A Security Analysis of Honeywords

Ding Wang, Haibo Cheng, Ping Wang
Peking University
{wangdingg, chenghaibo, pwang}@pku.edu.cn

Jeff Yan
Linköping University
jeff.yan@liu.se

Xinyi Huang
Fujian Normal University
xyhuang81@gmail.com

Abstract—Honeywords are decoy passwords associated with each user account, and they contribute a promising approach to detecting password leakage. This approach was first proposed by Juels and Rivest at CCS’13, and has been covered by hundreds of medias and also adopted in various research domains. The idea of honeywords looks deceptively simple, but it is a deep and sophisticated challenge to automatically generate honeywords that are hard to differentiate from real passwords. In Juels-Rivest’s work, four main honeyword-generation methods are suggested but justiﬁed by heuristic security arguments.

In this work, we for the ﬁrst time develop a series of practical experiments using 10 large-scale datasets, a total of 104 million real-world passwords, to quantitatively evaluate the security that these four methods can provide. Our results reveal that they all fail to provide the expected security: real passwords can be distinguished with a success rate of 29.29%~32.62% by our basic trawling-guessing attacker, but not the expected 5%, with just one guess (when each user account is associated with 19 honeywords as recommended). This ﬁgure reaches 34.21% to 49.02% under the advanced trawling-guessing attackers who make use of various state-of-the-art probabilistic password models. We further evaluate the security of Juels-Rivest’s methods under a targeted-guessing attacker who can exploit the victim’s personal information, and the results are even more alarming: 56.81% to 67.98%. Overall, our work resolves three open problems in honeyword research, as deﬁned by Juels and Rivest.

I. INTRODUCTION

Passwords firmly remain the most prevalent method for user authentication and are likely to keep their place in the foreseeable future, despite their notorious defects in both security and usability [3], [6], [30]. An inherent limitation of the existing password-based authentication systems is that the server need maintain a sensitive file comprised of passwords of all registered users, and this ﬁle provides attackers/insiders with a rich target for compromise. These days it is no news to hear that high-proﬁle web services have been compromised and millions of passwords were leaked, and some quite recent victims include Yahoo [18], Dropbox [20], Last.fm, LinkedIn, Weebly [27], and MySpace [31], to just name a few.

What’s most disturbing is that, these breaches were often detected only when the attackers had well exploited the data and then posted (or sold) it online, which is generally months, even years after the breach initially occurred. For instance, the most recent catastrophic password breach revealed on October 2017 involves the entire 3 billion Yahoo userbase, yet the breach actually had occurred four years before [18]; The Weebly password breach revealed in Oct. 2016 involves 43 million users, and users were asked to change passwords, yet the breach actually had occurred eight months before [27]; The 68 million Dropbox breach occurred in 2012, yet users are asked to change passwords four years later when the stolen data surfaced in the public in May 2016 [20]; The detection of the 360 million Myspace dataset took eight years: this dataset was compromised in 2008 [31] yet only detected until May 2016, when its sale information was posted online. The 2016 data breach report by Verizon also reveals that, among the 2,260 breaches investigated, over 85% were first detected by external parties, 91% took weeks to detect, 70% took months even years to detect [13]. All this highlights the imperative need for active, timely password-breach detection methods to enable responsive counter-actions.

Even if the leaked passwords are stored in salted-hash as standard practice, this poses no real obstacle for an attacker to recover them by an overwhelming percentage by using modern machine-learning based cracking algorithms [24], [30] and common hardware like GPUs (see [12]). Sophisticated hash functions (e.g., bcrypt) are designed to slow down the attacker’s cracking process by the same factor as the honest server is willing to spend on password veriﬁcation, but the attacker is likely to be better equipped with dedicated password-cracking hardware [15]. Thus, once the password hash ﬁle is obtained by an attacker, it is realistic to assume that most of them can be ofﬂine guessed. In addition, since the attacker only need perform guessing locally, slow hash functions do not facilitate the detection of password-ﬁle leakage.

Recently, there have been approaches proposed to completely eliminate the possibility of ofﬂine password guessing: (1) using a machine-dependent function (e.g., ErsatzPasswords [2]); (2) employing distributed cryptography (e.g., threshold password-authenticated secret sharing [7]); and (3) using external password-hardening services (e.g., Phoenix [23]). However, all these approaches require substantial changes to the server-side authentication systems. Besides, the ﬁrst approach does not support backing up password hash ﬁles in a distributed manner, and thus it is unsuitable for Internet-scale sites due to its poor scalability; The second necessitates client-side system changes, which is not user-friendly and widely deemed not desirable; The third is subject to a single point of failure and may leak user behavior information to external parties.

A more promising approach to improving the situation, ﬁrst proposed by Juels and Rivest in 2013 [21], is to introduce decoy passwords (called “honeywords”) to associate with each user’s account (see Fig. 1). Their intriguing idea is that, even if an attacker A has stolen the password ﬁle and recovered all the passwords, she has to ﬁrst tell apart the user’s real password from a set of k − 1 passwords, she has to tell apart the user’s real password from a set of k − 1 passwords, she has to tell apart the user’s real password from a set of k − 1 passwords, she has to tell apart the user’s real password from a set of k − 1 passwords, she has to tell apart the user’s real password from a set of k − 1 passwords, she has to tell apart the user’s real password from a set of k − 1 passwords, she has to tell apart the user’s real password from a set of k − 1 passwords intended generated honeywords. These k − 1 honeywords and the real password are uniﬁed called k “sweetwords”. When honeywords are well generated, to ﬁgure out the real
A. Challenges and motivations

A main challenge, as discussed in [7], [14], [21], is how to generate honeywords that cannot be easily distinguished from real passwords. In Juels-Rivist’s work [21], five honeyword-generation methods are suggested: four for the legacy user-interface (UI) and one for the modified-UI, and these legacy-UI methods are preferred due to usability advantages. Thus, we also focus on honeyword generation for the legacy-UI.

All the Juels-Rivist methods are random-replacement based, and thus inherently unable to resist semantic-aware attackers as shown by some typical counter-example passwords (e.g., bound007 and john1981) in the literature [10], [14]. Yet, to what extent these methods are effective (or ineffective) against semantic aware/un-aware attackers has never been theoretically or empirically quantified.

Most prior art on honeywords (see [10], [14], [21]) merely provide heuristic security arguments when evaluating a method, neither a rigorous theoretical analysis nor an empirical evaluation with real-world datasets was given. As we will show with realistic experiments, all the honeyword-generation methods in [21] largely fail to provide the expected security, and real passwords can be distinguished with high success rates. An improvement named “honeyindex” was given in [14], yet its evaluation is still heuristic-based and it suffers from the “peeling-onions style” distinguishing attack and critical deployment issues.1 Golla et al. [17] empirically showed how the Kullback-Leibler divergence (KLD) metric can be used to distinguish a real password distribution from a decoy one, yet this metric is unsuitable for the honeyword-attacking scenario where we need to distinguish the single, real password from a set of distinct sweetwords. In all, existing honeyword methods (see [14], [21]) are short of a sound evaluation, and whether they can achieve the claimed security is unknown.

What’s more, little attention has been given to the analysis of honeyword security under targeted guessing attackers. A targeted attacker (see [30]) exploits not only users’ behavior of selecting popular passwords (e.g., abc123, iloveyou) but also the victims’ personally identifiable information (PII) like name and birthday. To our knowledge, existing works [10], [14], [21] mainly consider trawling guessing attackers (see [5], [24], [32]) who exploit only users’ behavior of choosing popular passwords. However, a large fraction of users build passwords using their own PII. For instance, as we will show in Sec. II-A, 51.43% of Dodonew normal users, 27.16% of Qatar national bank users and 12.76% of Rootkit hackers employ their own, common PII to build passwords.

This user behavior is increasingly vulnerable, as users’ PII can now be easily learned from social networks and unending data breaches [16], [18], [26]. For example, the recent large-scale data breach in Oct. 2017 involves 3 billion Yahoo users, with their names, phone numbers, birthdates leaked [18]; In July 2017, the largest credit-reporting agency in America, Equifax, leaked PII data about half of the US population, including names, birthdates, SSN [25]; The April 2016 PII breach against Turkishens involves 64% of the total population [1]. As the targeted guessing threat is becoming increasingly realistic, honeywords shall be evaluated under this new threat.

B. Our contributions

In this work, we make the following key contributions:

- **Trawling guessing attacks.** We for the first time develop a series of experiments using large-scale real password data to evaluate the four honeyword-generation methods in [21], and find that they all fail to provide the expected security. We show that real passwords can be distinguished with a success rate of 29.29%–32.62%, but not the expected 5%, with just one guess under a basic, trawling guessing attacker (when each user account is associated with 19 honeywords as recommended, i.e., the parameter $k = 20$ [21]). This figure reaches 34.21%–49.02% under the advanced trawling guessing attackers who make use of various state-of-the-art probabilistic password cracking models (e.g., Markov [24] and PCFG [32]).

- **Targeted guessing attacks.** To see how Juels-Rivist’s methods perform under semantic-aware attackers, we for the first time evaluate the security of honeywords by performing targeted guessing attacks. We show that real passwords can be distinguished with a success rate of 56.8%–67.9% by performing just one guess (when $k = 20$), if the attacker knows some common personal information like name and birthday of the victim user. This answers the question of “how well can targeted attacks help identify users’ passwords for particular honeyword-generation methods” as left in [21].

- **Extensive evaluation.** Our experiments build on various leading probabilistic password cracking models

---

1The companion site: https://github.com/pkusec/rethinking-honeywords
We demonstrate that though QNB users speak English, their passwords have little correlation with the other four English userbases. This is expected: QNB users are mainly from Chinese, and we manage to recover 97,580 (81.69%) of them. The QNB dataset was leaked from the Qatar national bank, which is located in Middle East, in April 2016 [19]. To our knowledge, this is the first real-world banking-password dataset that is explored in an academic study.

Particularly, four password datasets (i.e., 12306, ClixSense, Rootkit and QNB) are associated with various kinds of PII as shown in Table II. To facilitate a more comprehensive empirical analysis of honeyword security under targeted attackers, we further match the non-PII-associated datasets with these PII-associated datasets by using email. As a result, this produces nine PII-associated password datasets as shown in the first row in of Table III: (1) the four Chinese ones are obtained by matching the corresponding non-PII-associated dataset with 12306; (2) the four US-English ones are: PII-Rootkit, PII-ClixSense, and two other ones obtained by matching 000webhost and Yahoo with ClixSense, respectively; and (3) PII-QNB, which is QNB itself. Note that, the non-PII-associated US-English dataset Rockyou includes neither email nor NID, and thus it cannot be matched.

We further employ two auxiliary PII datasets (i.e., Hotel and 51job) to augment each Chinese password dataset to obtain more PII-associated accounts by matching email or NID. We note that many PII-associated accounts miss some important PII attributes, and they can be supplemented by using the auxiliary PII datasets.

Table III demonstrates that users love to employ their personal information to build passwords. Here we measure the PII usages by using the type-based PII-tagging approach proposed in [30], for it has been shown much more accurate than other approaches. As high as 36.95%–51.43% of Chinese users employ at least one of their six kinds of PII to construct passwords, while this figure for US-English users is 12.76%–29.94% and for ME-English users is 27.16%. In comparison, the PII-associated US-English users show a more secure behavior in PII usages. This is expected, because they are all Rootkit hackers or ClixSense online cash-earning users (as resulted from email matches). In other words, our PII-associated US-English users well represent technique-savvy users. Our results show that a large number of users build passwords using PII, and thus sound honeyword-generation methods shall take this user behavior into account.

We highlight that though QNB users speak English, their passwords have little correlation with the other four English userbases. This is expected: QNB users are mainly from Middle East. Thus, we divide users into three groups: Chinese, US-English and ME-English. In all, our corpus is new, comprehensive and well represents real-world password distributions, and to our knowledge, it is also among the largest and most diversified ones ever collected for use in a password study.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Language</th>
<th>Items num</th>
<th>Types of PII useful for this work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hotel</td>
<td>Chinese</td>
<td>29,031,426</td>
<td>Name, Birthday, Phone, NID</td>
</tr>
<tr>
<td>51job</td>
<td>Chinese</td>
<td>2,327,573</td>
<td>Email, Name, Birthday, Phone</td>
</tr>
<tr>
<td>T2306</td>
<td>Chinese</td>
<td>129,307</td>
<td>Email, User name, Name, Birthday</td>
</tr>
<tr>
<td>ClixSense</td>
<td>English</td>
<td>2,222,045</td>
<td>Email, User name, Name, Birthday</td>
</tr>
<tr>
<td>Rootkit</td>
<td>English</td>
<td>79,580</td>
<td>Email, User name, Name, Birthday</td>
</tr>
<tr>
<td>QNB</td>
<td>English</td>
<td>77,790</td>
<td>Email, User name, Name, Birthday</td>
</tr>
</tbody>
</table>

*ND=A National identification number, e.g., social security number.

Two of our datasets were leaked in MD5 hash, and we manage to recover an overwhelming fraction of them by using various trawling guessing models [24] as well as the targeted guessing model TarGuess [30] on a common PC with GPU in one week. More specially, Rootkit initially consists of 71,228 passwords and we manage to recover 97.46% of them; QNB initially contains 97,415 passwords [19], and we manage to recover 79,580 (81.69%) of them. The QNB dataset was leaked from the Qatar national bank, which is located in Middle East, in April 2016 [19]. To our knowledge, this is the first real-world banking-password dataset that is explored in an academic study.
and ensures distributed security system, software stacks, security mechanisms and so on, in different administrative domains, runs different operating computer system of minimalist design. It may be placed is kept on the honeychecker which is a separate, hardened user or even per-sweetword manner. The record ($S_i$) authentication is that, ($S_{PW_i}$)

As shown in Fig. 3, honeyword system and can be possibly launched against the honeyword system. honeywords are in place. Without loss of generality, we consider one of these $U_i$'s policy, $S_{PW_i}$, then the honeychecker signals to $U_i$'s account until the user resets a new password; otherwise, it suggests that a login with honeyword is $\star$.

Table III. Percentages of Users Building Passwords with Their Own Heterogeneously Personally Identifiable Information (PII),

<table>
<thead>
<tr>
<th>Percentage of Users Building Passwords with Their Own Personal Identifiable Information</th>
<th>PII-Taiyun</th>
<th>PII-Dodonew</th>
<th>PII-12306</th>
<th>PII-CSND</th>
<th>PII-Rootkit</th>
<th>PII-000web</th>
<th>PII-ClixSense</th>
<th>PII-Yahoo</th>
<th>PII-QNB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name (7 subtypes; e.g., johnsmith, johnsmith, birthday (10 subtypes; e.g., 01171981, 1981, 0117, 1981, email_prefix (3 subtypes; e.g., moon123, moon, 123)</td>
<td>9.1%</td>
<td>23.82%</td>
<td>23.83%</td>
<td>16.71%</td>
<td>9.32%</td>
<td>12.06%</td>
<td>9.14%</td>
<td>6.95%</td>
<td>5.25%</td>
</tr>
<tr>
<td>Phone # (3 subtypes; e.g., 4153022671, 415, 2671)</td>
<td>6.31%</td>
<td>6.61%</td>
<td>6.85%</td>
<td>7.35%</td>
<td>7.89%</td>
<td>5.14%</td>
<td>3.86%</td>
<td>4.91%</td>
<td>7.61%</td>
</tr>
<tr>
<td>NID (3 subtypes; e.g., 620915357, 620, 5357)</td>
<td>3.09%</td>
<td>10.12%</td>
<td>7.46%</td>
<td>3.45%</td>
<td>5.74%</td>
<td>5.29%</td>
<td>3.82%</td>
<td>2.76%</td>
<td>1.53%</td>
</tr>
<tr>
<td>Total personal information usage (all above)</td>
<td>30.93%</td>
<td>51.19%</td>
<td>48.03%</td>
<td>41.76%</td>
<td>29.84%</td>
<td>24.31%</td>
<td>18.76%</td>
<td>22.41%</td>
<td>19.66%</td>
</tr>
</tbody>
</table>

*The specific sub-types of each kind of PII we consider are the same with that of TarGuess-I [30]. For instance, 23.82% in the top left corner means 23.82% of the 161,517 PII-associated Donow users employ at least one of their 7 sub-types of name information to build passwords.

B. Security model

In this paper, we mainly focus on the security that can be provided to the underlying user authentication system when honeywords are in place. Without loss of generality, we consider the most general case, i.e. the client-server architecture. We discuss the primary kinds of attacks which exploit honeywords and can be possibly launched against the honeyword system.

The honeyword system. As shown in Fig. 1, there are four entities involved: a user $U_i$, an authentication server $S$, a honeychecker, and an attacker $A$. User $U_i$ has registered an account ($ID_i, PW_i$) at $S$, and some PII may also be needed (e.g., registration requires name, birthday, phone and gender). On the server side, what’s different from the traditional password authentication is that, $S$ conducts a command $\text{Gen}(k; PW_i)$: $S$ generates a list of $k$-1 distinct, plausibly looking decoy passwords (called honeywords) to associate with $U_i$’s account, where $k = 20$ as recommended in [21]. Password $PW_i$ and its $k$-1 honeywords are unifiedly called $k$ sweetwords. Generally, there are two broad kinds of honeyword methods: random replacement based (e.g., tweaking tail [21]) and password-model based (see Sec. III-E).

Now $U_i$’s account record in $S$ can be represented as ($ID_i, SW_i$), where $SW_i = (sw_{i1}, sw_{i2}, \ldots, sw_{ik})$. Exactly one of these $k$ sweetwords, denoted by $sw_{ij}$, equals $U_i$’s password $PW_i$. Let $C_i$ denote the correct index of $U_i$'s password in the sweetword list $SW_i$, and thus $C_i = j$. The $k$ sweetwords on $S$ shall be hashed, with salting in a per-user or even per-sweetword manner. The record ($ID_i, C_i$) is kept on the honeychecker which is a separate, hardened computer system of minimalist design. It may be placed in different administrative domains, runs different operating system, software stacks, security mechanisms and so on, and ensures distributed security [7], [21]. The honeychecker is not publicly accessible, and it only interacts with $S$ by using a “dedicated and/or encrypted and authenticated” [21] communication channel.

When $U_i$ logs in with ($ID_i, PW_i^*$), $S$ first looks up the list $SW_i$ and sees whether there is one element (with index $C_i^*$) that matches $PW_i^*$. If not, the login is rejected. Otherwise, $S$ submits a command $\text{Check}(ID_i, C_i^*)$ to the honeychecker. If $C_i^* = C_i$, then the honeychecker signals to $S$ to accept $U_i$. Otherwise, it suggests that a login with honeyword is attempted, and an alarm is raised to $S$. Depending on the alarm policy, $S$ may take an appropriate action, such as: 1) accept the login but on a honeypot system, and more stringently monitor the user’s activities; 2) if the number of honeyword login attempts against $U_i$’s account exceeds a pre-defined threshold $T_1$ (e.g., 3), lock out $U_i$’s account until the user resets a new password; or 3) shut down the computer system and require all users to reset new passwords, if all users’ total honeyword login attempts exceeds a pre-defined threshold $T_2$. The value of $T_2$ depends on the system’s risk analysis and is out of our scope. Since the system has to balance honeyword-distinguishing attacks and DoS attacks, $T_2$ shall not be too small or too large, and without loss of generality, we set $T_2 = 10^4$.

Honeyword distinguishing attacker. As mentioned earlier, the most essential security goal of any honeyword method from $U_i$’s real password $PW_i$. This goal corresponds to the honeyword distinguishing attacker $A$ as shown in Fig. 1, who aims to tell the real password apart from the $k$-1 honeywords associated with $U_i$’s account by using $S$ as a querying oracle. $A$’s honeyword login attempts will be detected by the honeychecker, and if the number of such attempts against $U_i$’s account exceeds the per-user threshold $T_1$ (e.g., 3), $A$ will raise the alarm on $U_i$’s account. $A$ will also raise the system-wide alarm if her login attempts exceed the threshold $T_2$ (e.g., $10^5$). Thus, $A$’s honeyword login attempts shall be as few as possible.

We assume that $A$ has somehow already got access to the server $S$’s password hash file, knows the algorithm under which the honeywords are generated and hashed, and is armed with all the publicly available information (e.g., various publicly leaked password datasets and the target site’s password policy). We call this attacker a type-$A_1$ attacker. As said earlier, $A$ may also obtain the victim $U_i$’s PII. We call this advanced attacker a type-$A_2$ attacker. These assumptions about $A$’s capabilities are indeed realistic, yet they are often only implicitly made (or missed) in previous studies [7], [14], [21].

Other attackers. As discussed in [21], other valid threats facing honeywords include attacking the honeychecker system, denial-of-service (DoS) attacks that game the honeyword system and intersection attacks where an attacker exploits user passwords reused across different systems. Since these attacks have little relevance to the strength of a honeyword-generation method, and thus they are beyond our scope.

C. Evaluation metrics

Juels and Rivest [21] proposed a notion of $\epsilon$-flat to measure the security of a honeyword generation method. $\epsilon$-flat denotes the maximum success rate $\epsilon$ that, when given $U_i$’s $k$ sweetwords, the distinguishing attacker $A$ can gain by submitting only one online guess to $S$. However, this metric is inadequate for measuring a method’s security level when $A$ is allowed to make more than one online guess per user (e.g., when $T_1 > 1$). In addition, $\epsilon$-flat falls short of reflecting the system’s “low-hanging fruits” produced by a method, i.e. these most vulnerable honeywords that can be easily distinguished.

Accordingly, we propose two new metrics: "flatness graph" and "success-number graph". These two metrics (see Fig. 2)
measure a method’s resistance against the honeyword distinguishing attacker in the average and worse-case point of view.

**Flatness graph** plots the probability of distinguishing the real password vs. the number of sweetword login attempts per user. As shown in Fig. 2(a), a point \((x, y)\) on a curve indicates that the real password is guessed with a probability \(y\) in the first \(x\) attempts, where \(x \leq k\). In reality, it also requires that \(x \leq T_1\) (e.g., \(T_1 = 1\) or 3). Clearly, the data point \((x = 1, y)\) on our flatness curve corresponds to the \(\epsilon\)-flat metric introduced in [21], i.e., \(\epsilon = y|x = 1\). A flatness graph provides a view of the average resistance against a distinguishing attacker with varied guess numbers per user all the way to \(k\).

**Success-number graph** measures to what extent a method will produce vulnerable “low-hanging fruits” for \(A\). This graph plots the total number of successful login attempts (i.e., login with a real password) vs. the total number of failed login attempts (i.e., login with a honeyword). A point \((x, y)\) on a curve in Fig. 2(b) indicates that \(y\) real passwords are successfully distinguished from the system before the \(x\)-th honeyword login attempt occurs, where \(x \leq T_2\) (e.g., \(T_2 = 10^4\)). A \(\frac{1}{k}\)-flat method is perfect, since it produces no “low-hanging fruits” and each of the \(k\) sweetwords per user is of the same probability to be a real password.

### III. Trawling guessing attacks

The stated security goal for Juels-Rivest’s four primary honeyword-generation methods is that, when given \(U_i\)’s \(k\) sweetwords, the distinguishing attacker \(A\) can gain a maximum success rate about 1 by making an online query to \(S\) (see Table 1 of [21]). However, by using large-scale real-world datasets as listed in Table 1, we now show that these methods all fall short of its intended security goal. We first provide a review of Juels-Rivest’s methods, and then show that their methods fail to achieve the expected security (i.e., \(\frac{1}{k}\)-flat) under the weak assumptions of the attacker (i.e., a type-\(A_1\) attacker).

#### A. Review of Juels-Rivest’s methods

Juels-Rivest’s four main methods (see Table 1 of [21]) are all based on the random-replacement approach: each honeyword is produced by randomly replace parts (or whole) of the real password. Particularly, the first three are real-password related ones, i.e., the \(k - 1\) honeywords depend on the structure of the real password \(PW_i\).

**Tweaking by tail.** Their first method is to “tweak” the selected character positions of the real password \(PW_i\) to generate the \(k - 1\) honeywords. Let \(t\) (e.g., \(t = 2\) or 3) denote the desired number of positions to tweak. Juels and Rivest [21] mainly discussed the “tweaking by tail” version, and we also focus on this version. Each character in the last \(t\) positions of \(PW_i\) is replaced by a randomly-chosen character of the same type: a digit is replaced by a digit, a letter by a letter, and a special character by a special character. For instance, if \(PW_i\) is \(\text{trustno1}\) and \(t = 3\), then the sweetword list \(SW_j\) might be \(\text{trustyp0, trustmi1, trustno1, trustme5, etc.}\)

**Modeling syntax.** Their second method, inspired by [4], is to parse \(PW_i\) into tokens of consecutive characters of the same type and extract the corresponding syntax, and then honeywords are generated by replacing tokens with randomly selected values that match the syntax. For instance, the syntax \(L_7D_1\) can be abstracted from \(\text{trustno1}\), and then honeywords, such as \(\text{dfdphus3, letmein4 and kebrton9}\) can be generated.

**Hybrid.** Their third method is to combine the tweaking-tail and modelling syntax methods, aiming to gain advantages from both methods. More specially, when given \(U_i\)’s password \(PW_i\), the hybrid method first employs the modeling syntax approach to extract \(PW_i\)’s syntax and produces a seed sweetwords (including \(PW_i\)), and then generates \(b - 1\) honeywords by tweaking each seed sweetword, where \(a \cdot b \geq k\). Finally, \(k - 1\) honeywords except for \(PW_i\) are randomly selected from these \(a \cdot b - 1\) honeywords. It is recommended that \(a \approx b \approx \sqrt{k}\). For instance, assume \(k = 20\), and \(a = 4, b = 5\); the syntax \(L_7D_4\) can be abstracted from \(\text{trustno1}\), and 3 seed sweetwords \(\text{dfdphus3, letmein4 and kebrton9}\) are generated. Then, 4 new honeywords are produced for each of the 4 seed sweetwords, resulting in 20 sweetwords for \(U_i\).

**Simple model.** Their fourth method is to use real password samples as an aid to generate honeywords (see Appendix of [21]). First, a list \(L\) is built by combining large-scale real-life passwords and 8% random passwords of varying lengths. Then, a random element \(w = w_1w_2\cdots w_d\) of length \(d\) is picked from \(L\), and a honeyword \(c\) with char sequence \(w_1c_2\cdots c_d\) is generated heuristically. More specifically, for \(j = 2, 3, \cdots, d\),

- With probability 10%, replace \(w\) by a random element in \(L\), and then set \(c_j = w_j\);
- With probability 40%, replace \(w\) by a random element in \(L\) satisfying \(c_{j-1} = w_{j-1}\), and set \(c_j = w_j\);
- With probability 50%, set \(c_j = w_j\).

For fair evaluation, when implementing Juels-Rivest’s methods, we strictly follow the specifications in [21].

**Remark 1.** We note that in a passing comment, Juels and Rivest mention the possibility of using probabilistic password models (e.g., Weir et al.’s PCFG [32]) to build honeywords: “see Weir et al. [32] for a presentation of an interesting alternative model for passwords, based on probabilistic context-free grammars” [21]. However, due to the Zipf-distribution nature of passwords [28], generating decay passwords (according to a probabilistic password model or not) that are equally probable to the user’s real password is inherently impossible (see Sec. VI). Moreover, each password model has its own inherent weaknesses. Other challenges like sparsity also arise. They all make the use of probabilistic models not straightforward but rather challenging, Juels and Rivest explicitly leave it as an open question (see Section 9 of [21]): “can the password
models underlying cracking algorithms (e.g., [32]) be easily adapted for use?” In Sec. III-E, we will provide a negative answer to this question.

**Remark 2.** Also note that, besides the above four methods, there is another legacy-UI method called “Chaffing with tough nuts”. It needs to be used together with other methods, and aims to generate some tough honeywords that are much harder to crack to render A’s offline guessing work more challenging. However, as pointed out by Erguler [14], the attacker A cares cost-effectiveness: if the cracking time exceeds her acceptable time-threshold, A would stop cracking these tough nuts and simply avoid using these “tough nuts”, because common user rarely use “tough nuts”. In this regard, introducing “tough nuts” as honeywords would make the number of a user’s effective honeywords less than k-1 and thus actually decreases security. Hence, we do not consider it.

**B. Two attacking strategies**

We now present two simple yet practical honeyword distinguishing strategies (with varying tunings) against Juels-Rivest’s methods. They work on real-world password datasets, inherently different from the existing heuristic strategy (see [7], [14], [21]) that is mainly based on some specific counterexample passwords. Essentially, they make use of the fact that user-chosen passwords follow the Zipf’s law [28], and ranks the sweetwords based on some known probability distribution, either normalized within a user or not. The probability distribution can be calculated from a leaked password dataset, or based on a probabilistic password model such as Markov [24].

For fair evaluation, here we consider the attacker capabilities of A₁, i.e., a basic attacker (see Sec. II-B). In brief, A₁ has got the password hash file, knows all cryptographic algorithms and public info, but does not exploit user PII. Such an assumption of attacker capabilities is in line with Juels-Rivest’s work [21] as well as the other previous studies [7], [14].

**Top-PW attack.** This attack is simple yet effective. The distinguishing attacker A faces a password file F of n lists {SW₁, SW₂, · · · , SWₙ}, where each list consists of a set k of distinct sweetwords. Given such a multi-set of n · k sweetwords, A’s aim is to find as many real passwords as possible before making T₂ (e.g., 10⁴) failed honeyword login attempts. Note that A can only make at most T₁ (e.g., 1, 3, or 10) honeyword login attempts against each account. A simply tries these n · k sweetwords in decreasing order of probability, where the probability of each sweetword swᵢ,j (1 ≤ i ≤ n and 1 ≤ j ≤ k) comes directly from a known probability distribution Pₛ (e.g., a leaked dataset D like the 32 million Rockyou) as follows:

**Step 1.** ∀swᵢ,j ∈ F, if swᵢ,j ∈ D, set Pr(swᵢ,j) = Pₛ(swᵢ,j);
**Step 2.** ∀swᵢ,j ∈ F, if swᵢ,j ∉ D, set Pr(swᵢ,j) = 0.

We call Pₛ(·) the List password model, where ∀x ∈ D, Pₛ(x) = Count(x) / |D|, where Count(x) means the occurrence number of password x and |D| is the size of the multi-set D. We empirically evaluate this attack against the four methods in [21]. More specifically, we randomly split the Dodonew dataset into two parts of equal size, resulting in: 8.129 million Dodonew-tr and 8.129 million Dodonew-ts. For each method in [21], k – 1 honeywords are produced for each of the password in Dodonew-ts, while P₂ of the Top-PW attack comes from Dodonew-tr. As shown in Figs. 3(a)~3(d), the Top-PW attack tells apart at least 615,664 passwords from the 8,129,445 Dodonew-ts accounts protected by any of the four methods in [21]. However, according to Juels-Rivest’s original security goal, an attacker should distinguish only about 526(T₂ / (k – 1)) real passwords from Dodonew-ts. That is, in terms of the success-number metric, their methods are weaker than the claimed security by a factor of at least 1170.

We also employ the Top-PW attacking strategy to evaluate the flatness of the four methods in [21]. In brief, when given a list SWᵢ of k sweetwords for user Uᵢ, A simply tries these k sweetwords in decreasing order of probability, where each sweetword’s probability comes directly from a known probability distribution Pₛ as above. Fig. 3(e) shows that all methods are 0.35⁺-flat. In other words, there is over 35% of success rate in telling part the real password from the k=20 sweetwords with just one guess. See more details in Sec. III-D. Evidently, there is a large gap between these four methods and the expected (perfect) method.

Though this attack is effective, it is subject to an obvious defect: when identifying the next sweetword swᵢ,j to be guessed, this identifying process bears little relevance to the nature of the underlying sweetword list to which swᵢ,j belongs. For example, assume there are two sweetword lists left: SWᵢ and SWᵢ₊₁, where (1) Pr(swᵢ,j)=0.030 and all the other 19 sweetwords in SWᵢ are with a equal probability 0.029; (2) Pr(swᵢ₊₁,j)=0.028, and all the other 19 sweetwords in SWᵢ₊₁ are with a equal probability 0.001. Then, which sweetword shall be tried next, swᵢ,j or swᵢ₊₁,j? It is more likely that swᵢ₊₁,j will be the real password for SWᵢ₊₁ than swᵢ,j for SWᵢ. Thus, it is better to first try swᵢ₊₁,j against user Uᵢ₊₁. This intuition gives rise to the following more effective attacking strategy.

---

**Algorithm 1:** Distinguish the real passwords from a given password file F that is consisted of n lists of sweetwords.

- **Input:** n sweetword lists {SW₁, SW₂, · · · , SWₙ} one list per user; the threshold T₁ of honeyword login attempts per user; the threshold T₂ of honeyword login attempts for the system.
- **Output:** A vector V shows the success-number graph.

1. Initialize the vector V to be ∅;
2. Initialize the priority queue crackQueue to be ∅; /* crackQueue is ranked by the conditional probability of sweetwords. */
3. numFailure = 0, numSuccess = 0;
4. for i = 1 to n do
5. (p, sw) = getSweetword(SWᵢ); /* according to the specified attacking strategy, get the sweetword sw with the highest probability p normalized among all the un-attempted sweetwords in SWᵢ. */
6. crackQueue.Insert((p, SWᵢ));
7. while numFailure ≤ T₂ and crackQueue.empty() do
8. (p, SWᵢ, sw) = crackQueue.getFirst();
9. crackQueue.removeFirst();
10. if SWᵢ, numFailure < T₁ then
11. if success = Login((SWᵢ, id, sw)); /* use SWᵢ, id as user name sw and su as password to login as user Uᵢ. */
12. numSuccess++;
13. else SWᵢ, numFailure++, numSuccess++;
14. V.Insert((numFailure, numSuccess));
15. (p, su) = getSweetword(SWᵢ);
16. crackQueue.Insert((p, SWᵢ, sw));
19. return V.
Normalized top-PW attack. The essential difference between this strategy and the above one is that, A now tries these \( n \cdot k \) sweetwords from lists \( \{SW_1, SW_2, \ldots, SW_n\} \) in decreasing order of normalized probability. More specifically, the probability of each sweetword \( sw_{i,j} \) (\( 1 \leq i \leq n \) and \( 1 \leq j \leq k \)) comes directly from a known password distribution \( D \) (i.e., using the List password model \( P_D(\cdot) \)), but it has been normalized among each user’s own \( k \) sweetwords:

**Step 1.** \( \forall sw_{i,j} \in SW_i \), if \( sw_{i,j} \in D \), set \( Pr(sw_{i,j}) = P_D(sw_{i,j}) \);

**Step 2.** \( \forall sw_{i,j} \in SW_i \), if \( sw_{i,j} \notin D \), set \( Pr(sw_{i,j}) = 0 \);

**Step 3.** \( \forall sw_{i,j} \in SW_i \), set
\[
Pr(sw_{i,j}) = \frac{Pr(sw_{i,j})}{\sum_{i=1}^{k} Pr(sw_{i,t})}.
\]

If the system allows more than one honeyword login attempt (i.e., \( T_1 > 1 \)), after any one of the sweetwords in \( SW_i \) has been attempted, the probability of all other unattempted sweetwords in \( SW_i \) shall be normalized:

**Step 3’.** Let \( I \) denote the set of subscripts of all the sweetwords in \( SW_i \) that have already been used in login attempts. \( \forall sw_{i,j} \in SW_i \land j \notin I \), set
\[
Pr(sw_{i,j}) = \frac{Pr(sw_{i,j})}{\sum_{i=1}^{k} \sum_{j} \text{Pr}(sw_{i,j})}.
\]

After the probability of each unattempted sweetwords in \( SW_i \) has been normalized, one can sort the sweet list \( SW_i \), and identify the one with the highest probability. This corresponds to an instantiation of the function \( \text{getSweetword} \) in Line 5 of Algorithm 1. Once it has been instantiated, a run of Algorithm 1 will produce the success-number graph. Similarly, after the Steps 1~3, every sweetword in \( SW_i \) is with a (normalized) probability, and thus the function \( \text{getSweetword} \) in Line 5 of Algorithm 2 is instantiated. Finally, a run of Algorithm 2 produces the flatness graph.

Since password distributions in reality differ greatly from each other (see concrete examples in Fig. 3 of [30]), many sweetwords in the password file \( F \) do not appear in the distribution \( D \), and they will be assigned a probability 0. This causes great inaccuracies in sweetword ranking. For instance, when using Dodonew-tr as the training set (i.e., to be \( D \)) and Dodonew-ts as the test set, over 74% sweetwords in the file \( F \) (which is generated from Dodonew-ts) will be assigned a probability 0 according to Step 2. Crucially, one should not think that things that one has not yet seen are of probability 0. This is called sparsity issue in machine learning, and it can be addressed by applying the techniques of smoothing. A number of smoothing methods have been proposed for language modeling, such as Laplace and Good-Turing, and they have been widely used in password studies [24], [30]. However, they cannot be readily applied to our settings where the majority of sweetwords do not appear in the training set. We devise an “+1” smoothing method:

**Step 2’.** \( \forall sw_{i,j} \in SW_i \), if \( sw_{i,j} \notin D \), set \( Pr(sw_{i,j}) = \frac{1}{|D|+1} \).

We have experimented with the Laplace, Good-Turing and “+1” smoothing methods, and found our “+1” method though simple yet most effective. Thus, we prefer the “+1” method. Note that, “+1” method assigns the same, constant value \( \frac{1}{|D|+1} \) to all sweetwords that have not appeared, and thus it is mainly effective for ranking popular sweetwords, mainly suitable for attacking Juels-Rivest’s four methods. When the main goal is to rank unpopular sweetwords, other smoothing methods shall be designed. In Sec. III-E, we will devise a new strategy for attacking unpopular sweetwords.

**Remark 3.** It is worth noting that, the key difference between the above two attacking strategies, i.e. the (additional) Step 3 of the Normalized top-PW, will have a large effect on the success-number graph. However, this difference has no effect on the flat graph, because all of each user’s \( k \) sweetwords will be attempted and the Step 3 of the “normalized Top-PW” strategy does not change the relative rankings within each sweetword list. Both strategies (and the third one “Norm PW-model” in Sec. III-E) do not mean that “if a password appeared in a breach list \( D \), and appears in the list of honeywords, it is the actual password”. Generally, there are \( j \) \((1 \leq j \leq k)\) sweetwords appear in \( D \) and \( k - j \) sweetwords don’t, and the key difficulty lies in how best to rank these \( k \) sweetwords. This requires creative efforts: new normalization and smoothing techniques, and retooling password models to eliminate sweetword ties which have the same probability.

“Using a blacklist” is ineffective against all our proposed attacks, because: (1) Each service has vastly different password distributions—As shown in Fig. 3 of [30], every two services share less than 40% of passwords, and thus no pre-constructed blacklist will block the majority of popular passwords; (2) Users tend to circumvent the blacklist [8], [28], and new popular passwords will arise: if password is blocked, password1 will arise; if password1 is blocked, password123 (and p@ssword) will arise; (3) Blacklist is inherently unable to recognize/block PLL-based passwords, and thus is ineffective against the Type-A2 attacker.

**Remark 4.** Essentially, our two honeyword attacking strategies are effective in distinguishing popular sweetwords from unpopular ones. Thus, when more popular sweetwords are more likely to be users’ passwords, our attacking strategies will be effective. This indicates that they are particularly suitable for attacking non-password-model based methods (e.g., Juels-Rivest’s four methods) which generate a set of \( k - 1 \) honeywords that are generally less probable than the user’s original password. When password-model based methods are in place, the main goal is to rank unpopular sweetwords, and other attacking strategies shall be designed. In Sec. III-E, we will show how to revise the “Normalized top-PW attack” strategy for attacking password-model based methods.

---

**Algorithm 2:** Distinguish the real password from every list of sweetwords in a given password file \( F \).

**Input:** \( n \) sweetword lists \( \{SW_1, SW_2, \ldots, SW_n\} \), one list per user.

**Output:** A vector \( V \) shows the flatness graph.

1. Initialize the vector \( V \) to be empty;
2. for \( i = 1 \) to \( n \) do
3. \( r = 0; \) /* \( r \) counts the attempt num before a success for \( SW_i \)*/
4. while \( !SW_i \text{isEmpty()} \) do
5. \( [p, su] = \text{getSweetword}(SW_i); \) /* according to the specified attacking strategy, get sweetword \( su \) with the highest probability \( p \) among all unattempted sweetwords in \( SW_i \) */
6. success=Login(SW_i.id, su); /* use SW_i.id as user name and su as password to login as user \( U_i \). */
7. if success then
8. \( SW_i.deleteSweetword(su); \)
9. \( r++; \) /* the attempt num for \( SW_i \) increases by one. */
10. break /* if succeed, move to the list \( SW_{i+1} \). */
11. \( V.\text{Insert}(r); \)
12. return \( V \).
C. Experimental setups

When evaluating a honeyword method, the success-number graph is resulted from a run of Algorithm 1, while the flatness graph from a run of Algorithm 2. The differences in each evaluation lie in how the function getSweetword(SW_i) is instantiated: the attacking strategy (i.e., Top-PW, Norm top-PW, or Norm PW-model in Sec. IV), the training set and the test set. For each attacking strategy, there are 7 different password models (i.e., List, PCFG [32], Markov [24], their targeted versions [30], and no-training-set) for possible choice.

In most of our experiments, we use the first half of a password dataset for training (e.g., 8,129 million Dodonew-tr), and the second half for testing (e.g., 8,129 million Dodonew-ts). Now we explain why. On the one hand, the effectiveness of a machine-learning-based distinguishing algorithm depends on two factors: the algorithm itself and the training sets used. By randomly dividing a dataset into equally two parts, and using part-1 for training and part-2 for testing, we can preclude the impacts of training sets when evaluating an algorithm. On the other hand, as unending lists of real-world password lists have been disclosed (see [18], [20], [22]), the attacker can constantly improve her training set to make it as close as possible to the test set. For instance, the target system’s password distribution can be largely approximated by a leaked site with the same language, service type, password policy, etc. Actually, many sites (e.g., Yahoo [18], Phpbb and Anthem [26]) have leaked their user passwords more than once. Thus, we argue that when evaluating a honeyword generation method, it is desirable to employ a powerful yet realistic attacker and train the attacker on a training set close to the test set.

This “half-half” practice does not violate the machine-learning principle that the training set and test set shall be different. It enables the attacker knows a large portion of the knowledge of the target system’s password distribution, but not exactly the full knowledge. The underlying reason is that, according to the scale-free nature of Zipf’s law in passwords [28], a non-negligible portion of passwords in “the first half of dataset for training” will not appear in “the second half for testing”, making the training set and test set similar but not the same. For instance, among the 5.59 million distinct passwords in the test set Dodonew-ts, 4.54 million (81.22%) do not appear in the training set Dodonew-tr. Actually, this practice is also quite routine in password research (see [9], [30], [32]), but we for the first time explain why it is acceptable.

Besides experiments where the training set and test set stem from the same original distribution, we also investigate the impacts of varied training sets which come from different web services in Sec. III-E. In addition, our non-train-set model (see Sec. IV-A) does not employ any training set at all. In our figures, generally, the lower the line, the better the corresponding honeyword-method will be: Whenever a perfect method line is presented, the closer to the perfect line, the better the corresponding honeyword-method will be.

D. Our basic, trawling guessing attacks

In the above two attacking strategies, we instantiate the probabilities of sweetwords for a given password file by using the List password model \( P_D(x) \) as defined in Sec. III-B: \( \forall x \in D, P_D(x) = \frac{\text{count}(x)}{|D|} \). That is, the probability of a sweetword directly comes from the training set \( D \). We call it the basic, trawling guessing attack. In Sec. III-E, we will instantiate the probabilities of sweetwords by using much more sophisticated password models like PCFG and Markov.

Figs. 3(a)~3(d) show that, in terms of the success-number graph, the “norm Top-PW” attack strategy with smoothing performs significantly better than the no-smoothing version. Generally, it also performs much better than the “Top-PW” strategy, especially when the login number allowed is small. This suggests the critical role of smoothing in attacking honeywords. When trained on Dodonew-tr, it can tell at least 710930 (8.75%) real passwords apart from the 8,129,445 Dodonew-ts accounts protected by any of the 4 methods in [21]. This indicates that, in terms of the success-number metric, there are over \( 1352 = (710930/(7/8129445)) \) times of underestimation of the vulnerabilities in Juels-Rivest’s methods.

As said earlier, when evaluating the flatness security goal of the methods in [21], the two attacking strategies (see Sec. III-B) will essentially try the same sequences of sweetwords for each user account. Hence, they will produce the same flatness graph as shown in Fig. 3(e). Our results show that all 4 methods provide very similar security levels of flatness, with the modeling syntax being slightly better. Their similarity holds in all our later experiments, and due to space constraints, hereafter we only take the tweaking-tail method as an example.

As shown in Figs. 3(a)~3(d), no matter each user account can be attacked 1, 3, or 10 times, both attacking strategies perform rather stably. The underlying reasons are that: (1) when an account can be attacked just once (i.e., \( T_1=1 \)), both strategies can successfully identify at least 615,664 real passwords from Dodonew-ts against every method in [21]; and (2) When an account can be attacked more than once (i.e., \( T_1>1 \)), there are no more than 615,664 un-distinguished user accounts involved in the attack, and thus \( A \) will crack at most an additional \( T_2 \) real passwords as compared to \( T_1=1 \); and (3) \( 615,664 \approx T_2, \) thus 615,664+\( T_2 \approx 615,664 \). Therefore, from then on we only consider the case where \( T_1=1 \) when evaluating the success-number graphs. Note that, the setting of \( T_1=1 \) has no relevance to our evaluation of the flatness graph where \( k \) login attempts per user are always allowed.

Recall that the success-number metric measures a method’s strength against the distinguishing attacker \( A \) in the worst-case point of view, i.e., \( A \) first attacks the weakest user accounts. Take the case of “Top-PW: It” in Fig. 3(a) for example. Since 123456 is the top-1 password (1.44%) in Dodonew-tr, \( A \) will first use 123456 as the guess to test against all the 8,129,445 Dodonew-ts accounts. One might think that, 1.44% · 8,129,445 ≈ 110K successful logins shall occur before the 1st failed login occurs. However, this is not true, because 123456 can also be a honeyword for user accounts that are with a real password of the pattern 123xxx (e.g., 123123). Thus, \( A \)’s login with 123456 may fail for such accounts before reaching 110K successful logins. Fig. 3(a) shows that, to reach 110K successful logins, \( A \) fails about 100 times when using the “Norm Top-PW with smoothing” strategy, and about 1000 times when using the “Top-PW” strategy.

In the above, we mainly used the Dodonew dataset to evaluate Juels-Rives’s methods. The evaluation using the other 9 datasets show similar results, and due to space constraints, they are omitted here. The evaluation figures of three re-
maining million-sized datasets (i.e., Tianya, Rockyou and 000webhost) can be found in the companion site. Tables IV and V summarize all the evaluation results for our ten datasets. As said earlier, the choices of training and test sets follow the “half-half” practice. When allowing 10000 failed attempts, 11.03% - 16.63% accounts of the ten test sets can be successfully guessed; Tweak-tail provides the worst $\epsilon$-flatness (i.e., 0.3262), even the best method Model-syntax are 0.35 - flat, over 7 times weaker than expected in [21]. Sub-figure (f) exemplifies the correlations between the flatness and success-number metrics: (1) the upper-bound of success-number is $x + \eta \cdot \epsilon$, which means before the $x$-th failed login occurs, all the total Dodonew-ts accounts (i.e., n=8,129,445) have already been tried (since $x \ll n + \eta$, then $x + n + \eta \neq \epsilon$, being a horizontal line); and (2) the lower-bound is $\frac{1}{1+\epsilon} \cdot x$, which means before the $x$-th failed login occurs, $\mathbb{A}$ has at least tried $\frac{1}{1+\epsilon} \cdot x$ accounts, and her success number is $\epsilon \cdot \left( \frac{1}{1+\epsilon} \cdot x \right)$.

![Fig. 3. Experiment results for attacking the four methods in [21] in terms of the success-number and flatness metrics. Each method is evaluated by two attacking strategies with various tunings, trained on 50% of Dodonew (i.e., Dodonew-tr) and tested on the remaining 50% (i.e., Dodonew-ts). Whenever a perfect method line is presented, the closer to the perfect line, the better the corresponding honeyword method would produce the same success-number and flatness metrics. The experimental setups and the meaning of the bold/background-color emphasis are the same with that of Table IV.

### TABLE IV. SUCCESS-NUMBER INFORMATION (%) ABOUT EACH HONEYWORD METHOD, EVALUATED UNDER THE (TYPE-4) "NORM TOP-PW: SMOOTH, 1$^\text{st}$ STRATEGY AND $\mathbb{T}_2 = 10^4$.\dagger

<table>
<thead>
<tr>
<th></th>
<th>Tweak-tail</th>
<th>Modeling syntax</th>
<th>Hybrid</th>
<th>Simple model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tianya</td>
<td>14.41%</td>
<td>15.34%</td>
<td>14.30%</td>
<td>5.81%</td>
</tr>
<tr>
<td>Dodonew</td>
<td>10.10%</td>
<td>9.06%</td>
<td>10.46%</td>
<td>8.75%</td>
</tr>
<tr>
<td>CSDN</td>
<td>18.78%</td>
<td>15.75%</td>
<td>18.39%</td>
<td>16.32%</td>
</tr>
<tr>
<td>12306</td>
<td>9.32%</td>
<td>7.66%</td>
<td>9.17%</td>
<td>9.51%</td>
</tr>
<tr>
<td>Rockyou</td>
<td>21.03%</td>
<td>7.35%</td>
<td>14.01%</td>
<td>2.41%</td>
</tr>
<tr>
<td>000webhost</td>
<td>9.56%</td>
<td>14.33%</td>
<td>16.80%</td>
<td>4.56%</td>
</tr>
<tr>
<td>ClixSense</td>
<td>16.87%</td>
<td>5.27%</td>
<td>9.52%</td>
<td>6.08%</td>
</tr>
<tr>
<td>Yahoo</td>
<td>24.25%</td>
<td>7.61%</td>
<td>13.81%</td>
<td>16.84%</td>
</tr>
<tr>
<td>Rootkit</td>
<td>20.39%</td>
<td>12.72%</td>
<td>17.82%</td>
<td>19.57%</td>
</tr>
<tr>
<td>QNB</td>
<td>20.99%</td>
<td>20.85%</td>
<td>20.97%</td>
<td>20.48%</td>
</tr>
<tr>
<td>Average</td>
<td>16.63%</td>
<td>11.39%</td>
<td>14.59%</td>
<td>11.03%</td>
</tr>
</tbody>
</table>

\dagger A value in bold means the corresponding method perform best among four methods, while a value with background color means the worst.

### TABLE V. $\epsilon$-FLAT INFO ABOUT EACH HONEYWORD METHOD.\dagger

<table>
<thead>
<tr>
<th></th>
<th>Tweak-tail</th>
<th>Modeling syntax</th>
<th>Hybrid</th>
<th>Simple model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tianya</td>
<td>0.4368</td>
<td>0.4400</td>
<td>0.4380</td>
<td>0.4465</td>
</tr>
<tr>
<td>Dodonew</td>
<td>0.3755</td>
<td>0.3582</td>
<td>0.3796</td>
<td>0.3828</td>
</tr>
<tr>
<td>CSDN</td>
<td>0.3664</td>
<td>0.3437</td>
<td>0.3716</td>
<td>0.3978</td>
</tr>
<tr>
<td>12306</td>
<td>0.1309</td>
<td>0.1177</td>
<td>0.1297</td>
<td>0.1327</td>
</tr>
<tr>
<td>Rockyou</td>
<td>0.5498</td>
<td>0.4831</td>
<td>0.5334</td>
<td>0.5035</td>
</tr>
<tr>
<td>000webhost</td>
<td>0.3550</td>
<td>0.3587</td>
<td>0.3594</td>
<td>0.3541</td>
</tr>
<tr>
<td>ClixSense</td>
<td>0.3055</td>
<td>0.2221</td>
<td>0.2758</td>
<td>0.2943</td>
</tr>
<tr>
<td>Yahoo</td>
<td>0.2785</td>
<td>0.2080</td>
<td>0.5277</td>
<td>0.2661</td>
</tr>
<tr>
<td>Rootkit</td>
<td>0.2293</td>
<td>0.1636</td>
<td>0.2052</td>
<td>0.2210</td>
</tr>
<tr>
<td>QNB</td>
<td>0.2348</td>
<td>0.2342</td>
<td>0.2355</td>
<td>0.2313</td>
</tr>
<tr>
<td>Average</td>
<td>0.3262</td>
<td>0.2929</td>
<td>0.3200</td>
<td>0.3230</td>
</tr>
</tbody>
</table>

\dagger Both our attacking strategies in Sec. III-B would produce the same flatness graph. The experimental setups and the meaning of the bold/background-color emphasis are the same with that of Table IV.

### E. Further evaluations

#### Varying training sets.

In the above, we have evaluated Juels-Rivest’s four primary methods under two attacking strategies with various tunings. In all experiments, the training set and the test set stem from the same original distribution, and they would be of quite similar distributions. One may wonder...
what will happen when the training set and test set come from more distinct distributions. To address this issue, we fix the attacking strategy to be “normalized top-PW with smoothing” (and set $T_i=1$) and the test set to be Dodonew-ts, but with varying training sets: Dodonew-tr, Tianya and Rockyou. To understand the effects of the size of training sets, we also employ two additional training sets: Tianya_sample and Rockyou_sample, where Tianya_sample means a dataset randomly drawn from Tianya and its size equals Dodonew-tr, similarly for Rockyou_sample.

Fig. 4 shows the attacking results against the tweaking-tail method. In Fig. 4(a), when the honeyword login number allowed is small (e.g., <300), the attack trained on Tianya performs the best: As this number increases, the attack trained on Dodonew-tr performs the best. At $10^4$ failed honeyword logins, the attack trained on Dodonew-tr can distinguish 820,703 real passwords, while the least effective attack, which is trained on Rockyou_sample, can still distinguish 642,668 real passwords, 1222 times higher than against the perfect method. Fig. 4(b) reveals that: (1) even trained and tested on two distant distributions (i.e., Tianya vs Dodonew-ts), the “norm top-PW” strategy can gain 26% of success rate with just one guess, 5 times higher than Juels and Rivest originally expected (i.e., 5%); and (2) the 5 training sets show a clear hierarchy in flatness: Dodonew-tr > Tianya > Tianya_sample > Rockyou > Rockyou_sample. We also experimented on the three other methods, and get similar results.

The high success-rates of English datasets (e.g., Rockyou) against the Chinese dataset Dodonew-ts (see Fig. 4(a) and 4(b)) are because: (1) Different kinds of “low-hanging fruits” can be revealed by the List model when trained on Dodonew-tr and Rockyou, respectively; (2) there are many internationally popular passwords (e.g., digital-sequences and keyboard-patterns) in both Dodonew-tr and Rockyou (see Fig. 3 of [30]). However, when measuring “flatness” (see Fig. 4(b)), the training set Rockyou is ineffective against Dodonew-ts.

In all, our results suggest that: (1) the closer the training set is to the test set, the more effective the attack will be; (2) the larger the training set, the more effective the attack will be; (3) under quite practical attacks, all the four methods examined provide far less security than expected.

### Attacking password-model-based methods

Juels-Rivest’s four primary methods are random-replacement based: the $k$-1 honeywords for a given user are generated by randomly replacing parts (or whole) of the real password. In a passing comment, Juels and Rivest [21] mention the possibility of using probabilistic password models (e.g., Weir et al.’s PCFG [32]) to build password model based honeyword methods. They explicitly left it as an open question (see Section 9 of [21]): “can the password models underlying cracking algorithms (e.g., [32]) be easily adapted for use?” Here we provide a negative answer to this question.

Here we take the well-known PCFG password model for example. We assume that, for a given user’s password $PW_i$, the $k$-1 honeywords are produced according to the PCFG model (denoted by $P_{PCFG(D)}(\cdot)$) trained on a specified dataset $D$. Thus, the $k$-1 honeywords will be independent to $PW_i$.

In such methods, sweetwords with a higher probability generally will not be more likely to be the real password, because now the $k$-1 honeywords generated for $PW_i$ are completely dependent on the password model (e.g., $P_{PCFG(D)}(\cdot)$) but not $PW_i$: more popular events in the password model will be more likely selected as honeywords for $PW_i$.

As a result, the “Norm top-PW” strategy in Sec. III-B will not be effective. We now propose a new attacking strategy, called “Norm PW-model” strategy, by revising the “Norm top-PW” strategy. More specifically, all the settings in “Norm PW-model” strategy are the same with the “Norm top-PW” strategy, except for that (using the PCFG model for instance):

1) Add an additional Step 0 to the “Norm top-PW” attack strategy: the known password distribution $D$ (e.g., a leaked password list) is first applied to the PCFG model (we use the laplace smoothing, see [24]), and this results in a password distribution $PCFG(D)$.

2) Revise the Step 3 of “Norm top-PW” attack strategy to be: $\forall sw_{i,j} \in SW_i$, set

$$Pr(sw_{i,j}) = \frac{Pr_D(sw_{i,j})}{\sum_{t=1}^{T} Pr_{PCFG(D)}(sw_{i,t})}.$$
TABLE VII. SUCCESS NUMBER INFO AND ϵ-FLAT INFO ABOUT JUELS-RIVEST’S 4 METHODS (UNDER THE “NORM TOP-PW” STRATEGY) AND 2 PASSWORD-MODEL-BASED METHODS (UNDER THE “NORM PW-MODEL” STRATEGY): TRAINED ON DODONEW-TR, TESTED ON DODONEW-TS.

<table>
<thead>
<tr>
<th>Success number info</th>
<th>Weak tail</th>
<th>Model-syntax</th>
<th>Hybrid</th>
<th>Simple model</th>
<th>PCFG-based method</th>
<th>Markov-based method</th>
<th>Perfect method (base line)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ϵ-flat info</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Success number info</td>
<td>10.10%</td>
<td>9.06%</td>
<td>10.40%</td>
<td>8.75%</td>
<td>4.06%</td>
<td>7.22%</td>
<td>0.000%</td>
</tr>
<tr>
<td>ϵ-flat info</td>
<td>0.3755</td>
<td>0.4666</td>
<td>0.4063</td>
<td>0.3345</td>
<td>0.1761</td>
<td>0.2624</td>
<td>0.0500</td>
</tr>
</tbody>
</table>

For the Markov-model-based honeyword method, we can similarly define the “Norm PW-model” attack strategy.

As shown in Fig. 5(a), if \( A \) exploits our “Norm PW-model” strategy (trained on Dodonew-tr, tested on Dodonew-ts, and \( T_2 = 10^3 \)), honeyword-generation methods directly adapted from state-of-the-art password models are still vulnerable. More specifically, before \( A \) makes the first failed login attempt, she can successfully login 58,122 (0.71%) accounts against the PCFG-based method, and 150,459 (1.84%) accounts against the Markov-based method; before \( A \) makes the 10\(^{th}\) wrong login attempts, she can successfully login 329,957 (4.06%) accounts and 611,388 (7.52%) accounts, respectively. Fig. 5(b) reveals that both methods are 0.1761+ flat.

Fig. 5. An investigation of the effectiveness of two representative password-model-based honeyword-generation methods that are mentioned in [21]: trained on Dodonew-tr, tested on Dodonew-ts, attacking with the “Norm PW-model” attack strategy (see Sec. III-E). Both methods are vulnerable.

As summarized in Table VII, the PCFG-based method performs the best among the six methods in terms of both the success number metric and \( ϵ \)-flatness metrics. Still, in terms of the success-number metric, it is \( 6828/(329957/(T_2/(k-1))) \) times weaker than the optimal method; it is 0.1761-flat, three times weaker than the expected 0.05. It is likely that under more effective attacking strategies, the PCFG-based method will be even weaker. Naturally, an interesting question arises: is our “Norm PW-model” attacking strategy optimal for password-model based honeyword methods? Or, what’s the optimal attacking strategy for password-model based methods? We left both questions as future research.

Summary. Our real-data grounded attacks show that the four methods examined all fall short of their expected security: under Juels-Rivest’s assumed adversary model (i.e., a type-\( A_1 \) attacker), the real passwords can be distinguished with a success rate of 29.29%~32.62% by using our basic trawling-guessing attack, but not the claimed 5%, with just one guess when each user account is associated with 19 honeywords. We also provided a negative answer to the question of “can the password models underlying cracking algorithms (e.g., [32]) be easily adapted for use?” as left in [21].

IV. ADVANCED TRAWLING GUESSING ATTACKS

In the above, we have evaluated Juels-Rivest’s four primary methods under two attack strategies by using the List password model (which is the most basic/simple password model). One defect of the List model is that, it is not good at eliminating “sweetword ties” (which are of similarly low probabilities) in uncommon sweetwords. In this Section, we address this issue by exploiting two sophisticated password models (PCFG and Markov) and smoothing techniques. Particularly, both models follow the Zipf’s law (see Fig. 13(k) of [30]), and thus are good at assigning monotonically decreasing probabilities to strings/tokens to eliminate “sweetword ties”.

A. Our no-training-set password model

Before presenting our advanced-model-based experiments, we propose a special password model for honeywords. It exploits our “Norm PW-model” strategy (see Sec. III-E). As there are some sweetwords with \( c_1 \) attackers, the real passwords can be distinguished with a success rate of 29.29%~32.62% by using our basic trawling-guessing attack, but not the claimed 5%, with just one guess when each user account is associated with 19 honeywords. We also provided a negative answer to the question of “can the password models underlying cracking algorithms (e.g., [32]) be easily adapted for use?” as left in [21].

Then, we have

\[
\Pr_{PW}(x) = k \cdot \frac{\text{count}_{SW}(x)}{\text{count}_{SW} - (k-1) \cdot \frac{\text{count}_{PW}(T(x))}{\text{count}_{SW}}} \cdot \frac{1}{|T(x)|}.
\]

Since the probability of a sweetword, denoted by \( \Pr_{SW}(x) \), can generally be approximated by its empirical probability \( \text{count}_{SW}(x) / \text{count}_{SW} \), we get an estimation of \( \Pr_{PW}(x) \):

\[
\hat{\Pr}_{PW}(x) = k \cdot \text{count}_{SW}(x) / \text{count}_{SW} - (k-1) \cdot \text{count}_{SW}(T(x)) / \text{count}_{SW} \cdot \frac{1}{|T(x)|}.
\]

As there are some sweetwords with \( \hat{\Pr}_{PW}(x) < 0 \), smoothing is needed. We tested five smoothing techniques for \( \hat{\Pr}_{PW}(x) \) in Figs. 6(c) and 6(f), and results show ver5 performs the best.
training sets, with Dodonew-ts as the test set. The List password model trained on Dodonew-tr is used as the baseline.

attacks (see Fig. 6). The probability of sweetwords comes from the List model—a known (leaked) password distribution. However, every leaked password distribution is of very limited space compared to the total password space. For instance, the currently known largest one is the 3 billion Yahoo leak [18], which is still far smaller than the space (about 10^16) of passwords that comply with a typical policy: consists of three kinds of characters and with 8 ≤ len ≤ 16. Thus, one may conjecture that, under more sophisticated probabilistic password models (e.g., PCFG [32] and Markov [24]), the four methods in [21] will be even weaker. Here we follow the recommendations in [24], and use Laplace smoothing for the PCFG model and Backoff smoothing for the Markov model.

We now establish this conjecture. We design a series of attacks (see Fig. 6) against the tweaking-tail method to explore the effectiveness of various password-cracking models, including PCFG [32], Markov [24] and our above no-training-set ones. In our Markov-based attacks, all the settings are the same with the “Norm top-PW: smooth, It” in Fig. 3(a), except for that: the known password distribution D (e.g., a leaked password list) is first applied to the Markov cracking model (we use the backoff approach, see [24]), and this results in a password distribution Markov(D), then P_{Markov(D)}(\cdot) is used instead of P_D(\cdot) to assign probabilities. We similarly denote the PCFG-based password model to be P_{PCFG(D)}(\cdot) and the no-training-set model to be P_{\bar{F}}(\cdot). See the title of Fig. 6 for how these models were trained and tested.

B. Sophisticated password models

In the basic attacks in Sec. III, the probability of sweetwords comes from the List model—a known (leaked) password distribution P_D(\cdot). However, every leaked password distribution is of very limited space compared to the total password space. For instance, the currently known largest one is the 3 billion Yahoo leak [18], which is still far smaller than the space (about 10^{16}) of passwords that comply with a typical policy: consists of three kinds of characters and with 8 ≤ len ≤ 16. Thus, one may conjecture that, under more sophisticated probabilistic password models (e.g., PCFG [32] and Markov [24]), the four methods in [21] will be even weaker. Here we follow the recommendations in [24], and use Laplace smoothing for the PCFG model and Backoff smoothing for the Markov model.

We now establish this conjecture. We design a series of attacks (see Fig. 6) against the tweaking-tail method to explore the effectiveness of various password-cracking models, including PCFG [32], Markov [24] and our above no-training-set ones. In our Markov-based attacks, all the settings are the same with the “Norm top-PW: smooth, It” in Fig. 3(a), except for that: the known password distribution D (e.g., a leaked password list) is first applied to the Markov cracking model (we use the backoff approach, see [24]), and this results in a password distribution Markov(D), then P_{Markov(D)}(\cdot) is used instead of P_D(\cdot) to assign probabilities. We similarly denote the PCFG-based password model to be P_{PCFG(D)}(\cdot) and the no-training-set model to be P_{\bar{F}}(\cdot). See the title of Fig. 6 for how these models were trained and tested.

Ver1: \( P_{\bar{F}}(x) = \max\{0, P_D(x)\} \);
Ver2: \( P_{\bar{F}}(x) = \max\{\frac{1}{\text{count}_{SW}}, P_D(x)\} \);
Ver3: \( P_{\bar{F}}(x) = \max\{\frac{\text{count}_{SW}}{\text{count}_{PW}}, P_D(x)\} \);
Ver4: \( P_{\bar{F}}(x) = \frac{\text{count}_{SW}}{\text{count}_{PW}} \cdot P_D(x) \);
Ver5: \( P_{\bar{F}}(x) = \frac{\text{count}_{SW}}{\text{count}_{PW}} \).

TABLE VIII. SUCCESS-NUMBER INFO (%) ABOUT THE TWEAKING-TAIL METHOD BY USING FOUR ADVANCED PASSWORD CRACKING MODELS AND FIVE DIFFERENT TRAINING SETS, UNDER THE (TYPE-A_1) “NORMALIZED TOP-PW ATTACK: 1T” ATTACK, TEST SET = DODONEW- TS, AND T_3 = 10^4.

<table>
<thead>
<tr>
<th>Model</th>
<th>List</th>
<th>PCFG</th>
<th>Markov</th>
<th>No-training-set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dodonew-tr</td>
<td>10.10%</td>
<td>12.35%</td>
<td>14.48%</td>
<td>13.80%</td>
</tr>
<tr>
<td>Tianya</td>
<td>8.94%</td>
<td>12.22%</td>
<td>14.01%</td>
<td>10.43%</td>
</tr>
<tr>
<td>Tianya_sample</td>
<td>7.91%</td>
<td>11.55%</td>
<td>12.23%</td>
<td>10.43%</td>
</tr>
<tr>
<td>Rockyou</td>
<td>8.86%</td>
<td>10.73%</td>
<td>10.51%</td>
<td>8.87%</td>
</tr>
<tr>
<td>Rockyou_sample</td>
<td>8.05%</td>
<td>9.59%</td>
<td>9.94%</td>
<td>1.79%</td>
</tr>
</tbody>
</table>

\*A value in bold means the corresponding password model performs best. All the % are obtained by dividing the 8,129,445 Dodonew-tr accounts.

As shown in Fig. 6, and summarized in Table VIII and Table IX: (1) in general, the PCFG-based model \( P_{PCFG(D)}(\cdot) \) and the Markov-based model \( P_{Markov(D)}(\cdot) \) perform better than the List model \( P_D(\cdot) \), when given the same training set and test set; (2) the tweaking-tail method is 0.47 \( \sim \) flat (see Fig. 6(d)), 9 times weaker than expected in [21]; and (3) surprisingly, our no-training-set model \( P_{\bar{F}}(\cdot) \) (with smoothing version 5) well approaches the List model \( P_D(\cdot) \), indicating that \( A_1 \) can even perform effective attacks by just using the sweetword file \( F \) (and needs no external training sets).

Summary. Results show that Juels-Rivest’s four methods are much weaker under our more advanced trawling-guessing attackers who can exploit various state-of-the-art password models like PCFG [32] and Markov [24]. More specifically, under the adversary model (i.e., a type-\( A_1 \) attacker) assumed by Juels and Rivest, the real passwords can be distinguished with a success rate of 34.21% \sim 49.02% (see the left category of Table X), but not the claimed 5%, with just one guess when each user account is associated with 19 honeywords.
is trained on PII-Dodonew-tr and tested on PII-Dodonew-ts. P
N denote the TarMarkov model to be
of TarPCFG password model [leaked password list] is first used as the training set to the
D attackers perform much better than type-A1 ones: a clear security hierarchy appears. Sub-figure (7f) reveals that, under a Type-A2 attacker, every method in [21] is 0.568\textsuperscript{-flat}.  

V. TARGETED GUESSING ATTACKS

In the above, we have evaluated Juels-Rivest’s four main honeyword methods [21] under their assumed adversary model (i.e., a type-A1 attacker). As users love to build their (real) passwords with PII and such PII is becoming increasingly easy to be learned by A through various means (see Sec. II-A), a natural question arises: What’s the security of these methods when A is further equipped with the knowledge of user PII (i.e., under a type-A2 attacker)? We now answer this question.

We employ three targeted password-cracking models: TarPCFG [30], TarList and TarMarkov. Here TarPCFG is just the TarGuess-I in [30], while TarList and TarMarkov are PII-enriched Markov model and List model, respectively. The TarMarkov model can be converted from the Markov model by applying the type-based PII segment matching method as proposed in [30]. To accomplish the conversion, we only need to incorporate the various type-based PII tags \{N₁, \ldots, N₉; B₁, \ldots, B₁₀; A₁, A₂, A₃; \ldots\} defined in [30] into the alphabet \(\Sigma\) (e.g., \(\Sigma = \{95\) printable ASCII characters\} in [24]) of the Markov n-gram model. Then, all operations for these type-based PII tags are the same with the original characters in \(\Sigma\). Note that, the PII tag N stands for name usages, B for full name “John Smith” and N₂ for the abbrev. of full name: js=“John Smith”. Similarly, we can convert the List model into the TarList model.

In our TarPCFG-based attacks, all the settings are the same with the “Norm top-PW: smooth, It” in Fig. 3(a), except for that: the known password distribution \(D\) (e.g., a leaked password list) is first used as the training set to the TarPCFG password model [30], and this results in a password distribution \(P_{\text{TarPCFG}}(D)\). Then, \(P_{\text{TarPCFG}}(D)\) is used instead of \(P_D\) to assign sweetword probabilities. We similarly denote the TarMarkov model to be \(P_{\text{TarMarkov}}(D)\) and the TarList model to be \(P_{\text{TarList}}\). Note that, each password model is trained on PII-Dodonew-tr and tested on PII-Dodonew-ts.

Fig. 7 compares the effectiveness of type-A₁ and type-A₂ attackers against Juels-Rivest’s methods. Within each subfigure, it is evident that type-A₂ attackers are substantially more effective than type-A₁ ones. More specifically, A can guess 57.6\%-96.2\% more real passwords when \(T=10^4\), and achieve 27.2\%-65.9\% more success rates (see Table X) in terms of the \(\epsilon\)-flatness metric. Alarmingly, against every method in [21] and based on some targeted-guessing password model, PII-enriched A now can reach 56.8\%-67.9\% of success rates in telling apart the real password from 19 honeywords by making just one online guess (see the point \((x=1, y=0.5)\) in Fig. 7(f)), while the claimed (desired) security is about 5\%. In all, honeyword methods that do not consider user PII cannot withstand type-A₂ attackers that exploit the victim’s PII.

Summary. We have evaluated Juels-Rivest’s four methods under a type-A₂ attacker, a powerful yet realistic attacker, by employing three targeted password-cracking models. Our results show that, these four methods all perform poorly: the real passwords can be distinguished with a success rate of 56.8\%-67.9\% with just one online guess when each user account is associated with 19 honeywords, under the assumption that A knows some of the victim’s common PII.

VI. POTENTIAL COUNTERMEASURES

Only when we know how to attack, can we know how to protect. Our above attacking results highlight three key contributions to harden the process of generating honeywords.

An impossibility result. As said in Sec. III-A, all Juels-Rivest’s four methods essentially belong to the random re-

\[ A \]

\[ B \]

\[ C \]

\[ D \]

\[ E \]

\[ F \]

\[ G \]

\[ H \]

\[ I \]

\[ J \]

\[ K \]

\[ L \]

\[ M \]

\[ N \]

\[ O \]

\[ P \]

\[ Q \]

\[ R \]

\[ S \]

\[ T \]

\[ U \]

\[ V \]

\[ W \]

\[ X \]

\[ Y \]

\[ Z \]
placement based honeyword-generation approach: each honeyword is generated by randomly replacing parts (or whole) of the real password $PW_i$. Let $T(PW_i)$ denote the sweetword space resulted from $PW_i$, and $D$ denote the password space. We have $T(PW_i) \subseteq D$. For instance, the tweak-tail method implies $T(123456) = \{123t_1t_2t_3t_4t_5t_6 | t_1, t_2, t_3 \in \{0, 1, \ldots, 9\}\}$. Under the random-replacement approach, each of the $k - 1$ honeywords in the sweetword list $SW_i$ can be seen as randomly drawn from $T(PW_i)$, and thus they each have the same probability (i.e., $\frac{1}{|T(PW_i)|}$) to be chosen as a honeyword. If the real password $PW_i$ is also randomly drawn from $T(PW_i)$, then the random-replacement approach will be perfect.

However, users generally do not randomly choose their real passwords from $T(PW_i)$, but follow the highly skewed Zipf’s law (see [28]) in $D$: $p_r = C \cdot r^{-s} - C \cdot (r - 1)^{-s} \approx C \cdot s \cdot r^{-s+1}$, where $p_r$ denotes the probability of the $r$th popular password in $D$, and $s \in [0.15, 0.30]$ and $C \in [0.001, 0.1]$ are constants depending on $D$. It is easy to see that $\forall x_1, x_2 \in T(PW_i)$, if $x_1, x_2 \in SW_i$ and $Pr(x_1) > Pr(x_2)$, then $x_1$ is more likely to be the real password than $x_2$. Both our “Top-PW” and “Norm-PW” attacking strategies tend to rank $x_1$ higher than $x_2$, and thus will successfully pick $x_1$ out earlier.

To resist such attacks, a honeyword method should ensure that the elements in $T(PW_i)$ satisfy: $\forall x_1, x_2 \in T(PW_i)$, $Pr(x_1) = Pr(x_2)$. Unfortunately, the Zipf’s law reveals that $p_r$ is a monotonically decreasing function with $r$, which is particularly true for popular passwords (see Fig. 9 for a concrete example). Thus, when given the real password $PW_i$, it is inherently difficult to produce $|T(PW_i)|$ sweetwords with the same probability of $Pr(PW_i)$. This indicates that random-replacement methods (including Juels-Rivest’s) are impossible to achieve expected security and cannot be easily remedied.

**A counter-intuitive insight.** The above impossibility result suggests us to adopt the password model based approach. However, in Sec. III-E we have demonstrated that, as opposed to common belief (as hold in [21]), existing probabilistic password models cannot be readily used as honeyword generation methods. Now we further investigate the underlying reason.

The goodness of a password model depends on how it approximates user password behaviors. Fig 8 shows that each password model has its own weaknesses: (1) the List model is good at approximating the probability of popular passwords, yet worst at least popular passwords; (2) the PCFG model is pretty good at approximating the probability of least-popular passwords, yet worst at top popular passwords; and (3) The Markov model lies between them.

**A possible solution.** The above findings naturally lead us to propose a hybrid approach: combine varied password models to avoid individual defect. The key issue lies in how to effectively combine them. We conduct a preliminary experiment by using the simplest (i.e., linear) combination: $\frac{1}{3}$List+$\frac{1}{3}$Markov+$\frac{1}{3}$PCFG. Though simple in its nature, the hybrid password model greatly improves security: only 1 time more real passwords can be identified than the perfect method.
when $T_2=10^4$ (i.e., 1113 vs 526, see Fig. 10(a)) and being 0.11-flat (see Fig. 10(b)). This suggests a promising direction for the research community. Also note that, if there exist more effective attacking strategies, the hybrid password model might be less secure than it currently appears, and thus developing honeyword attacking theories is important future research.

VII. CONCLUSION

We have for the first time empirically evaluated the four primary honeyword-generation methods proposed by Juels and Rivest at CCS’13, by using both trawling guessing attackers with no user PII and targeted guessing attackers with some common user PII. We showed that their methods all fail to meet the expected security level by a large margin. We also provided a negative answer to the open question of “can the password models underlying cracking algorithms (e.g., [32]) be easily adapted for use?” left by Juels and Rivest.

We consider the value of our work mainly two-folds. First, our work is the first to give comprehensive, convincing quantitative evidence of how vulnerable the state-of-the-art honeyword-generation approaches are, highlighting the deceptively simple nature of honeyword research—it is not easy, but a great challenge to automatically generate honeywords that are hard to differentiate from real passwords. Second, as the limitations of the four honeyword methods examined cannot be readily addressed, our work calls for novel and creative efforts to contribute to principled honeyword-generation approaches that are significantly better than the current heuristic approaches. Considering the prevalence and catastrophic impacts of password-file leakage, we believe that achieving active and timely compromise detection is of broad interest, and our work constitutes an important step forward in this direction and will trigger interest for new honeyword research.

ACKNOWLEDGMENT

The authors are grateful to the shepherd, Prof. Adam Aviv, of our paper. We also thank the reviewers for their constructive comments. Ping Wang is the corresponding author. This research was supported by the National Natural Science Foundation of China under Grants Nos. 61472016 and 61472083, by the National Key Research and Development Plan under Grants Nos. 2016YFB0800603 and 2017YFB1200700, and by the Wallenberg Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

REFERENCES