. EMNLP 2014*, pp. 1724–1734.

[18] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Trans. Inf. Theory*, vol. 13, no. 1, pp. 21–27, 1967.

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

[20] M. Dell'Amico and M. Filippone, "Monte carlo strength evaluation: Fast and reliable password checking," in *Proc. ACM CCS 2015*.

[21] M. Dürmuth, F. Angelstorf, C. Castelluccia, D. Perito, and C. Abdelberi, "OMEN: faster password guessing using an ordered markov enumerator," in *Proc. ESSoS 2015*, pp. 119–132.

[22] 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.

[23] N. Eide, *Anthem breached again after contractor emailed file with 18,500 members' info*, Aug. 2017, https://www.ciodive.com/news/anthem-breached-again-after-contractor-emailed-file-with-18500-members-in/448334/.

[24] D. Florêncio, C. Herley, and P. C. van Oorschot, "An administrator's guide to internet password research," in *Proc. USENIX LISA 2014*.

[25] S. Gatlan, *Hacker leaks full database of 77 million Nitro PDF user records*, Jan. 2021, https://www.bleepingcomputer.com/news/security/hacker-leaks-full-database-of-77-million-nitro-pdf-user-records/.

[26] 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., June 2017.

[27] W. Han, M. Xu, J. Zhang, C. Wang, K. Zhang, and X. S. Wang, "Transpcfg: Transferring the grammars from short passwords to guess long passwords effectively," *IEEE Trans. Inf. Forensics Secur.*, vol. 16, pp. 451–465, 2021.

[28] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Journal of the royal statistical society. series c (applied statistics)*, vol. 28, no. 1, pp. 100–108, 1979.

[29] M. X. Heiligenstein, *Twitter Data Breaches: Full Timeline Through 2022*, Dec. 2022, https://firewalltimes.com/twitter-data-breach-timeline/.

[30] C. Herley and P. Van Oorschot, "A research agenda acknowledging the persistence of passwords," *IEEE Secur. Priv.*, vol. 10, pp. 28–36, 2012.

[31] B. Hitaj, P. Gasti, G. Ateniese, and F. Perez-Cruz, "Passgan: A deep learning approach for password guessing," in *Proc. ACNS 2019*.

[32] S. Houshmand, S. Aggarwal, and R. Flood, "Next gen PCFG password cracking," *IEEE Trans. Inf. Forensics Secur.*, vol. 10, no. 8, pp. 1776–1791, 2015.

[33] M. I. Jordan and T. M. Mitchell, "Machine learning: Trends, perspectives, and prospects," *Science*, vol. 349, no. 6245, pp. 255–260, 2015.

[34] A. Juels and R. L. Rivest, "Honeywords: Making password-cracking detectable," in *Proc. ACM CCS 2013*, pp. 145–160.

[35] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," in *Proc. NIPS 2017*, pp. 3146–3154.

[36] Y. Li, H. Wang, and K. Sun, "A study of personal information in human-chosen passwords and its security implications," in *Proc. IEEE INFOCOM 2016*, pp. 1–9.

[37] B. Lu, X. Zhang, Z. Ling, Y. Zhang, and Z. Lin, "A measurement study of authentication rate-limiting mechanisms of modern websites," in *Proc. ACSAC 2018*, pp. 89–100.

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

[39] 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 2016*, pp. 175–191.

[40] R. Morris and K. Thompson, "Password security: A case history," *Commun. ACM*, vol. 22, no. 11, pp. 594–597, 1979.

[41] A. Narayanan and V. Shmatikov, "Fast dictionary attacks on passwords using time-space tradeoff," in *Proc. ACM CCS 2005*, pp. 364–372.

[42] W. S. Noble, "What is a support vector machine?" *Nature biotechnology*, vol. 24, no. 12, pp. 1565–1567, 2006.

[43] V. Pai, *Yahoo Discloses Its Second Data Breach In 4 Months*, Dec. 2016, https://www.medianama.com/2016/12/223-yahoo-2nd-data-breach/.

[44] 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.

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

[46] D. Pasquini, M. Cianfriglia, G. Ateniese, and M. Bernaschi, "Reducing bias in modeling real-world password strength via deep learning and dynamic dictionaries," in *Proc. USENIX SEC 2021*, pp. 821–838.

[47] 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.

[48] R. Polikar, "Ensemble learning," in *Ensemble machine learning*. Springer, 2012, pp. 1–34.

[49] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.

[50] R. E. Schapire, "A brief introduction to boosting," in *Proc. IJCAI 1999*.

[51] C. E. Shannon, "A mathematical theory of communication," *ACM SIGMOBILE Mobile Comput. Commun. Rev.*, vol. 5, no. 1, pp. 3–55, 2001.

[52] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. NeurIPS 2014*, pp. 3104–3112.

[53] S. Taylor, *Music Service Deezer Admits Data Breach via Third Party, Possibly Affecting 200M+ Users*, Dec. 2022, https://restoreprivacy.com/music-service-deezer-data-breach/.

[54] B. Ur, P. G. Kelley, S. Komanduri, and et al., "How does your password measure up? The effect of strength meters on password creation," in *Proc. USENIX SEC 2012*, pp. 65–80.

[55] B. Ur, S. M. Segreti, L. Bauer, N. Christin, L. F. Cranor, S. Komanduri, D. Kurilova, M. L. Mazurek, W. Melicher, and R. Shay, "Measuring real-world accuracies and biases in modeling password guessability," in *Proc. USENIX SEC 2015*, pp. 463–481.

[56] R. Veras, C. Collins, and J. Thorpe, "On the semantic patterns of passwords and their security impact," in *Proc. NDSS 2014*, pp. 1–16.

[57] Veras, Rafael and Collins, Christopher and Thorpe, Julie, "A large-scale analysis of the semantic password model and linguistic patterns in passwords," *ACM Trans. Priv. Secur.*, vol. 24, no. 3, pp. 1–21, 2021.

[58] C. Wang, J. Zhang, M. Xu, H. Zhang, and W. Han, "# segments: A dominant factor of password security to resist against data-driven guessing," *Comput. Secur.*, vol. 121, p. 102848, 2022.

[59] D. Wang, H. Cheng, P. Wang, X. Huang, and G. Jian, "Zipf's law in passwords," *IEEE Trans. Inf. Forensics Secur.*, vol. 12, no. 11, pp. 2776–2791, 2017.

[60] D. Wang, H. Cheng, P. Wang, J. Yan, and X. Huang, "A security analysis of honeywords," in *Proc. NDSS 2018*, pp. 1–15.

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

[62] 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.

[63] 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.

[64] D. Wang, Y. Zou, Q. Dong, Y. Song, and X. Huang, "How to attack and generate honeywords," in *Proc. IEEE S&P 2022*, pp. 966–983.

[65] 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.

[66] D. Wheeler, "zxcvbn: Low-budget password strength estimation," in *Proc. USENIX SEC 2016*, pp. 157–173.

[67] M. Xu, C. Wang, J. Yu, J. Zhang, K. Zhang, and W. Han, "Chunk-level password guessing: Towards modeling refined password composition representations," in *Proc. ACM CCS, 2021*, pp. 5–20.

[68] K. Yang, X. Hu, Q. Zhang, J. Wei, and W. Liu, "Studies of keyboard patterns in passwords: Recognition, characteristics and strength evolution," in *Proc. ICICS 2021*, pp. 153–168.

A Algorithm applicability analysis

In this section, we analyze the feasibility of several classical machine learning algorithms that tackle multi-classification tasks when applied to password guessing.

Support vector machine (SVM). SVM [42] maps the sample space to a high-dimensional space through a nonlinear function, so that the sample is linearly separable in the feature space. The goal of SVM is to find a hyperplane that maximizes the distance between the points closest to the decision boundary in samples of different classifications. The potential problem is that SVM is susceptible to noise interference. More specifically, due to the uncertainty of the password, that is, any string/character may be followed by any string/character, the sample space is noisy. As a result, it is not efficient when applying SVM to password guessing.

K-means clustering. **K**-means [28] is an unsupervised clustering algorithm, and its goal is to classify similar samples into a specific category automatically. Specifically, the algorithm first selects **K** samples randomly as the initial clustering center. Then the distances between each sample and **K** centers are calculated. Next, each sample is assigned to the nearest center, and the cluster center for each category is recalculated by the mean value. This step is repeated until the termination condition is met. However, when applying this algorithm to password guessing, several tricky problems exist. First, it is not easy to calculate the probability of classification. Second, the algorithm is difficult to converge when the samples are noisy. Third, the value of **K** needs to be set manually, and different **K** values will get pretty different results.

K-nearest neighbor algorithm (KNN). The core idea of KNN [18] is that if most of the K nearest neighbor samples of a given sample in the feature space belong to a specific category, then the given sample is classified into this category. Compared with the statistical-based models (e.g., Markov [41]), this classification method has the potential to address the issue of data sparseness (since it's based on sample similarity). However, the calculation amount of KNN is relatively large, and the value of K is difficult to determine. In particular, this algorithm relies heavily on the quality of feature construction, and the similarity is not learnt by the algorithm itself, but relies on artificial definitions.

Random forest. Random forest [14] integrates multiple decision trees (it is a predictive model, which represents a mapping relationship between object attributes and object values) in

Table 6: The notations used in this paper.†

Notations	Descriptions
B_s E_s	Begining symbol and Ending symbol
$\mathsf{L_n}$ $\mathsf{D_n}$ $\mathsf{S_n}$	The letters, digits and special characters strings with length n
$\mathsf{N_1}$~$\mathsf{N_{10}}$	N stands for name usages, while $\mathsf{N_1}$ for the usage of full name, $\mathsf{N_2}$ for the abbr. of full name (e.g., lw from "lei wang"),$\cdots$.
$\mathsf{B_1}$~$\mathsf{B_5}$	B stands for birthday usages, $\mathsf{B_1}$ for full birthday in YMD format (e.g., 19820607), $\mathsf{B_2}$ for full birthday in MY, $\cdots$.
$\mathsf{U_1}$~$\mathsf{U_5}$	U stands for username usages, $\mathsf{U_1}$ for full username, $\mathsf{U_2}$ for the letter segment of the user name,$\cdots$.
$\mathrm{td_c}$ $\mathrm{ti_s}$ $\mathrm{S_p}$	$\mathrm{td_c}$ means deleting a character from the tail; $\mathrm{ti_s}$ means inserting a structure from the tail; $\mathrm{S_p}$ means special operation (e.g, leet and capitalization).

parallel through ensemble learning and votes together to get the final result. When random forest is applied to password guessing, the decision tree handles classification problems similarly to KNN. That is, strings with similar features are assigned to the same leaf node. The difference is that there is no need to determine the value of k, and the calculation amount is relatively small. For features that have never appeared before, the algorithm will classify them according to the samples with similar features in the leaf nodes.

Boosting. The Boosting algorithm [50] belongs to ensemble learning. Its main idea is to reduce the training error by iteratively training multiple weak classifiers, and then perform weighted fusion of these weak classifiers to produce a strong classifier. It can better control the generalization error while reducing the training error. At the same time, Boosting can add regular terms to prevent overfitting and is insensitive to noise. In the application of password guessing, the Boosting and its variants can also overcome the limitations of the Markov model well. While it's likely to be suitable for applications in password guessing, the training and guess-generation speed are relatively slow in our preliminary exploratory experiments. Thus, we leave related research as future work.

Algorithm analysis summary. Based on the above analysis and our preliminary exploratory experiments, we find that ensemble learning tends to be more suitable for password guessing. In this work, we mainly take Random forest [14] as a typical case study. As for the research on other machine learning techniques unexplored (e.g., LightGBM [35], Stacking and its variants [48]), we leave it as future work.

B Parameter selection

As far as we know, there are few scientific methods to find the best hyperparameters. However, a task-oriented analysis along with a number of empirical experiments provide a promising way: Since the password length of most users is at least six [38, 59], the order of our model is set to six. In this way, the total number of prefixes (6-order strings) is about 8.9~10.4 times the number of passwords, so the minimum number of samples in each leaf node is set to 10. For the number of trees, we take the CSDN dataset as an example, and set this value to 10, 30, 50, and 70, respectively. We find that when the number of trees is $>$30, the increase in the attack success rate is very limited ($<$0.5%). Also, the greater the value, the larger the RAM consumed during training (e.g., with 30 trees and five million training sets, it occupies about 40GB of RAM), and the slower the password training and generating will be. Therefore, we set this value to 30. In addition, the maximum ratio of features is determined by the importance score of each feature after our preliminary exploratory experiment (see Sec. 3.4). Compared with the complex parameters (e.g., number and type of layers, number of neurons per layer, activation function, etc.) of deep learning based models, the hyperparameter tuning of random forest are more concise and straightforward.

Note that, we have conducted parameters tuning for deep learning-based models implemented in this paper (i.e., FLA [39] and Pass2Path [44]). For FLA, we have initially referred to the parameter choices (i.e., the larger model that performs better in the original paper [39]) in an existing paper (IEEE S&P'21 [47]) that takes FLA as a baseline, but found its guessing success rate is low due to the small training-set regime. So we adopt the setup of the "small" model discussed in [39] and get significantly better results. For Pass2Path, we have tried to adjust the original setup in [44] (i.e., lowering the number of LSTM layers from three to two, and lowering the hidden size from 128 to 64). However, we find that its guessing success rate decreases by 1%~3% after the parameter adjustment. So we retain the original parameter settings.

Table 7: Performance of different trawling guessing models.†

Model	RFGuess	PCFG [65]	3-order Markov [38]	FLA [39]
Training time	0.3h	24s	102s	16h
Model size	4.5G	93.2M	1.4G	5.8M
Generated PW/s	130	82,372	13,303	2,500

† CPU: Xeon silver 4210R 2.4GHz; GPU: GeForce RTX 3080 (5M dataset).

C Supplementary experiment results

We put some additional experimental results in this section to show the performance of RFGuess against other models more thoroughly. Table 10 summarizes the concrete result values at some specific guess numbers in the trawling guessing scenario, and Figs. 10, 13 and 16 are the guess-number-graph of targeted guessing scenarios. Note that we have also compared our RFGuess-PII with other PII-models in *cross-site* guessing scenarios, and Fig. 15 shows the results.

We compare different approaches in terms of training time, generation speed as well as trained model size. Table 7 re-

Table 8: Model size of different PII-based models.†

Model	RFGuess-PII	TarGuess-I [63]	3-order Tar-Markov [61]	FLA [39]-PII
Model size	101M	893K	12.2M	5.8M

†CPU: Xeon silver 4210R 2.4GHz; GPU: GeForce RTX 3080 (50% CSDN-PII).

Table 9: Model size of different reuse-based models.†

Model	RFGuess-Reuse	Pass2Path [44]	TarGuess-II [63]
Model size	121M	40.1M	1.04G

†CPU: Xeon silver 4210R 2.4GHz; GPU: GeForce RTX 3080 (CSDN$\rightarrow$12306).

veals that statistics-based models (i.e., PCFG [65] and 3-order Markov [38]) require the shortest training time, followed by our RFGuess, and FLA [39] is the longest. Our RFGuess has the largest model size even after the compress (we set the compress parameter in the joblib tool to three and the number of trees to 30), but its fast training speed enables it to be trained *on site* without the need to save/maintain model files, and this property is quite desirable. While computational complexity is not particularly important for online guessing, we give the detailed model size of all tested models in targeted guessing scenarios, and the results are shown in Tables 8 and 9.

As for the guess-generation speed, RFGuess is first built on the scikit-learn framework [2], which does not support GPU acceleration. As a result, the generation speed is low: 130 passwords/s. We further migrate our RFGuess to the cuML framework (which supports GPU acceleration) [1], and the generation speeds increase by 5.2 times to 677 passwords/s in our preliminary experiments with 1,000 training data. Besides, if we use the decision tree model (i.e., set the number of trees to one), the password generation speed will be further increased to 1,520 passwords/s. In general, for online password guessing, an account should have been blocked quickly after a predefined number of failed login attempts (e.g., 100 and 1,000 are typical values considered by the main-stream standard [26] and academic literature [44, 63]); For offline guessing, memory hard hash algorithms such as SCRYPT or Argon2 are recommended [26], and they might move offline attackers closer to $10^6\sim10^7$ guesses [10]. Thus, the guess-generation speed of RFGuess is practically acceptable.

D Problems in mainstream approaches

PCFG. The core assumption of PCFG [65] is that different types of segments (i.e., letter, digit, and special character) in a password are independent. This is because the association between segments is much smaller than that within segments for most passwords. However, there exist some popular passwords that contain keyboard patterns or specific transformations, such as "1qa2ws3ed" and "p@ssw0rd" with the basic structure "$D_1L_2D_1L_2D_1L_2$" and "$L_1S_1L_3D_1L_2$". Such relatively long structures have a very low probability in the training set. When generating such passwords, the probabilities of each segment are inevitably required to be multiplied. This leads to a extremely low generation probability of these passwords, making their strength overestimated. Moreover, to our knowledge, PCFG has no effective smoothing method for the basic structure that has not appeared in the training set.

Markov. The order of the Markov model [41] determines how many characters need to be considered when predicting the following character. Intuitively, the higher the order, the better the cracking effect. However, in IEEE S&P'14, Ma et al. [38] demonstrates that the cracking success rate is best when the order is four, and a certain degree of overfitting occurs when the order is too high. This is because the Markov model calculates the conditional probability by the Bayesian formula, and the calculation of the probability needs to count the frequency of the string in the training set (i.e., $\Pr(c_i|c_{i-d}\cdots c_{i-1}) = \frac{\texttt{Count}(c_{i-d}\cdots c_{i-1}c_i)}{\texttt{Count}(c_{i-d}\cdots c_{i-1}\cdot)}$, where c_i represents individual characters). When the order is too high, the frequency of the prefix (i.e., n+1-gram string) in the training set is too small or even zero (i.e., the data sparseness issue).

To address the issue of data sparseness, Ma et al. [38] proposed a number of smoothing techniques, among which the most straightforward and relatively effective technique is the Laplace smoothing: One can add a small value (the recommended value is 0.01) to the frequency of each character after training. However, this smoothing technique is more just to alleviate this problem. We take the string `*1234` as an example. Generally, the string `*12345` rarely or does not appear in the training set, then $\Pr(5|*1234) = \frac{\texttt{Count}(5|*1234)+0.01}{\sum_{\alpha\in\Sigma}+0.01\cdot 95}$ (here Σ is the set of 95 printable characters) is still a tiny number even after applying the smoothing technique. In reality, a practical model should first consider the more important sub-string `1234` and then `*1234`. However, the 5-order Markov model only considers the entire strings of length five and overlooks some more important sub-string features.

Pass2Path. In 2019, Pal et al. [44] proposed a targeted guessing model (called Pass2Path) based on deep learning. This model is based on the transformation path between users' reused password pairs. More specifically, it employs the seq2seq model [17] for training, where the input is user's existing password, and the output is the transformation path. However, there exist multiple transformation paths between password pairs, but only one path will be used for training. For example, password pair `12a3`$\rightarrow$`12@` has two transformation paths. One is to delete `3` and then replace `a` with `@`, the other is to replace `3` with `@` and then delete `a`. Both of them are possible paths with the same edit distance of two. However, when the second path is used for training, the first path will be regarded as noise, which is unrealistic. As a result, how to determine which path to choose for training is a controversial issue. In comparison, if we choose to count the number of operations (e.g., number of substitutions and deletions), the statistical results obtained by different transformation paths will be the same. This inspires us more tend to propose a new algorithm based on statistics of the transformation operation number rather than the transformation path.

Table 10: Comparison of the attack success rate of four different trawling guessing approaches.†

Experimental setup		RFGuess	4-order Markov [38]	3-order Markov [38]	PCFG [65]	FLA [38]	Min-auto [55]
Guessing scenario	Number of guesses						
#3: 0.01M Rockyou → Rockyou	10^7	**36.76%**	26.32%	27.35%	19.73%	31.63%	44.76%
	10^{14}	**92.77%**	75.47%	79.77%	20.24%	91.58%	94.27%
#1: 0.1M Rockyou → Rockyou	10^7	**46.91%**	38.84%	41.24%	36.68%	36.35%	55.66%
	10^{14}	**94.59%**	85.56%	89.34%	40.22%	92.33%	95.97%
#2: 1M Rockyou → Rockyou	10^7	**54.22%**	50.28%	45.01%	49.99%	44.07%	61.67%
	10^{14}	**95.81%**	91.30%	93.10%	58.78%	94.06%	97.11%
#6: 0.01M Rockyou → 000Webhost	10^7	**6.19%**	4.12%	4.06%	5.99%	4.24%	9.98%
	10^{14}	56.83%	31.04%	35.70%	7.19%	**58.78%**	63.00%
#4: 0.1M Rockyou → 000Webhost	10^7	9.61%	6.95%	7.35%	**12.37%**	7.40%	15.53%
	10^{14}	59.94%	41.29%	45.73%	19.00%	**60.80%**	66.47%
#5: 1M Rockyou → 000Webhost	10^7	12.56%	11.08%	9.05%	**15.87%**	9.92%	18.28%
	10^{14}	62.74%	48.88%	53.46%	34.04%	**64.48%**	70.15%
#7: 75% 000Webhost → 000Webhost	10^7	**28.16%**	19.29%	13.04%	22.89%	20.02%	31.56%
	10^{14}	**76.88%**	68.29%	71.32%	62.12%	76.52%	81.87%
#8: 75% 000Webhost → Wishbone	10^7	**29.80%**	25.79%	17.02%	28.09%	21.11%	37.07%
	10^{14}	93.26%	89.12%	91.35%	63.07%	**94.50%**	97.20%

† A **bold** value (i.e., the attack success rate) means that it is the highest one in each row (excluding the ideal Min-auto approach).

TarGuess-II. By observing the characteristics of users' password reuse behaviors, Wang et al. [63] proposed TarGuess-II in 2016. TarGuess-II [63] is based on PCFG [65] and defines two types of transformations: the structure-level transformation (insertion and deletion of L, D, and S structure segments, e.g., $L_6D_3 \rightarrow L_6$) and the segment-level transformation (insertion and deletion of characters in L, D, and S segments, e.g., `123456`→`12345` in digit segment). We take the password pair `password` and `p@sswor123` as an example. The probability is calculated by: $\Pr(\texttt{password} \rightarrow \texttt{p@sswor123}) = \Pr(L_8 \rightarrow td_c) * \Pr(L_8 \rightarrow ti_s) * \Pr(ti_s \rightarrow D_3) * \Pr(D_3 \rightarrow 123) * \Pr(S_p \rightarrow \text{Leet}) * \Pr(\text{Leet} \rightarrow \texttt{a,@})$, where td_c means deleting a character from the tail, ti_s means inserting a structure to the tail, and S_p means special operation (e.g, leet and capitalization).

Note that TarGuess-II [63] does not have the issue of transformation path selection. However, it has the following three shortcomings: (1) For structure-level transformation, since the basic structure of some passwords (e.g., `1qa2ws3ed!`) appears very rarely in the dataset, the probability of insertion and deletion of each structure obtained through statistics on the training set may lead to over-fitting. (2) For segment-level transformation, the frequency in the training set is not accurate because the probability of inserting and deleting characters in the same segment is likely to be different. For example, the probability of tail deletion for `123450` and `123456` is apparently different. `123456` is likely to remain unchanged, while `123450` is likely to delete the last digit `0`. However, the probability of inserting and deleting characters is the same for the same structure segment (Here the basic structure of `123450` and `123456` are both L_6). (3) Still considering the segment-level transformation, if a character (e.g., `#`) is inserted after a segment (e.g., `123456`), the probability of this character is obtained through a Markov model [38]. Since the Markov model cannot deal with the low-frequency problem (i.e., data sparseness issue) well, the probability calculation of TarGuess-II also has the same problem when inserting characters.

Algorithm 1: Password generation algorithm.

```
Input: The probability threshold $\mathcal{T}$ of generated passwords
Output: Passwords set $\mathcal{X}$.
1  $char\_set = \{PrintableCharacters\} \cup \{E_s\}$;
2  $\mathcal{P}.push(B_s * ngram)$; /* $\mathcal{P}$ is the set of currently generated password
       prefixes. */
3  $\mathcal{L}[B_s * ngram] = 1$; /* $\mathcal{L}$ is the lookup table from currently generated
       password prefix to probability. */
4  while $!\mathcal{P}.empty()$ do
5      $current\_prefix = \mathcal{P}.pop()$;
6      $next\_char\_prob = RandomForest(current\_prefix)$;
7      for $c$ in $char\_set$ do
8          $new\_prob = \mathcal{L}[current\_prefix] * next\_char\_prob[c]$
9          if $new\_prob > \mathcal{T}$ then
10             if $c == E_s$ then
11                 $\mathcal{X}.append(current\_prefix)$;
12             else
13                 $\mathcal{P}.push(current\_prefix + c)$;
                   $\mathcal{L}[current\_prefix + c] = new\_prob$;
14 return $\mathcal{X}$
```

E Password generation algorithm

Algorithm 1 briefly formalizes the process of generating passwords with our RFGuess. First, we take the password prefix as the input of a trained random forest model to get the probability distribution *next_char_prob* of the next character. Then

Algorithm 2: PII matching algorithm.

```
Input: Passwords set $\mathcal{X} = \{pw_1, pw_2, ..., pw_n\}$.
Output: Passwords with corresponding PII matching structures ($\mathcal{P}$).
1  $match\_set = matchOrder(\mathcal{X})$;/* Get all PII structure
     representations and their corresponding frequency of set $\mathcal{X}$ in
     descending order. $match\_set$ is a priority queue.*/
2  $item = match\_set.pop()$; /* $item$ contains the PII structure
     representation and its frequency i.e., (structure,frequency). */
3  while $!match\_set.empty()$ and $item.frequency > 1$ do
4      for $pw_i$ in $\mathcal{X}$ do
5          $match\_pw_i = pwMatch(pw_i)$/* All PII structure
             representations of $pw_i$. */
6          if $item.structure$ in $match\_pw_i$ then
7              $\mathcal{P}.push((pw_i, item.structure))$;
8              $\mathcal{X}.remove(pw_i)$;
9          for $remain\_item.structure$ in $match\_pw_i$ do
10             $remain\_item.frequency$-=1;
11 while $!\mathcal{X}.empty()$ do
12     $pw = \mathcal{X}.pop()$;
13     $structure = shortMatch(pw)$;/* The shortest PII matching
         structure of $pw$. */
14     $\mathcal{P}.push((pw, structure))$
15 return $\mathcal{P}$
```

we insert a character to the end of the password prefix to generate a new password prefix. If the probability of the newly generated password prefix is greater than the threshold $\mathcal{T}$ (e.g., 1.2×10^{-8}) , it is inserted to set $\mathcal{P}$ for subsequent password generation; otherwise, it is discarded. If the inserted character is an end symbol (i.e., E_s), it means that a new password has been generated, and we insert it to the passwords set $\mathcal{X}$. In the initialization phase, $\mathcal{P}$ contains only one password prefix, namely $B_s * n$ (n-order beginning symbol), and the corresponding probability is 1. In the generation process, the password prefix in $\mathcal{P}$ will be continuously inserted and consumed. When the $\mathcal{P}$ becomes empty, the algorithm runs to the end. At this time, $\mathcal{X}$ contains all passwords with a probability greater than the predefined threshold $\mathcal{T}$.

F Evaluation of PII matching algorithm

Experimental evaluation. Considering that the Chinese dataset contains complete PII (which can better reflect the advantages of our PII matching algorithm), we take three Chinese datasets as examples, and compare the attack success rate of TarGuess-I [63] and Targeted-Markov model [61] (4-order) after using the two PII matching methods, respectively (the results can be seen in Table 11). We find that, within 100 guesses, our proposed PII matching algorithm can improve the guessing success rate of TarGuess-I [63] by 7%, and can improve Targeted-Markov [61] by 13%. For three English datasets, our PII matching algorithm has not much optimization effect. This is because: 1) Many PII attributes in English datasets are missing; 2) The three English PII-associated passwords are from more security-savvy users (i.e., hackers/administrators/tech-savvyers) [63]. Specifically, Rootkit is a hacker forum, and 000webhost is a free web hosting site and is mainly used by web administrators. Therefore, the users of both sites are likely to be more security-savvy than normal users, and this has been observed in Fig.13 of [63].

Table 11: The effect of PII matching algorithm (100 guesses).[†]

Targeted guessing model	TarGuess-I [63]		Targeted-Markov [61]	
Attack scenarios	Optimal	Original	Optimal	Original
50% PII-CSDN→50% PII-CSDN	27.90%	22.90%	27.01%	25.55%
50% PII-Dodo→50% PII-Dodo	19.10%	19.00%	20.33%	17.48%
50% PII-12306→50% PII-12306	20.30%	20.20%	20.91%	17.86%

[†]Optimal means using our new proposed PII matching algorithm, and original means using the leftmost&longest PII matching algorithm; Dodo=Dodonew.

Theoretical proof. We now prove the effectiveness of our proposed PII matching algorithm (in Sec. 4.2) in theory. Assume that there are N passwords in the password set D. For any two PII representations R_p and R_q, passwords that can be represented as R_p is denoted as S_p, and passwords that can be represented as R_q is denoted as S_q, where $|R_p| > |R_q|$. Then D can be divided into the following four sets.

$$\begin{aligned} A_{pq} = S_p \bigcap S_q,\ \ A_p = S_p - S_q, \\ A_q = S_q - S_p,\ \ A_o = D - S_p - S_q. \end{aligned} \tag{7}$$

The calculation of information entropy is given by: $H = \sum_{i=1}^{n} -p_i \cdot \log(p_i)$, so let

$$f(x) = -\frac{x}{D} \cdot \log(\frac{x}{D}), \tag{8}$$

where $f(x)$ is an upward convex function. Then the information entropy is expressed as H=$\sum_{i=1}^{m} f(c_i)$, where m is the number of representation tags, and c_i is the frequency of representation tags R_i. We now prove that when only two representations are considered, the information entropy is lower when the password is first represented as R_p with higher frequency than as R_q. Here, only the influence of R_p and R_q on the information entropy is considered, so the passwords that cannot be represented in these two ways (i.e., the set A_o) is not considered. For the set A_{pq}, A_p and A_q, if the password is first represented as R_p, and then represented as R_q, the information entropy is H_p=$f(|A_{pq}| + |A_p|) + f(|A_q|)$. If the password is first represented as R_q, the information entropy is $H_q = f(|A_{pq}| + |A_q|) + f(|A_p|)$. Considering $|A_{pq}| + |A_p| = |R_p| > |R_q| = |A_{pq}| + |A_q|$, we get $|A_p| > |A_q|$.

Let $g(x) = f(x) - f(x + |A_{pq}|)$, where $x > 0$, and take the derivative of $g(x)$, we get

$$\begin{aligned} g'(x) &= f'(x) - f'(x + |A_{pq}|) \\ &= \frac{\frac{-1-ln(\frac{x}{|D|})}{|D|} - \frac{-1-ln(\frac{x+|A_{pq}|}{|D|})}{|D|}}{ln2} \\ &= \frac{ln(\frac{x+|A_{pq}|}{|D|}) - ln(\frac{x}{|D|})}{|D| \cdot ln2} > 0. \end{aligned} \tag{9}$$

Since the derivative of $g(x)$ is greater than 0, $g(x)$ is a monotonically increasing function. And $|A_p| > |A_q|$, we have

$g(|A_p|) > g(|A_q|)$, namely

$$g(|A_p|) = f(|A_p|) - f(|A_p| + |A_{pq}|) > g(|A_q|) \\ = f(|A_q|) - f(|A_q| + |A_{pq}|). \quad (10)$$

By shifting the term, we get

$$f(|A_q|)+f(|A_p|+|A_{pq}|) < f(|A_p|)+f(|A_q|+|A_{pq}|), \quad (11)$$

that is, $|H_p| < |H_q|$. Therefore, when only two representations are considered, the information entropy is lower when the password is first expressed as R_p with higher frequency than as R_q. It can be seen from this conclusion that preferential representation as R_1 can make the information entropy the lowest. According to the algorithm proposed in Sec. 4.2, the highest frequency representation taken out for the first time is R_1. Then the current highest frequency representation is taken out in each round. That is, the representation taken out each time can be used as a priority representation. As a result, each round of selection is the current optimal choice, and the representation obtained at the end of the algorithm can be regarded as an approximately optimal solution.

Overhead. Although computational complexity is not particularly important for online guessing, we have tested the time consumption of our optimal PII matching algorithm. More specifically, it takes about 440s on a common server (CPU: Xeon Silver 4200R; System: Ubuntu 20.04) to complete PII matching on 50% of the Dodonew-PII dataset (about 80,000 pieces of data), which is acceptable.

G Structure-level transformation behavior statistics

Through the analysis of the problems in TarGuess-II [63], we find that the focus is whether the behavior of inserting and deleting structures is related to the password itself. Taking the CSDN→126 dataset as an example (which is a dataset composed of password pairs matched through email), we count the similar but different password pairs among them ("similar" here means the similarity score s is greater than 0 and less than 0.5, and it is calculated as $s = 1 - \text{EditDistance}(pw1, pw2)/\max(|pw1|, |pw2|).$). More specifically, there are a total of 25,917 items, accounting for 26.47% of the entire dataset. We have made statistics on the insertion and deletion of structural behaviors of them, and the top-ten frequent ones are shown in Table 12.

Table 12 shows that the tenth-ranked reuse behavior (i.e., insert or delete the string "11") only occurs 49 times, which makes it challenging to learn the behavior of inserting and deleting password structures based only on the number of occurrences in the dataset. To address this issue, we divide the probability of structure-level transformation into two parts in RFGuess-Reuse: the probability of the structure-level transformation, and the probability of which specific structure is

Table 12: Structure-level insertion/deletion statistics.

Insertions/Deletions	Position	Frequency	Example
`a`	Prefix	264	`a3221041 →3221041`
`123`	Suffix	196	`cwhwan123→cwhwan`
`a`	Suffix	154	`4231294a →4231294`
`1`	Suffix	93	`wuchunlei→wuchunlei1`
`qq`	Suffix	87	`qq849210 →849210`
`aa`	Suffix	79	`5631842aa→5631842`
`aa`	Prefix	71	`aa123321 →123321`
`.`	Prefix	56	`3232334. →3232334`
`abc`	Suffix	53	`81983064 →81983064abc`
`11`	Suffix	49	`resing11 →resing`

Table 13: Structure-level transformation in each length.

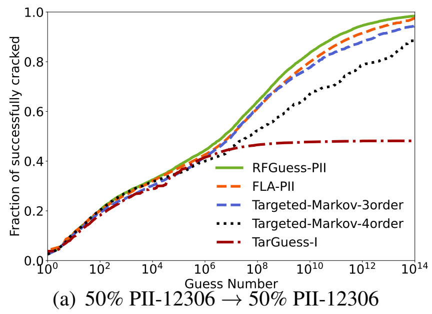
Password length	Tail insertion	Tail deletion	Head insertion	Head deletion
3	0	0	0	0
4	3	0	11	0
5	14	0	128	0
6	1757	0	2274	0
7	1853	3	2339	2
8	396	1010	380	1223
9	178	1141	96	1667
10	95	1061	42	1169
11	37	429	37	556
12	23	358	2	373
13	5	159	1	166
14	3	131	1	115
15	2	39	0	19
16	0	30	0	20

performing on structure-level transformation.

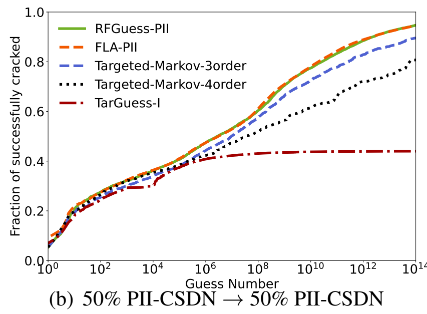
For the first part, we consider the correlation between password length and structure-level transformation behaviors. We still take the CSDN→126 dataset as an example, and the statistics are summarized in Table 13. We find that the behavior of structure-level transformation has a great relationship with the password length. More specifically, passwords with lengths of 6 and 7 are more likely to be inserted into new structures, while passwords with lengths of 8∼10 are more tend to delete existing structures. Therefore, the probability of structure-level transformation can be obtained by statistics of corresponding transformation behaviors of passwords with different lengths in the training set. As for the transformation probability of a specific structure, it can be learned in a relatively large password set through PCFG [65].

H Feature importance

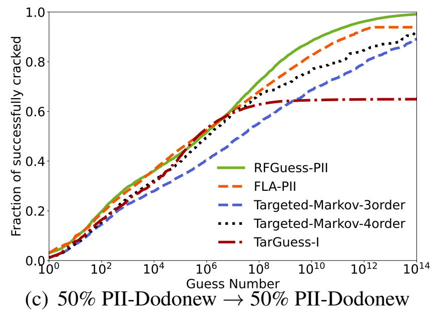
Although there is a slight difference in feature importance between the Chinese and English datasets (see Fig. 12), they are still very similar overall (the value of the cosine similarity between Chinese and English datasets is 0.98). Furthermore, there is almost no difference in the feature importance of the same language datasets (the cosine similarity within the Chinese and English datasets are both 0.99). Therefore, we calculate the average of the feature importance scores of the four datasets for observation (see Fig. 14).



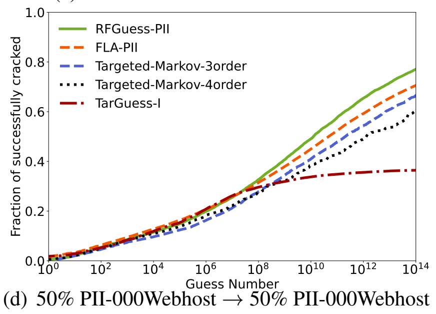
(a) 50% PII-12306 $\rightarrow$ 50% PII-12306



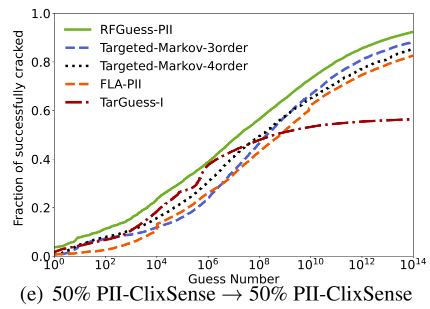
(b) 50% PII-CSDN $\rightarrow$ 50% PII-CSDN



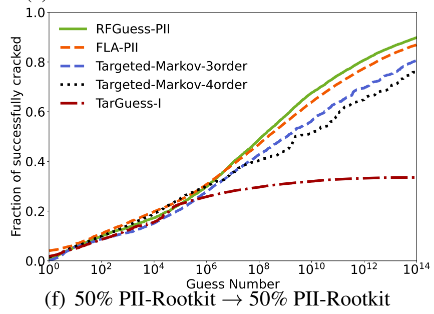
(c) 50% PII-Dodonew $\rightarrow$ 50% PII-Dodonew



(d) 50% PII-000Webhost $\rightarrow$ 50% PII-000Webhost



(e) 50% PII-ClixSense $\rightarrow$ 50% PII-ClixSense



(f) 50% PII-Rootkit $\rightarrow$ 50% PII-Rootkit

Figure 10: Experiment results of our RFGuess-PII against other PII-models (i.e., TarGuess-I [63], Targeted-Markov [61], and FLA [39]-PII) in the targeted guessing scenarios. The guessing success rates of our RFGuess-PII are always the best within 10^{14} guesses (through the Monte-Carlo simulation [20]).

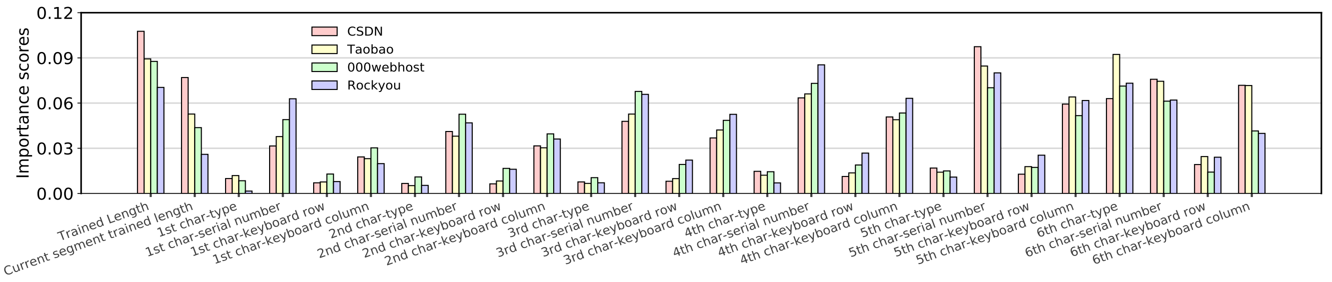


Figure 11: Feature importance (for four datasets). The Y-axis represents the proportion of the feature as the model classification rule: It reflects the importance of the feature. Thus, the larger the value, the higher the importance, and the sum of all feature importance scores for one dataset is one.

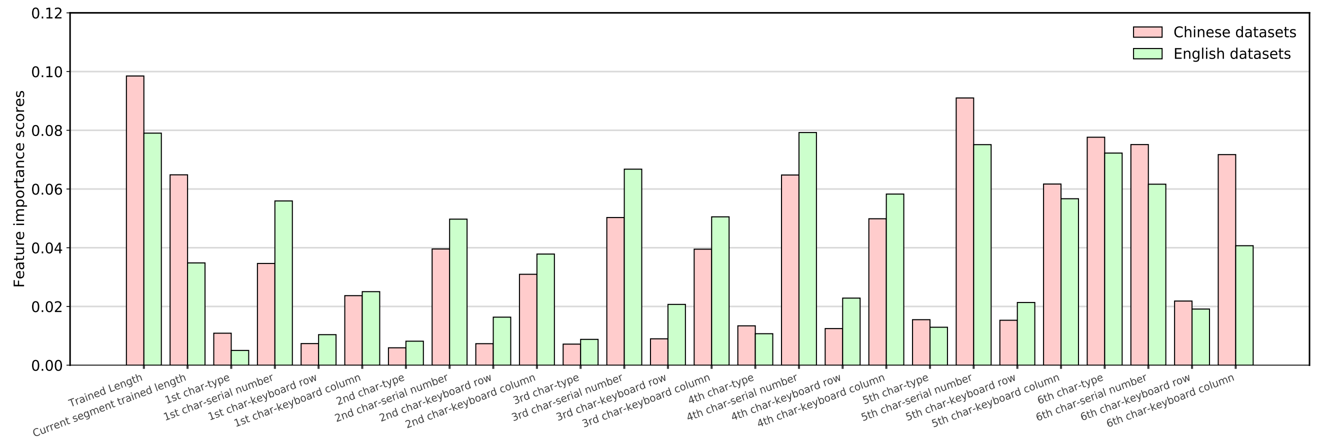
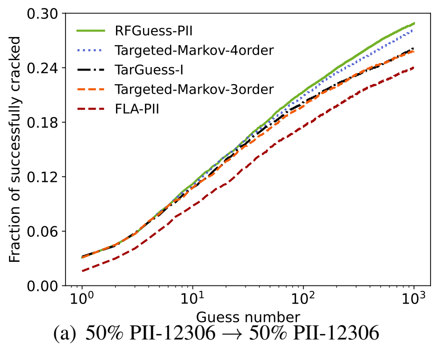
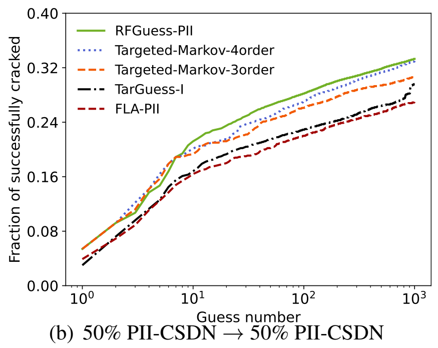


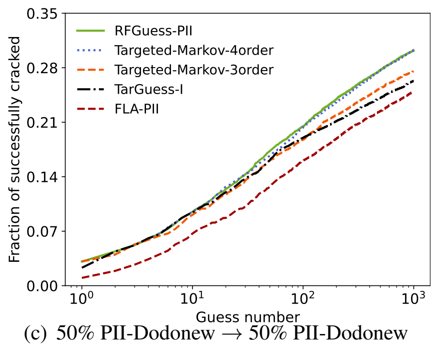
Figure 12: Feature importance. The green bar is the average of the feature importance scores of the 000Webhost and Rockyou datasets (English datasets); the red bar is the average of the Taobao and CSDN datasets (Chinese datasets). Overall, the length of the trained characters (position of the character in a password) and the characters close to the predicted target character are more important in the Chinese datasets. While in English datasets, characters near the middle position (relative to the order) are more important (third and fourth character). We calculate the cosine similarity of feature importance scores between the two language and find this value to be 0.98. Besides, the cosine similarity scores in the same language datasets is greater than 0.99. This shows that these two scores are very similar, indicating that language has little effect on feature importance scores, so we further calculate the average feature importance scores in Fig. 14.



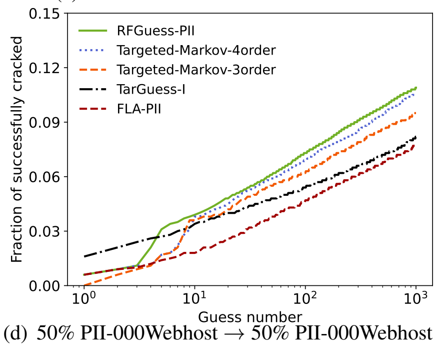
(a) 50% PII-12306 → 50% PII-12306



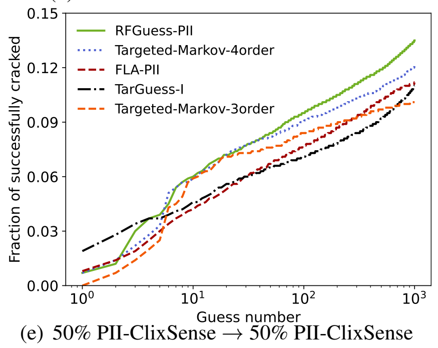
(b) 50% PII-CSDN → 50% PII-CSDN



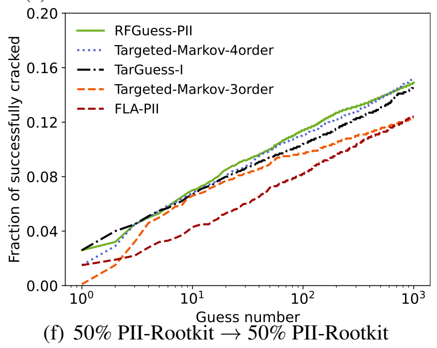
(c) 50% PII-Dodonew → 50% PII-Dodonew



(d) 50% PII-000Webhost → 50% PII-000Webhost



(e) 50% PII-ClixSense → 50% PII-ClixSense



(f) 50% PII-Rootkit → 50% PII-Rootkit

Figure 13: Guessing performance of our RFGuess-PII in comparison with other targeted guessing models (i.e., TarGuess-I [63], Targeted-Markov [61], and FLA [39]-PII). One can see that, when *explicitly* generating 10^3 guesses, our RFGuess-PII matches or beats other most effective PII-models.

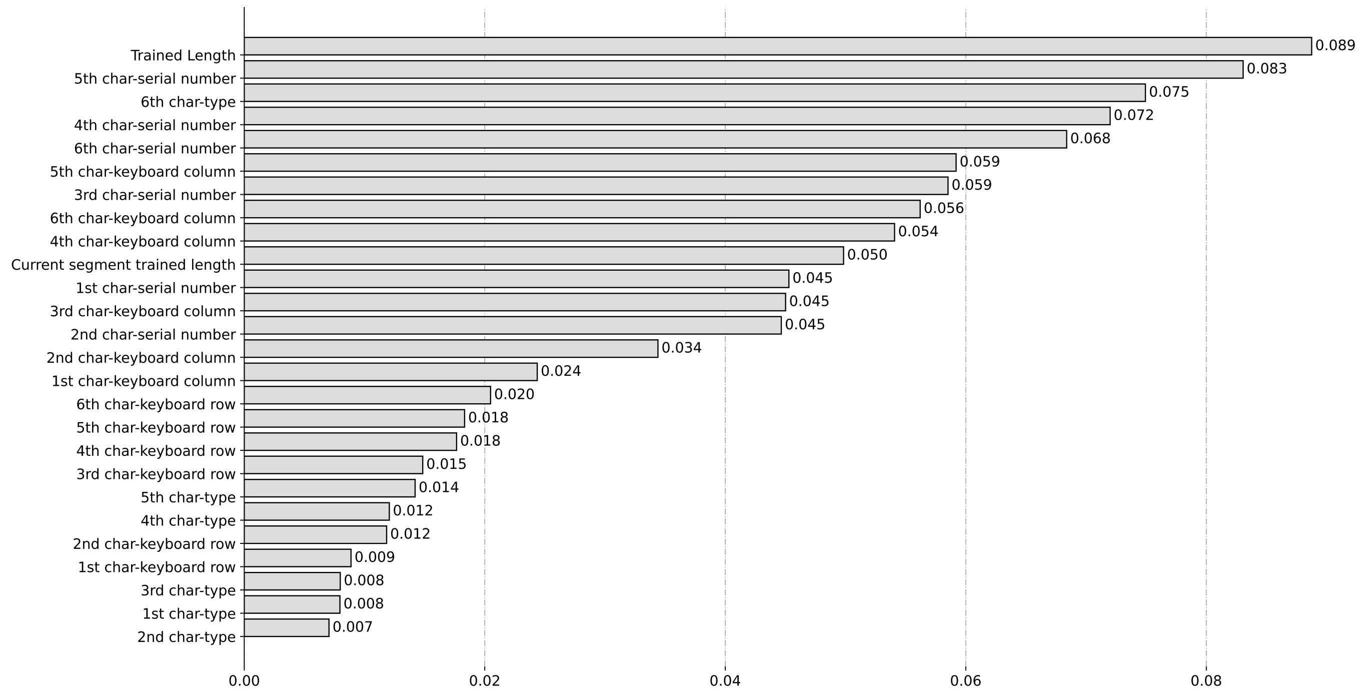
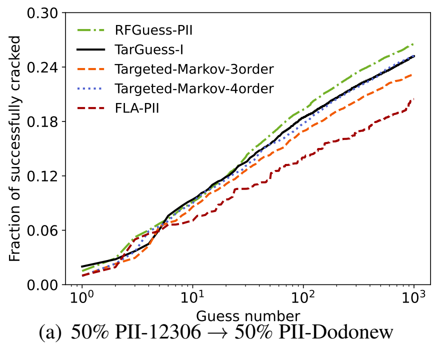
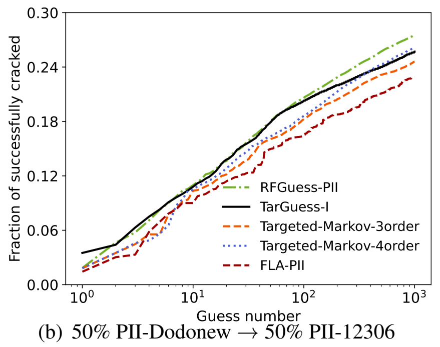


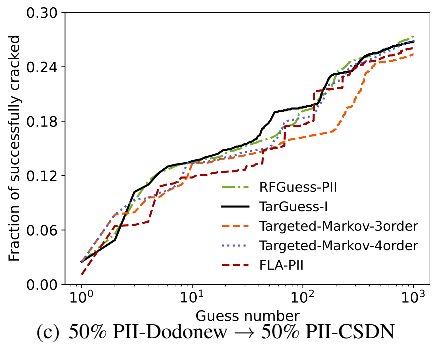
Figure 14: Feature importance (average). We sort the average of feature importance scores of two Chinese and two English datasets. Among these features, the serial number feature (e.g., `a` is the first in alphabetic types `a`∼`z`, 0 is the first of digits 0∼9) and the keyboard column number feature (e.g., `d` is located in the third column of the keyboard) are more effective, while the type of character (whether this character is a letter, digit or special character) and the current segment trained length (position of the character in the segment) are relatively unimportant, and the feature of keyboard row number has little effect on the model fitting. Since random forest can filter features, existing of some unimportant features will not affect the fitting ability of the model. In particular, relatively unimportant features can be selectively removed before training. For example, our experiments show that if the relatively unimportant 10-dimensional features are removed, the model training speed is improved by 30%. However, the maximum decrease in success rates is no more than 0.4% compared with the original.



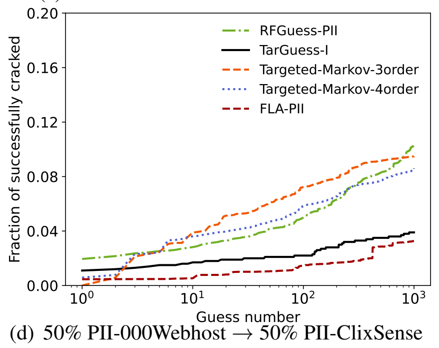
(a) 50% PII-12306 $\rightarrow$ 50% PII-Dodonew



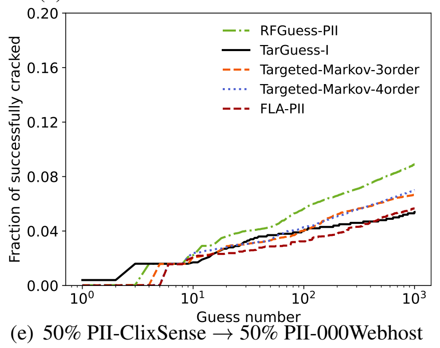
(b) 50% PII-Dodonew $\rightarrow$ 50% PII-12306



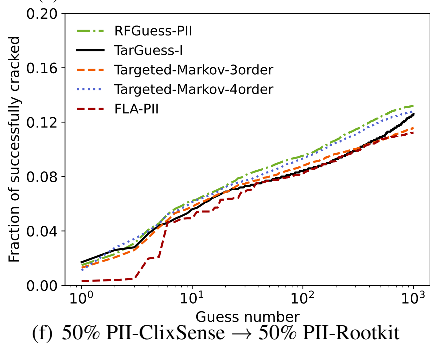
(c) 50% PII-Dodonew $\rightarrow$ 50% PII-CSDN



(d) 50% PII-000Webhost $\rightarrow$ 50% PII-ClixSense

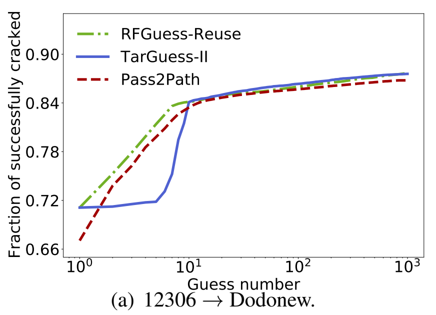


(e) 50% PII-ClixSense $\rightarrow$ 50% PII-000Webhost

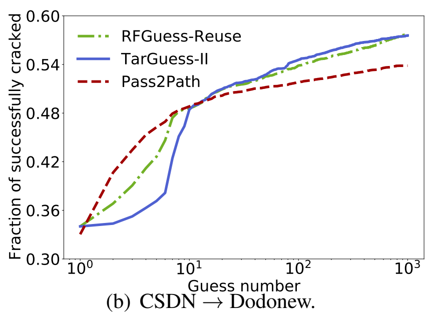


(f) 50% PII-ClixSense $\rightarrow$ 50% PII-Rootkit

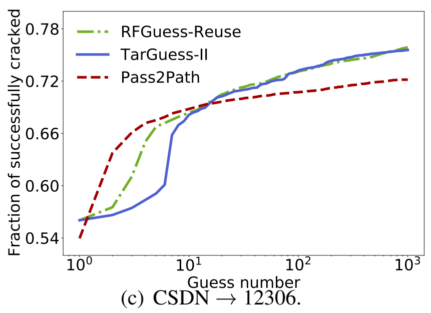
Figure 15: Guessing performance of our RFGuess-PII in comparison with other targeted guessing models (i.e., TarGuess-I [63], Targeted-Markov [61], and FLA [39]-PII) in *cross-site* guessing scenarios. We can see that, when *explicitly* generating 10^3 guesses, our RFGuess-PII is highly effective, and it matches or beats other most effective PII-models in most cases. This indicates that our RFGuess-PII has satisfactory generalization ability.



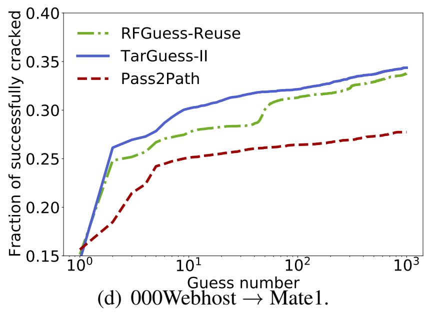
(a) 12306 $\rightarrow$ Dodonew.



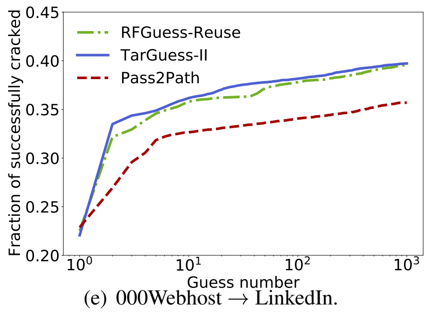
(b) CSDN $\rightarrow$ Dodonew.



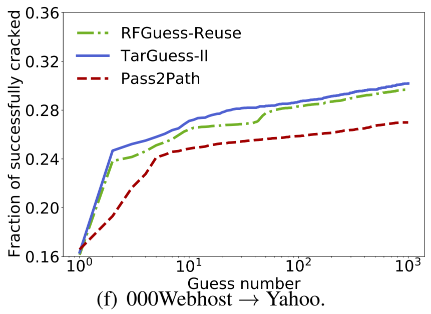
(c) CSDN $\rightarrow$ 12306.



(d) 000Webhost $\rightarrow$ Mate1.



(e) 000Webhost $\rightarrow$ LinkedIn.



(f) 000Webhost $\rightarrow$ Yahoo.

Figure 16: Guessing performance of our RFGuess-Reuse in comparison with other password reuse-based models (i.e., Pass2Path [44] and TarGuess-II [63]). One can see that, our RFGuess-Reuse and Pass2Path [44] outperform TarGuess-II [63] within 10 guesses in Chinese datasets. While in English datasets, TarGuess-II and our RFGuess-Reuse outperform Pass2Path. It's likely because Pass2Path is based on deep learning techniques and is more suitable for extremely large training sets (e.g., the 1.4 billion-sized 4iQ), and it does not consider users' vulnerable behavior of choosing popular passwords.